内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
  - 局部性原理
  - 请求式分页
  - 页面置换
  - 内存保护
- 存储管理实例

常规存储管理的问题

常规存储管理方式的特征：

- 一次性：要求一个作业全部装入内存后方能运行。
- 驻留性：作业装入内存后一直驻留内存，直至结束。

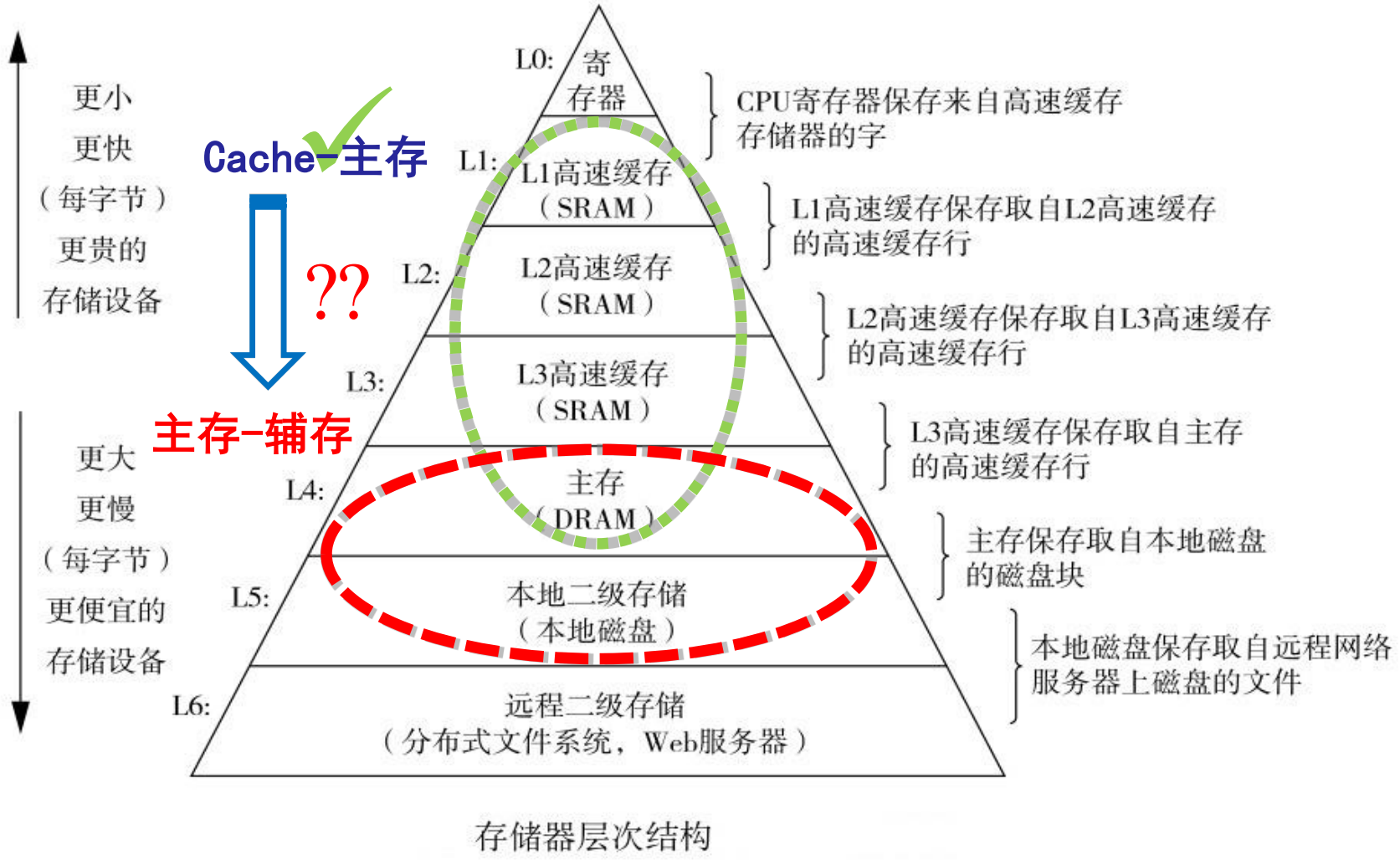
可能出现的问题：

- 有的作业很大，所需内存空间大于内存总容量，使作业无法运行。
- 有大量作业要求运行，但内存容量不足以容纳下所有作业，只能让一部分先运行，其它在外存等待。

已有解决方法：

- 增加内存容量
- 从逻辑上扩充内存容量：覆盖、对换。

再次回顾存储体系——借鉴已有方法



存储器层次结构

局部性原理

- 程序在执行时，大部分是顺序执行的指令，少部分是转移和过程调用指令。
- 过程调用的嵌套深度一般不超过5，因此执行的范围不超过这组嵌套的过程。
- 程序中存在相当多的循环结构，它们由少量指令组成，而被多次执行。
- 程序中存在相当多对一定数据结构的操作，如数组操作，往往局限在较小范围内。

局部性原理

- 指程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。还可以表现为：
  - 时间局部性，即一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内；
  - 空间局部性，即当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。

思考与讨论

- 对于一个大数组进行排序
  - 快排序
  - 堆排序

- 那种算法的局部性相对较好

快排序

- 快速排序是图灵奖得主 C. R. A. Hoare 于1960 年提出的一种划分交换排序。 它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。
- 分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题。
  - 在数据集之中，选择一个元素作为”基准” 。
  - 所有小于”基准”的元素，都移到”基准”的左边；所有大于”基准”的元素，都移到”基准”的右边。这个操作称为分区 (partition) 操作，分区操作结束后，基准元素所处的位置就是最终排序后它的位置。
  - 对”基准”左边和右边的两个子集，不断重复第一步和第二步，直到所有子集只剩下一个元素为止。

矩阵乘法的局部性

- $\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

```
// ijk版
for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum+=A[i][k]*B[k][j];
        C[i][j] = sum;
    }
```

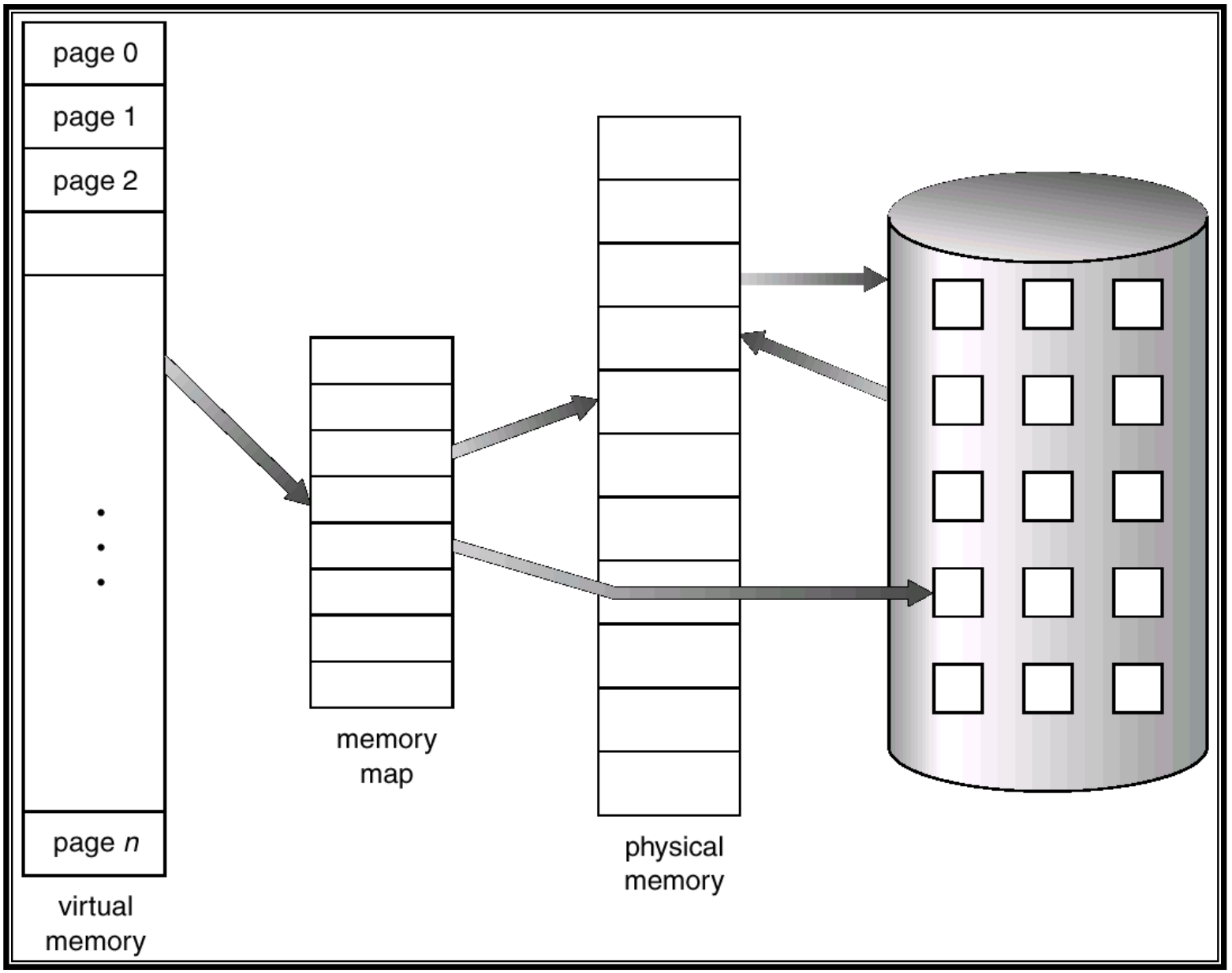
```
// kij版
for (k=0;k<n;k++)
    for (i=0;i<n;i++) {
        r=A[i][k];
        for (j=0; j<n; j++)
            C[i][j] += r*B[k][j];
    }
```

- 哪个实现的空间局部性更好？为什么？

虚拟存储的基本原理

- 在程序装入时，不必将其全部读入到内存，而只需将当前需要执行的部分页或段读入内存，就可让程序开始执行。
- 在程序执行过程中，如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页或段调入到内存，然后继续执行程序。
- 另一方面，操作系统将内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段——具有请求调入和置换功能，只需程序的一部分在内存就可执行，对于动态链接库也可以请求调入

Virtual Memory That is Larger Than Physical Memory



虚拟内存

虚拟内存是计算机系统存储管理的一种技术。它为每个进程提供了一个大的、一致的、连续可用的和私有的地址空间（一个连续完整的地址空间）。

虚拟存储提供了3个能力：

1. 给所有进程提供一致的地址空间，每个进程都认为自己是在独占使用单机系统的存储资源；
2. 保护每个进程的地址空间不被其他进程破坏，隔离了进程的地址访问；
3. 根据缓存原理，上层存储是下层存储的缓存，虚拟内存把主存作为磁盘的高速缓存，在主存和磁盘之间根据需要来回传送数据，高效地使用了主存；

虚拟存储管理的目标

- 借鉴覆盖技术：不必把程序的所有内容都放在内存中，因而能够运行比当前的空闲内存空间还要大的程序。
- 与覆盖不同：由操作系统自动完成，对程序员是透明的。

- 借鉴交换技术：能够实现进程在内存与外存之间的交换，因而获得更多的空闲内存空间。
- 与交换不同：只将进程的部分内容(更小的粒度，如分页)在内存和外存之间进行交换。

虚拟存储技术的特征

- 离散性：物理内存分配的不连续，虚拟地址空间使用的不连续（数据段和栈段之间的空闲空间，共享段和动态链接库占用的空间）
- 多次性：作业被分成多次调入内存运行。正是由于多次性，虚拟存储器才具备了逻辑上扩大内存的功能。多次性是虚拟存储器最重要的特征，其它任何存储器不具备这个特征。
- 对换性：允许在作业运行过程中进行换进、换出。换进、换出可提高内存利用率。

虚拟存储技术的特征（续）

- 虚拟性：虚拟存储器机制允许程序从逻辑的角度访问存储器，而不考虑物理内存上可用的空间数量。
  - 范围大，但占用容量不超过物理内存和外存交换区容量之和。
  - 占用容量包括：进程地址空间中的各个段，操作系统代码。

虚拟性以多次性和对换性为基础，

多次性和对换性必须以离散分配为基础。

优点、代价和限制

优点：

- 可在较小的可用(物理)内存中执行较大的用户程序；
- 可在(物理)内存中容纳更多程序并发执行；
- 不必影响编程时的程序结构（与覆盖技术比较）
- 提供给用户可用的虚拟内存空间通常大于物理内存

代价：虚拟存储量的扩大是以牺牲 CPU 处理时间以及内外存交换时间为代价。

限制：虚拟内存的最大容量主要由计算机的地址结构决定。例如 32 位机器的虚拟存储器的最大容量就是 4GB。

与Cache-主存机制的异同

相同点：

1. 出发点相同：二者都是为了提高存储系统的性能价格比而构造的分层存储体系，都力图使存储系统的性能接近高速存储器，而价格和容量接近低速存储器。

2. 原理相同：都是利用了程序运行时的局部性原理把最近常用的信息块从相对慢速而大容量的存储器调入相对高速而小容量的存储器。

与Cache-主存机制的异同

不同点：

1. 侧重点不同：cache主要解决主存与CPU的速度差异问题；虚存主要解决存储容量问题，另外还包括存储管理、主存分配和存储保护等。
2. 数据通路不同：CPU与cache和主存之间均有直接访问通路，cache不命中时可直接访问主存；而虚存所依赖的辅存与CPU之间不存在直接的数据通路，当主存不命中时只能通过调页解决，CPU最终还是要访问主存。

与Cache-主存机制的异同

3. 透明性不同：cache的管理完全由硬件完成，对系统程序员和应用程序员均透明；而虚存管理由软件（OS）和硬件共同完成，由于软件的介入，虚存对实现存储管理的系统程序员不透明，而只对应用程序员透明（段式和段页式管理对应用程序员“半透明”）。
4. 未命中时的损失不同：由于主存的存取时间是cache的存取时间的5～10倍，而主存的存取速度通常比辅存的存取速度快上千倍，故主存未命中时系统的性能损失要远大于cache未命中时的损失。

人类社会活动和生活的借鉴

- 这种管理模式在人类生活中十分常见：商品销售，家庭生活……

电脑配件销售方法	虚拟存储技术
配件存放于柜台与仓库，是一种存放体系	虚拟存储器由内存和外存共同组成，是一种存储体系
柜台取配件快，仓库取配件慢	内存访问速度快，外存访问速度慢
容量由柜台和仓库容量之和决定	容量由内存和外存容量之和决定
柜台单位容量成本高，仓库则低	内存单位容量价格高，外存则低
配件的销售也存在局部性现象	程序访问的局部性原理
柜台摆放顾客购买频率最高的配件	内存存放经常访问的数据
顾客要购买柜台没有而仓库有的配件，则出现缺货现象	存在缺页（段）现象
缺货时，需从仓库拿配件	缺页（段）时则需要请求调页（段），由缺页（段）中断机构调入
具有柜台和仓库之间货物的置换策略	有置换算法
以人力和租金便宜的仓库来换取租金昂贵的柜台空间	以CPU时间和外存空间换取宝贵的内存空间

实存管理与虚存管理

实存管理：

- 分区（Partitioning）（连续分配方式）（包括固定分区、可变分区）
- 分页（Paging）
- 分段（Segmentation）
- 段页式（Segmentation with paging）

虚存管理：

- 请求分页（Demand paging） – 主流技术
- 请求分段（Demand segmentation）
- 请求段页式（Demand SWP ）
- 交换

请求分页（段）系统

- 在分页(段)系统的基础上，增加了请求调页(段)功能、页面(段)置换功能所形成的页(段)式虚拟存储器系统。
- 允许只装入若干页(段)的用户程序和数据，便可启动运行，以后在硬件支持下通过调页(段)功能和置换页(段)功能，陆续将要运行的页面(段)调入内存，同时把暂不运行的页面(段)换到外存上，置换时以页面(段)为单位。
- 系统须设置相应的硬件支持和软件：
  - 硬件支持：请求分页(段)的页(段)表机制、缺页(段)中断机构和地址变换机构。
  - 软件：请求调页(段)功能和页(段)置换功能的软件。

请求分页与分段系统的比较

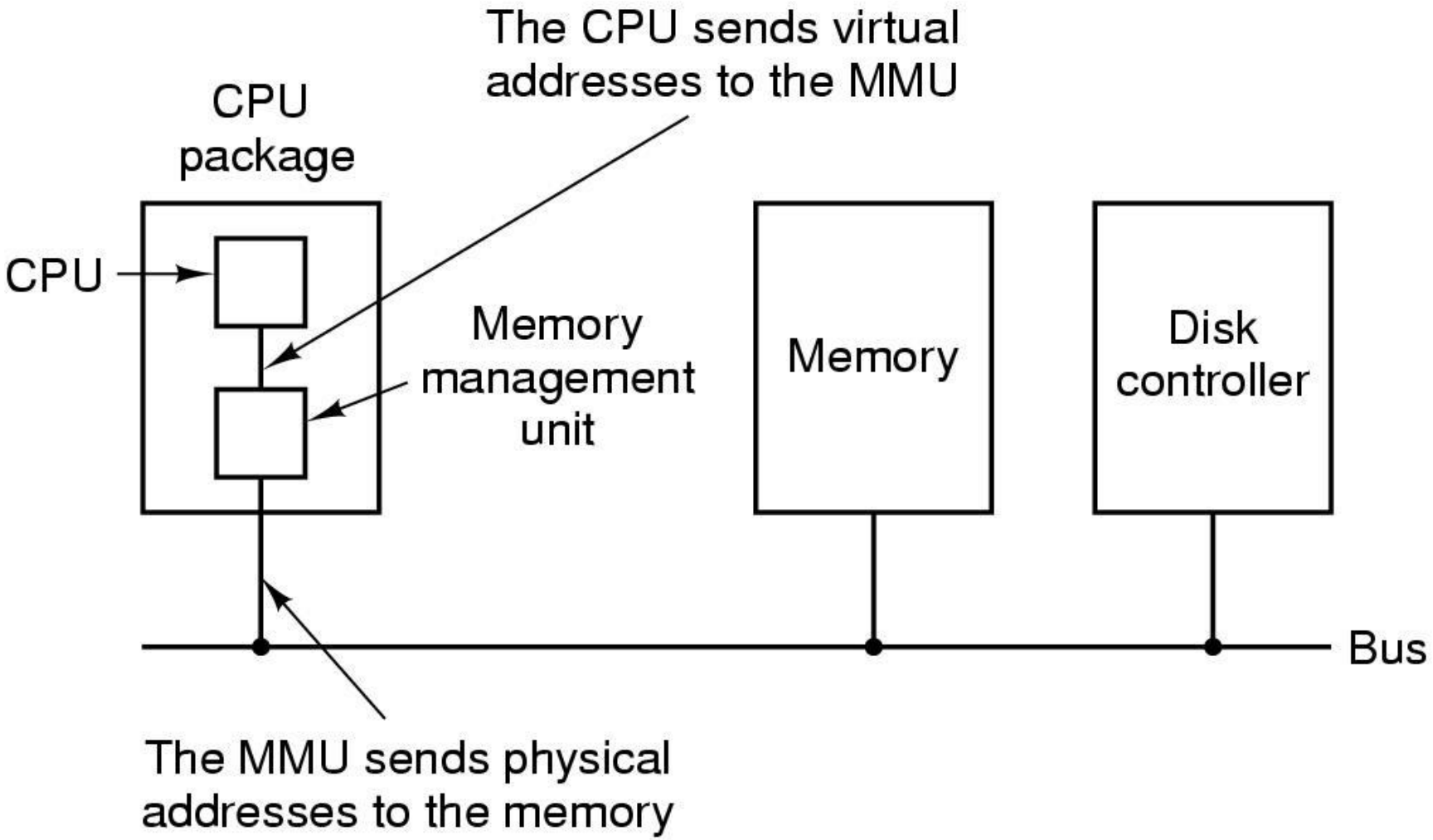
	请求分页系统	请求分段系统
基本单位	页	段
长度	固定	可变
分配方式	固定分配	可变分配
复杂度	较简单	较复杂

虚存机制要解决的关键问题

1. 地址映射问题：进程空间到虚拟存储的映射问题；
2. 调入问题：决定哪些程序和数据应被调入主存，以及调入机制。
3. 替换问题：决定哪些程序和数据应被调出主存。
4. 更新问题：确保主存与辅存的一致性。
5. 其它问题：存储保护与程序重定位等问题

在操作系统的控制下，硬件和系统软件为用户解决了上述问题，从而使应用程序的编程大大简化。

MMU


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

基本概念

1. 进程的逻辑空间（虚拟空间）

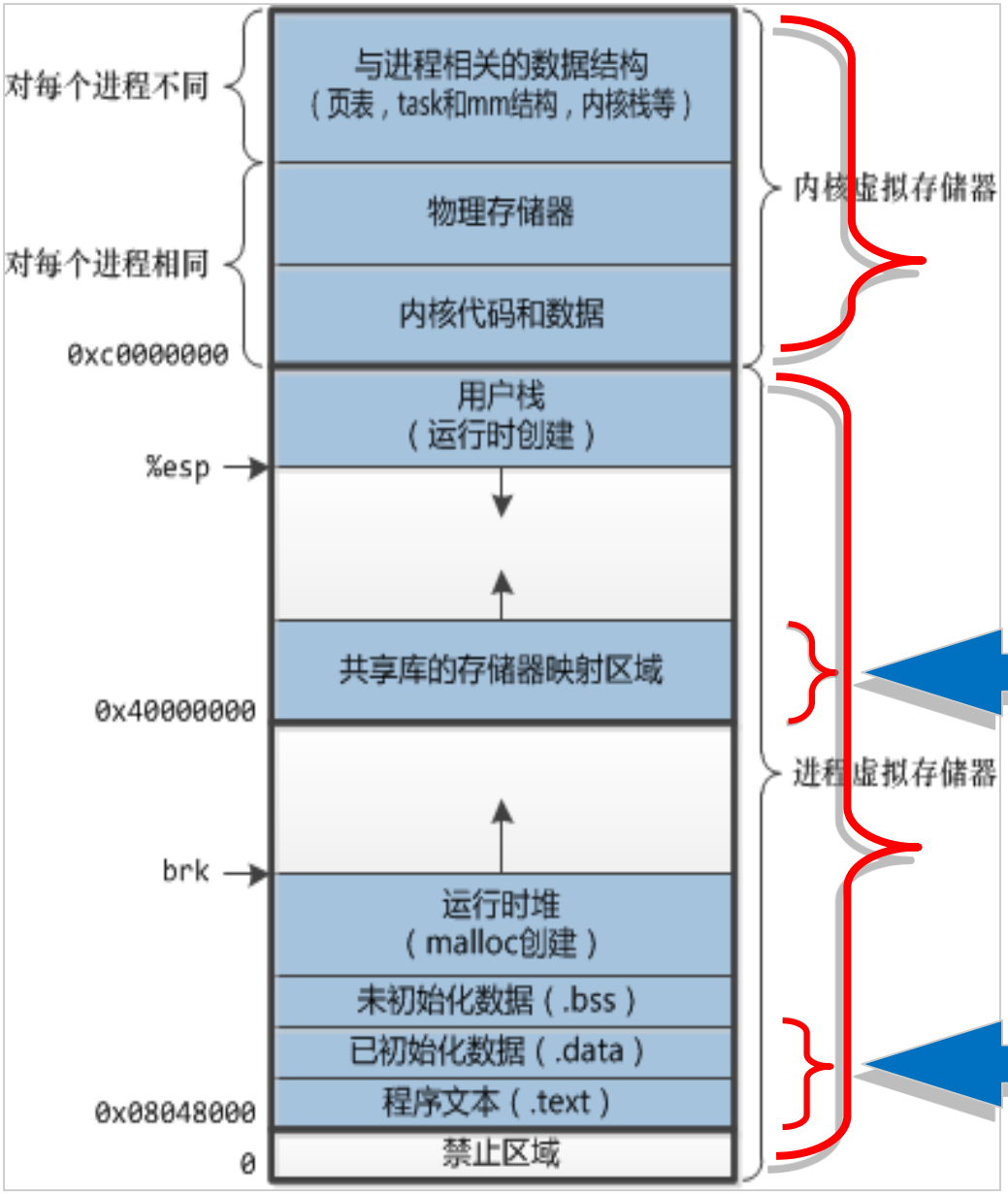
- 一个进程的逻辑空间的建立是通过链接器（Linker），将构成进程所需要的所有程序及运行所需环境，按照某种规则装配链接而形成的一种规范格式(布局)，这种格式按字节从0开始编址所形成的空间也称为该进程的逻辑地址空间。其中OS所使用的空间称为系统空间，其它部分称为用户空间。系统空间对用户空间不可见。后面只讨论用户可见部分。由于该逻辑空间并不是真实存在的，所以也称为进程的虚拟（地址）空间。

如：Hello Word进程包含Hello Word可执行程序、printf函数（所在的）共享库程序以及OS相关程序。

- 小问题：在计算机什么是“可见”？

→ 可读

进程的逻辑空间



用户不可见的进程空间，由OS使用

库文件(printf函数)

用户可见的进程逻辑空间，如：程序、数据、堆和栈

Hello可执行文件

基本概念

2. 虚拟地址空间和虚拟存储空间

- 进程的虚拟地址空间(虚拟内存空间)即为进程在内存中存放的 逻辑视图。因此，一个进程的虚拟地址空间的大小与该进程的虚拟存储空间的大小相同。且都从0开始编址有些书中也将虚拟存储空间称虚拟内存空间。
- 含有空白的虚拟地址空间称为 稀疏（sparse） 地址空间。

基本概念

3. 交换分区（交换文件）

- 是一段连续的磁盘空间（按页划分的），并且对用户不可见。它的功能就是在物理内存不够的情况下，操作系统先把内存中暂时不用的数据，存到硬盘的交换空间，腾出物理内存来让别的程序运行。
- 在Linux系统中，交换分区为Swap；在Windows系统中则以文件的形式存在（pagefile.sys）。
- 交换器的大小：交换分区的大小应当与系统物理内存（M）的大小保持线性比例关系(Linux中）：

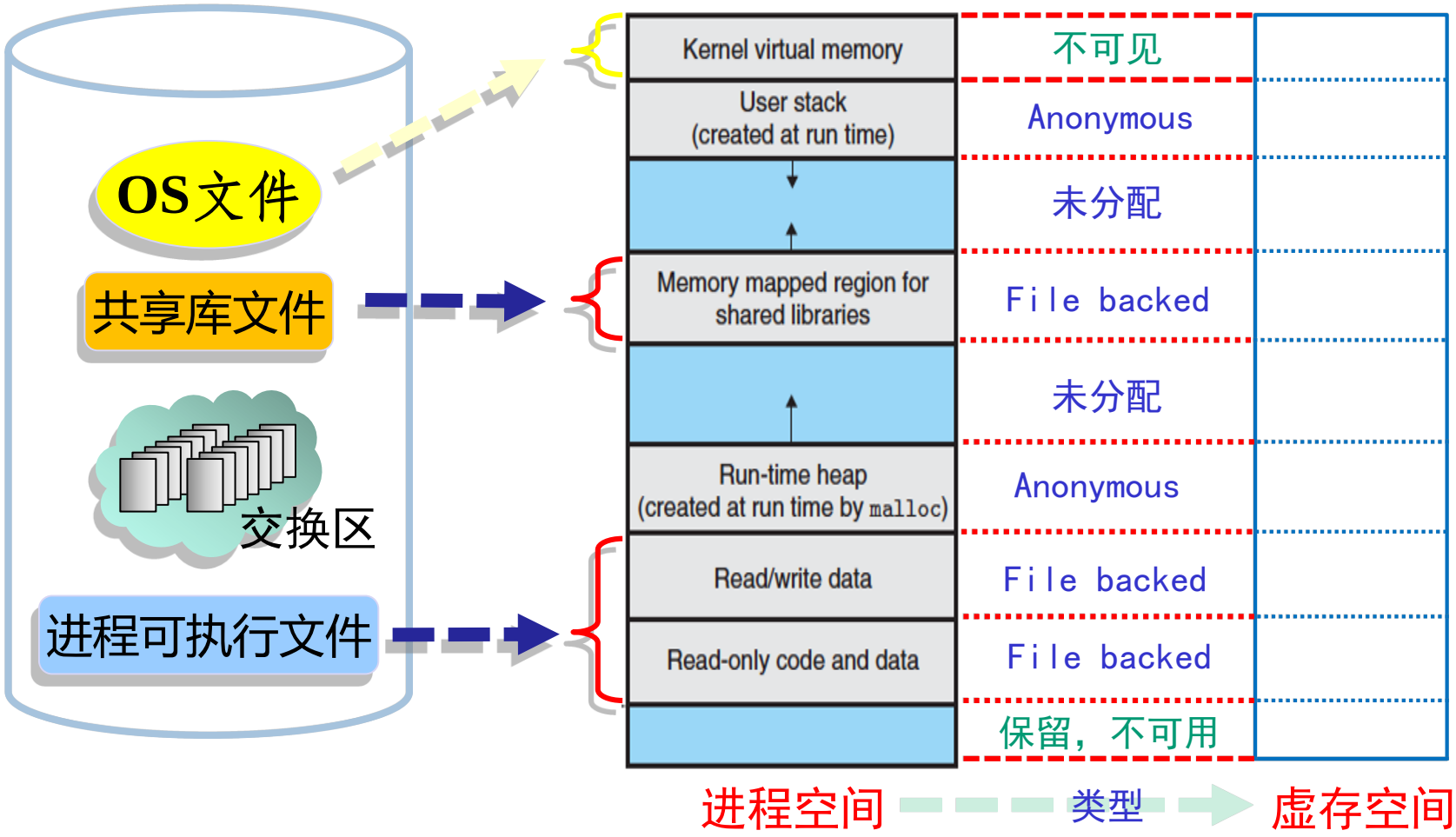
  If (M < 2G) Swap = 2*M

  else Swap = M+2

- 原因在于，系统中的物理内存越大，对于内存的负荷可能也越大。

地址映射问题（以32位Linux为例）

1. 进程空间到虚存空间的映射（进程的虚存分配）



地址映射问题

进程空间到虚存空间的映射（进程的虚存分配）

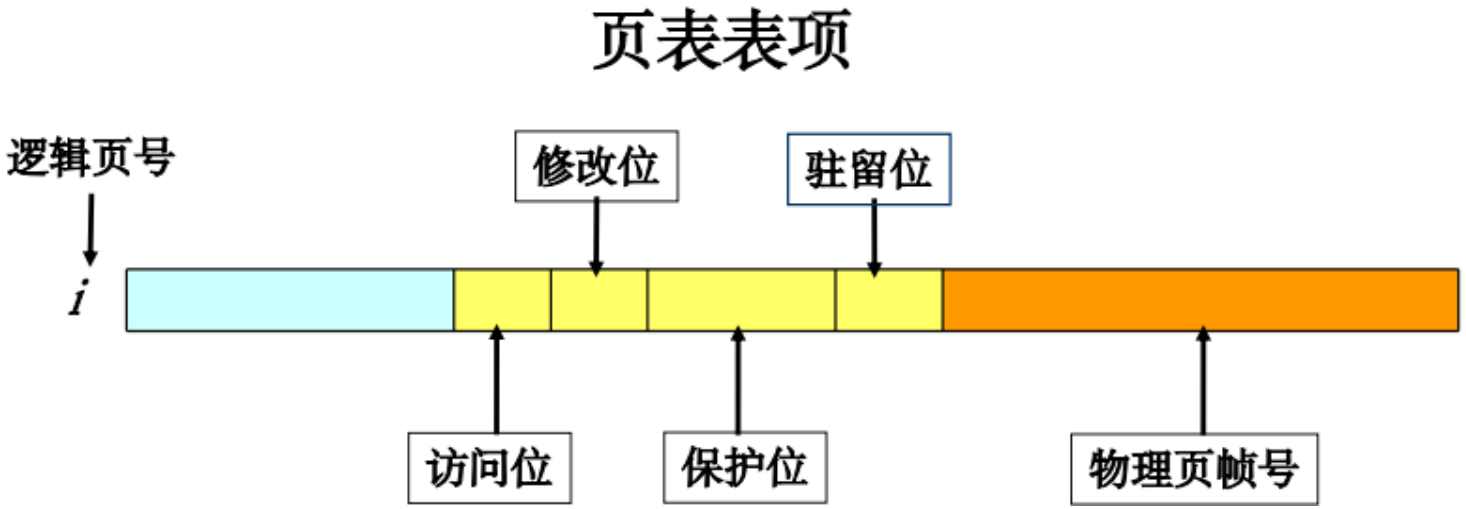
- 在程序装入时，由装载器（Loader）完成。
- 分配是以段为单位（需页对齐）进行的。
- 事实上，在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射（叫做存储器映射），等到运行到对应的程序时，才会通过缺页异常，来拷贝数据。

地址映射问题

进程空间到虚存空间的映射（进程的虚存分配）

- 用户可执行文件（如Hello World可执行文件）及共享库（如HelloWorld中调用的库文件中的printf函数）都是以文件的形式存储在磁盘中，初始时其在页表中的类型为file backed，地址为相应文件的位置。
- 堆（heap）和栈（stack）在磁盘上没有对应的文件，页表中的类型为anonymous，地址为空。
- 未分配部分没有对应的页表项，只有在申请时（如使用malloc()申请内存或用mmap()将文件映射到用户空间）才建立相应的页表项。

请求式分页管理的页表



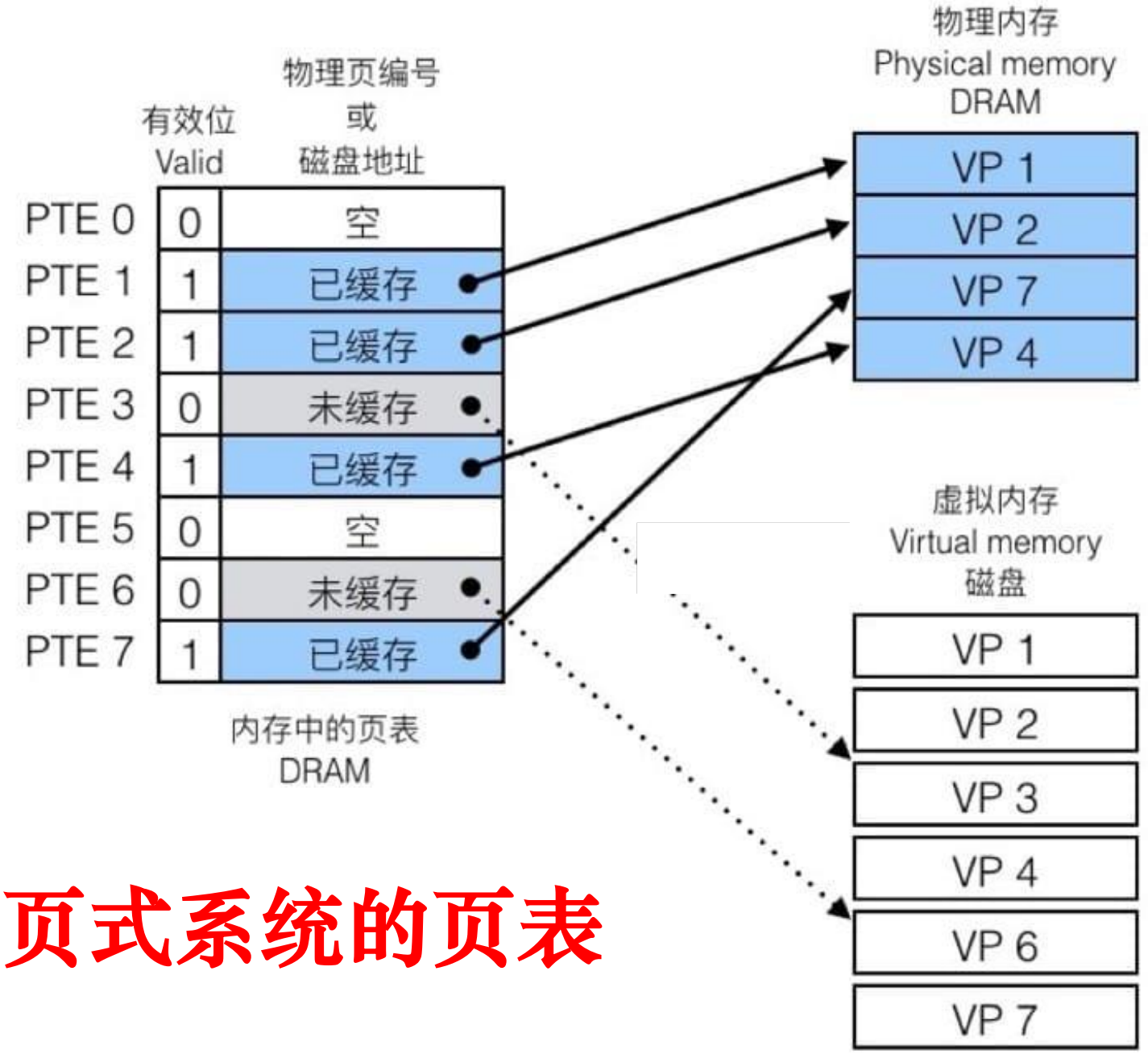
- 驻留位：1表示该页位于内存当中，0，表示该页当前还在外存当中。
- 保护位：只读、可写、可执行。
- 修改位：表明此页在内存中是否被修改过。
- 访问（统计）位：用于页面置换算法。

Intel处理器的PDE和PTE

页目录项 PDE (Page Directory Entry)

页表项 PTE (Page Table Entry)

PFN(Page Frame Number)：页框号
P(Present)：有效位
A(Accessed)：访问位
D(Dirty)：修改位
R/W(Read/Write)：只读/可读写
U/S(User/Supervisor)：用户/内核
PWT(Page Write Through)：缓存写策略
PCD(Page Cache Disable)：禁止缓存
PS(Page Size)：大页4M


物理内存
Physical memory
DRAM
物理页编号
或
磁盘地址
有效位
Valid
PTE 0
PTE 1
PTE 2
PTE 3
PTE 4
PTE 5
PTE 6
PTE 7
0
1
1
0
1
0
0
1
空
已缓存
已缓存
未缓存
已缓存
空
未缓存
已缓存
VP 1
VP 2
VP 7
VP 4
虚拟内存
Virtual memory
磁盘
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7
内存中的页表
DRAM

请求分页式系统的页表

虚拟存储器的管理

- 页面调入
- 页错误处理
- 置换问题
- 最小物理块数问题
- 分配问题

调入问题

什么程序和数据调入主存，何时调入，如何调入?

1. 什么程序和数据调入主存?
   - OS的核心部分的程序和数据;
   - 正在运行的用户进程相关的程序及数据。
2. 何时调入?
   - OS在系统启动时调入。
   - 用户程序的调入取决于调入策略。常用的调度策略有:
     1. 预调页：事先调入页面的策略。
     2. 按需调页：仅当需要时才调入页面的策略。
3. 如何调入?
   - 缺页错误处理机制。

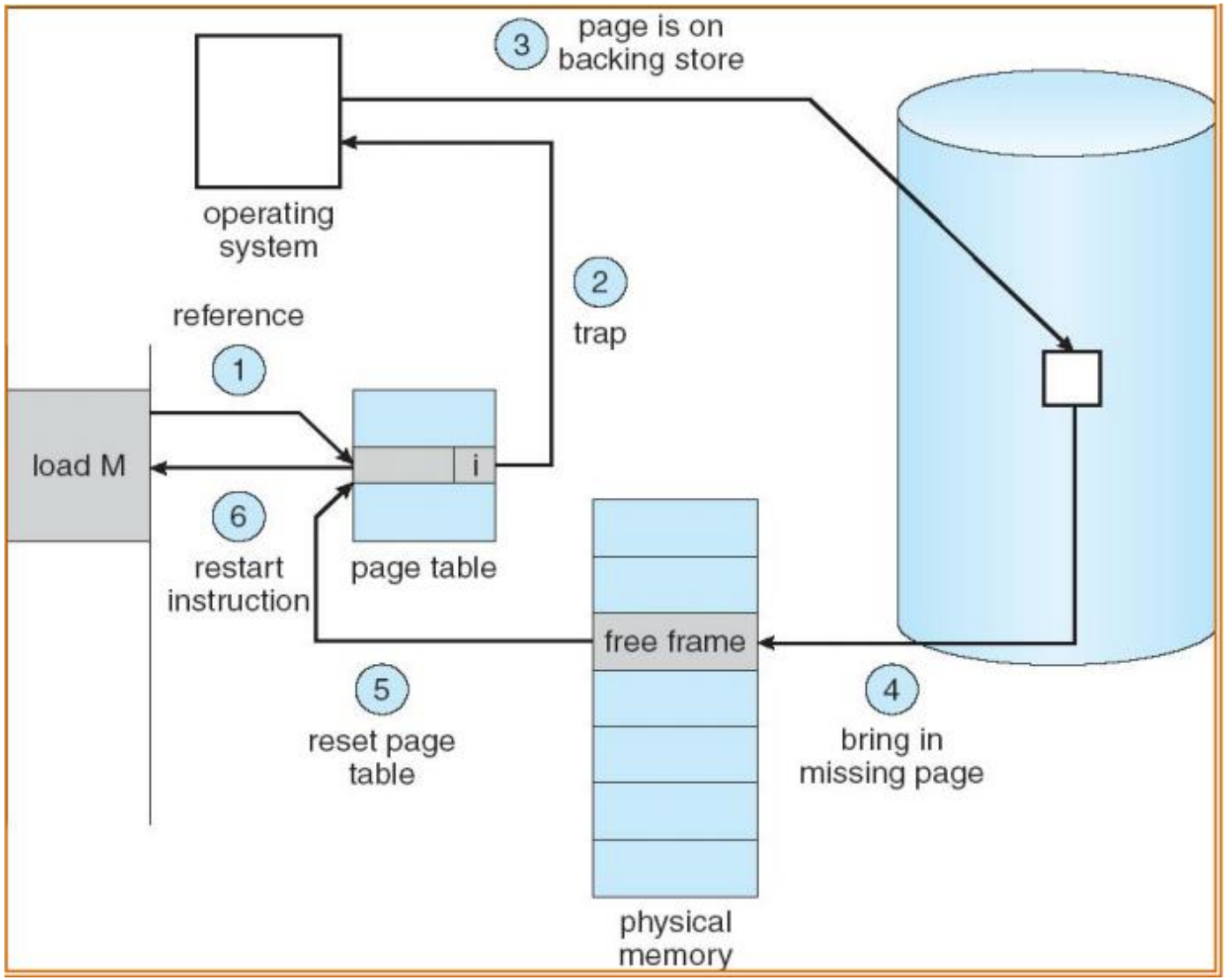
预调页（prepaging）

- 当进程开始时，所有页都在磁盘上，每个页都需要通过页错误来调入内存。预调页同时将所需要的所有页一起调入内存，从而阻止了大量的页错误。部分操作系统如 Solaris 对小文件就采取预调页调度。
- 实际应用中，可以为每个进程维护一个当前工作集合中的页的列表，如果进程在暂停之后需要重启时，根据这个列表使用预调页将所有工作集合中的页一次性调入内存。
- 预调页有时效果比较好，但成本不一定小于不使用预调页时发生页错误的成本，有很多预调页调入内存的页可能没有被使用。

按需调页（Demand Paging）

- 当且仅当需要某页时才将该页调入内存的技术称为 按需调页（demand paging），被虚拟内存系统采用。按需调页系统类似于使用交换的分页系统，进程驻留在二级存储器上（磁盘），进程执行时使用 懒惰交换（lazy swapper）换入内存。
- 按需调页需要使用备份存储，保存不在内存中的页，通常为快速磁盘，用于和内存交换页的部分空间称为交换空间（swap space）。

缺页错误（Page Fault）处理机制



缺页错误处理过程

当进程执行过程中需访问的页面不在物理存储器中时，会引发发生缺页中断，进行所需页面换入，步骤如下：

1. 陷入内核态，保存必要的信息（OS及用户进程状态相关的信息）。（现场保护）
2. 查找出来发生页面中断的虚拟页面（进程地址空间中的页面）。这个虚拟页面的信息通常会保存在一个硬件寄存器中，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析该指令，通过分析找出发生页面中断的虚拟页面。（页面定位）
3. 检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程。（权限检查）

缺页错误处理过程

4. 查找一个空闲的页框(物理内存中的页面)，如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框。（新页面调入（1））
5. 如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上[1]。（注：此时需要将页框置为忙状态，以防页框被其它进程抢占掉）（旧页面页面写回）
6. 页框“干净”后，操作系统将保持在磁盘上的页面内容复制到该页框中[2]。（新页面调入（2））

[12] 此时会引起一个写磁盘调用，发生上下文切换（在等待磁盘写的过程中让其它进程运行）。

缺页错误处理过程

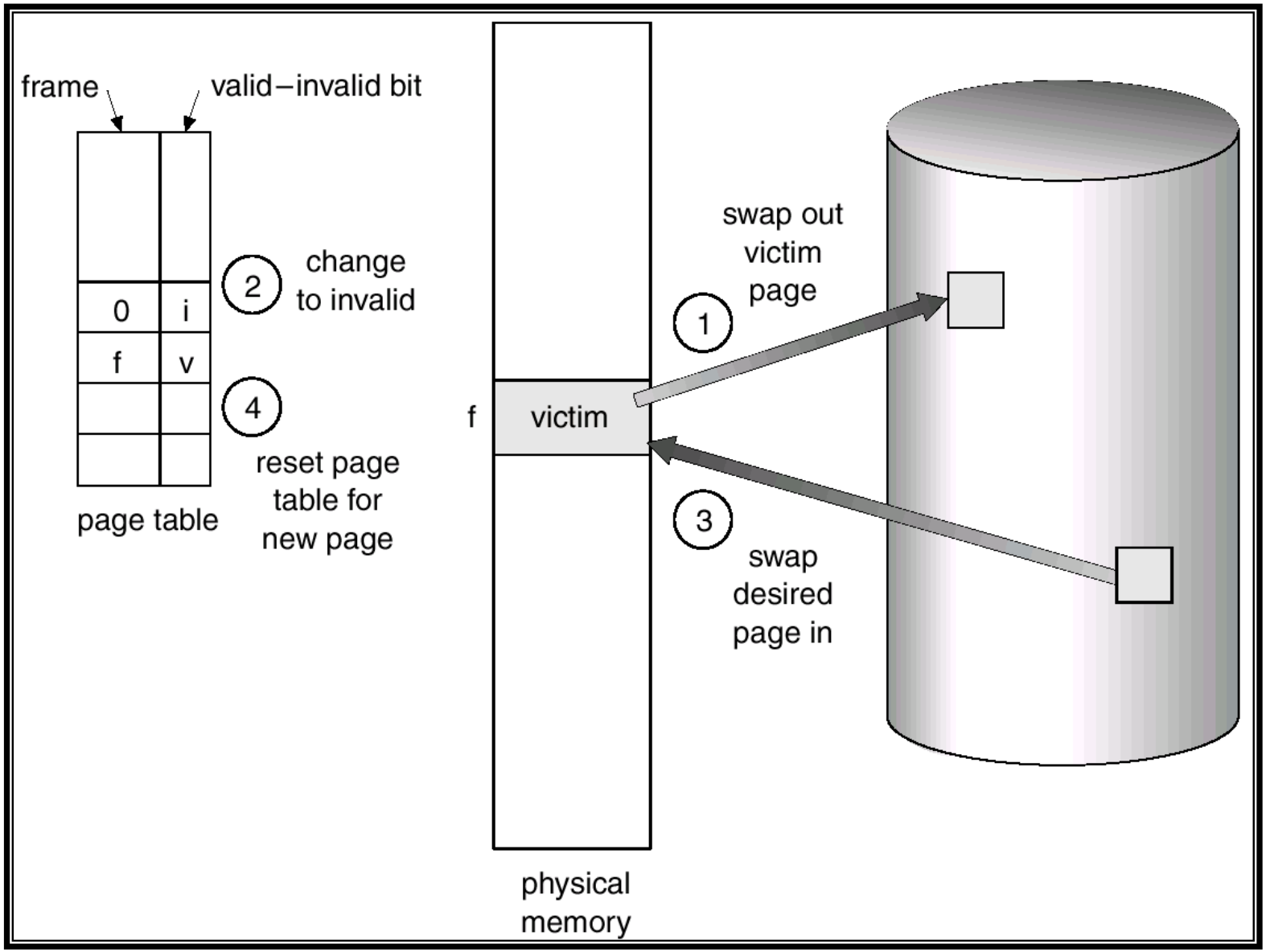
7. 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态。（更新页表）
8. 恢复缺页中断发生前的状态，将程序指针重新指向引起缺页中断的指令。（恢复现场）
9. 程序重新执行引发缺页中断的指令，进行存储访问。（继续执行）

缺页处理过程涉及了用户态和内核态之间的切换，虚拟地址和物理地址之间的转换（这个转换过程需要使用MMU和TLB）

替换问题

- 当物理内存已满，而新的页面（位于Swap区或磁盘上其它文件中）又必须调入时，必须选择适当的页面（Victim Page）换出。如何选择？——答案很简单，将造成系统运行损失最小（代价最小）的页面换出。
- 系统代价？
  - 选择的替换复杂，系统代价高；
  - 被换出的页面很快又被换入，系统代价高；
  - ……

页面置换


frame
valid–invalid bit
0
i
f
v
page table
2
change
to invalid
4
reset page
table for
new page
f
victim
1
swap out
victim
page
3
swap
desired
page in
physical
memory

页面置换策略（Replacement Strategies）

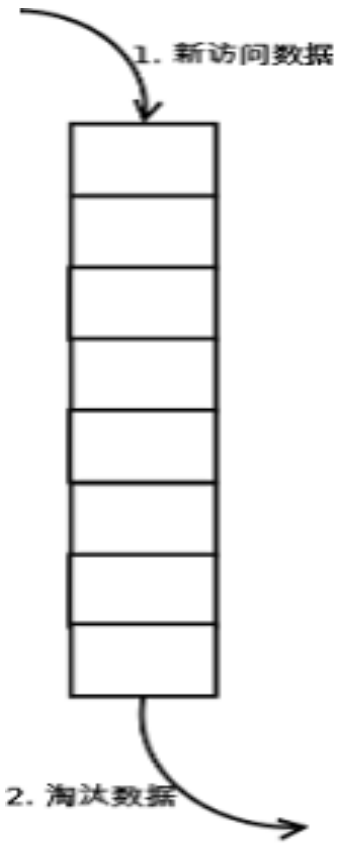
- 最优置换（Optimal）：从主存中移出永远不再需要的页面，如无这样的页面存在，则应选择最长时间不需要访问的页面。
- 二，先进先出算法（First-in, First-out）:总选择作业中在主存驻留时间最长的一页淘汰。
- 三，最近最久不用的页面置换算法（Least Recently Used Replacement）:当需要置换一页面时，选择在最近一段时间内最久不用的页面予以淘汰。
- ……

最优置换（optimal page-replacement）

- 是所有页置换算法中页错误率最低的，但它需要引用串的先验知识，因此无法被实现。
- 它会将内存中的页 P 置换掉，页 P 满足：从现在开始到未来某刻再次需要页 P，这段时间最长。也就是 OPT 算法会 置换掉未来最久不被使用的页 。
- OPT 算法通常用于比较研究，衡量其他页置换算法的效果。

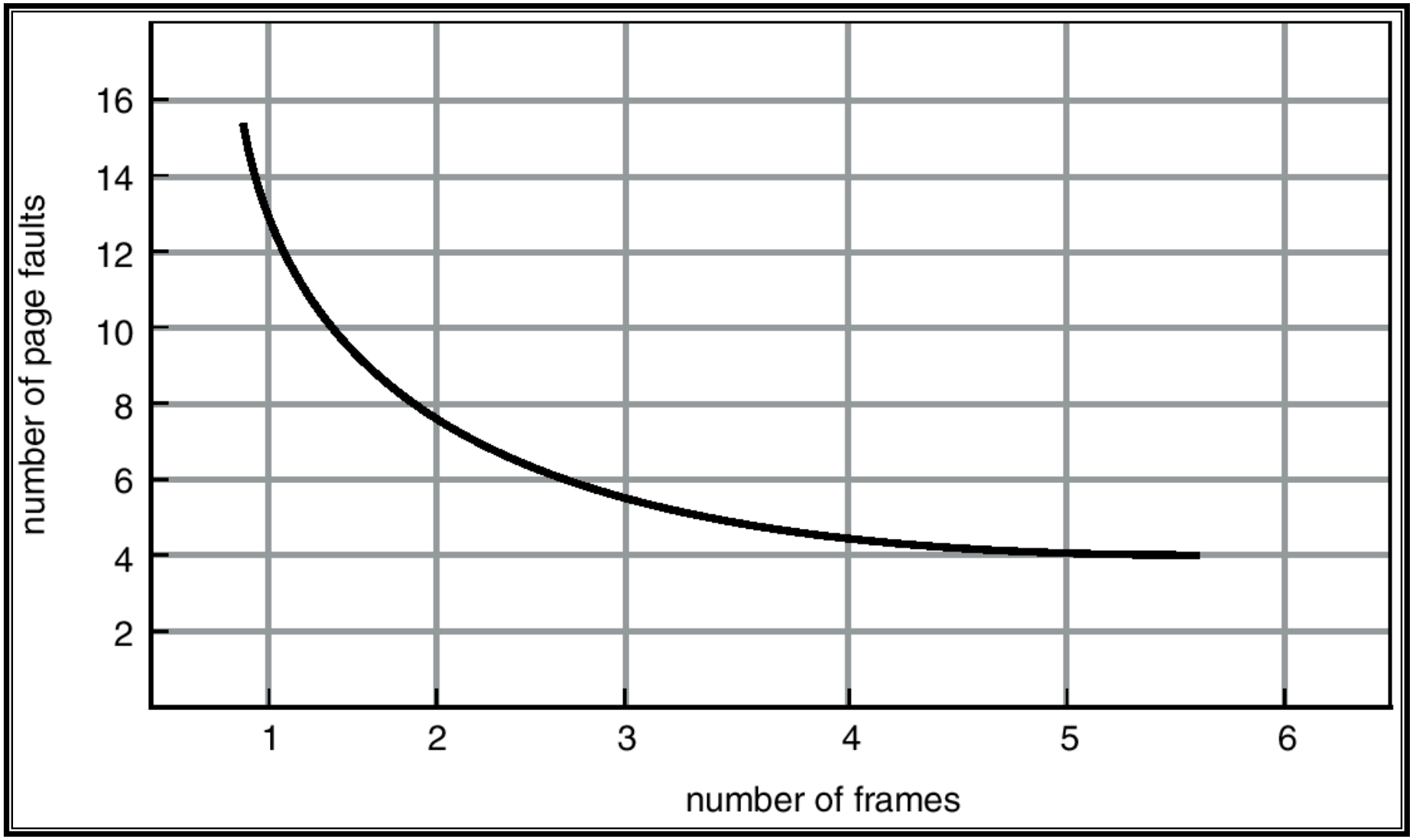
先进先出（First-in, First-out）

- 最简单的页置换算法，操作系统记录每个页被调入内存的时间，当必需置换掉某页时，选择最旧的页换出。具体实现如下：
  - 新访问的页面插入FIFO队列尾部，页面在FIFO队列中顺序移动；
  - 淘汰FIFO队列头部的页面；
- 性能较差。较早调入的页往往是经常被访问的页，这些页在FIFO算法下被反复调入和调出。并且有Belady现象。

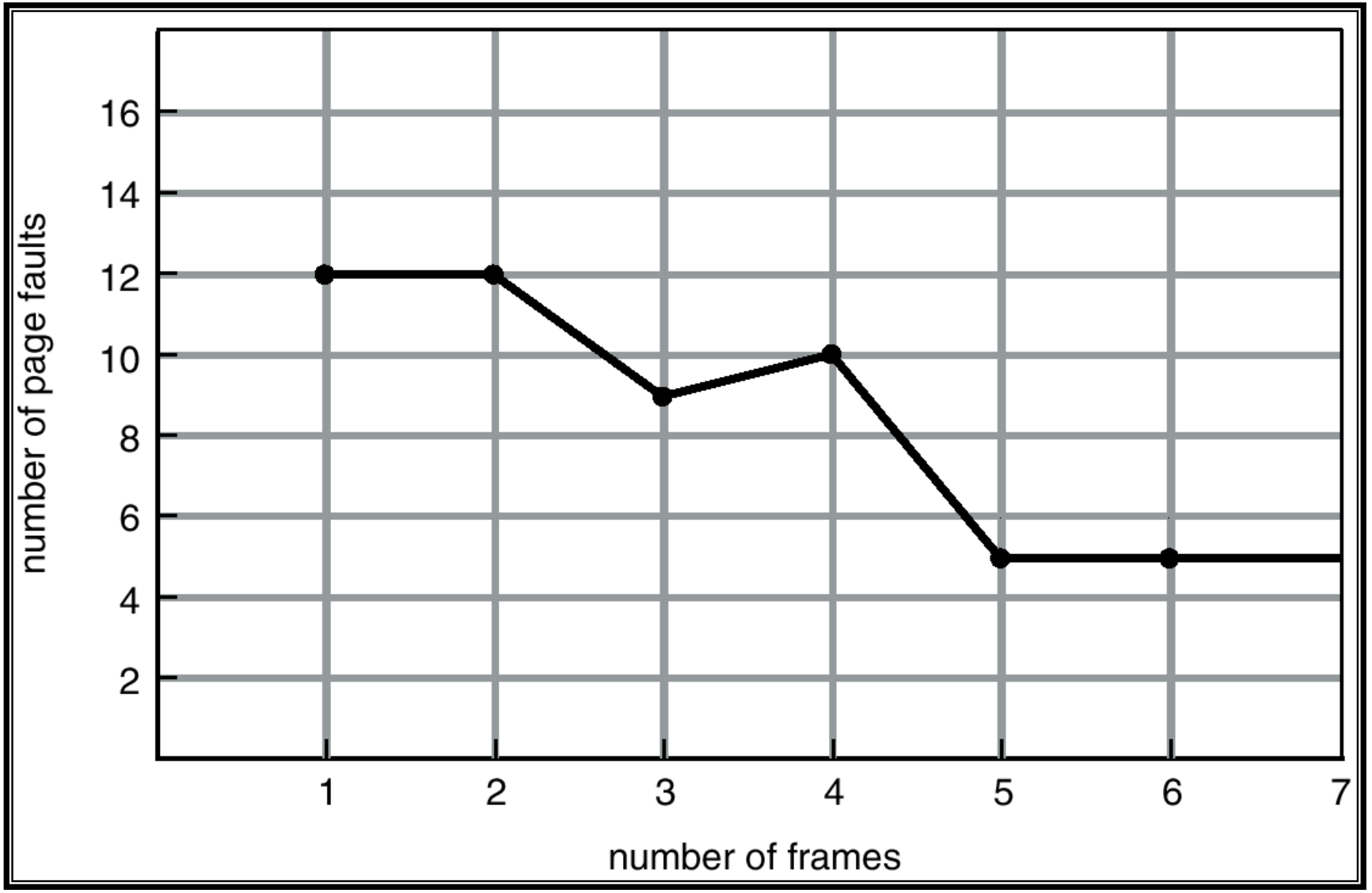


Belady现象

- 在使用FIFO算法作为缺页置换算法时，分配的缺页增多，但缺页率反而提高，这样的异常现象称为belady Anomaly。
- 虽然这种现象说明的场景是缺页置换，但在运用FIFO算法作为缓存算法时，同样也是会遇到，增加缓存容量，但缓存命中率也会下降的情况。


number of page faults
16
14
12
10
8
6
4
2
1
2
3
4
5
6
number of frames

FIFO Illustrating Belady's Anamoly



Belady现象

- 进程P有5页程序访问页的顺序为：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5；
- 如果在内存中分配3个页面，则缺页情况如下：12次访问中有缺页9次；

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	1	2	5	5	5	3	4	4
页 1		1	2	3	4	1	2	2	2	5	3	3
页 2			1	2	3	4	1	1	1	2	5	5
缺页	x	x	x	x	x	x	x	√	√	x	x	√

Belady现象

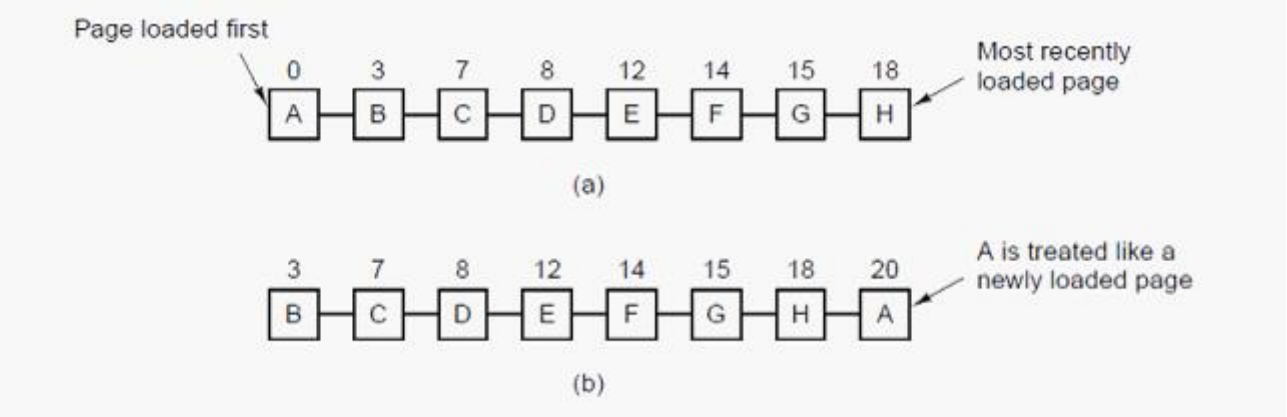
- 如果在内存中分配4个页面，则缺页情况如下：12次访问中有缺页10次；

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
页 0	1	2	3	4	4	4	5	1	2	3	4	5
页 1		1	2	3	3	3	4	5	1	2	3	4
页 2			1	2	2	2	3	4	5	1	2	3
页 3				1	1	1	2	3	4	5	1	2
缺页	x	x	x	x	√	√	x	x	x	x	x	x

改进的FIFO算法—Second Chance

其思想是“如果被淘汰的数据之前被访问过，则给其第二次机会（Second Chance）”实现：

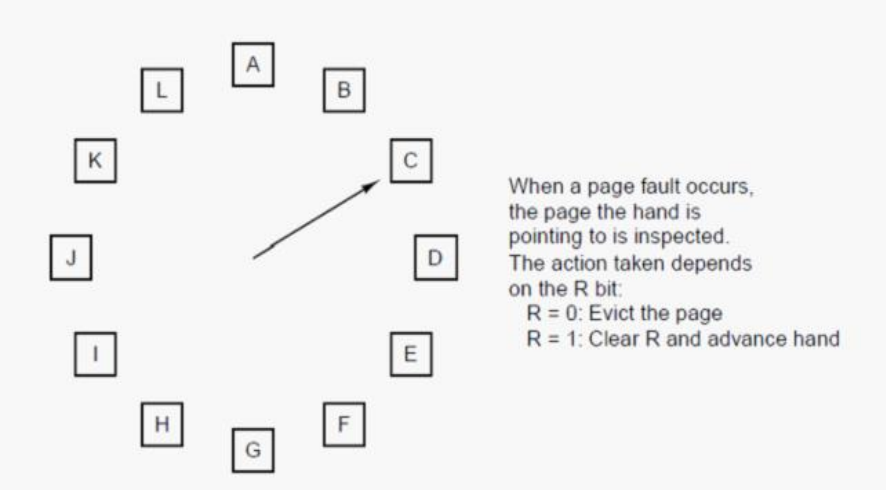
- 每个页面会增加一个访问标志位，用于标识此数据放入缓存队列后是否被再次访问过。



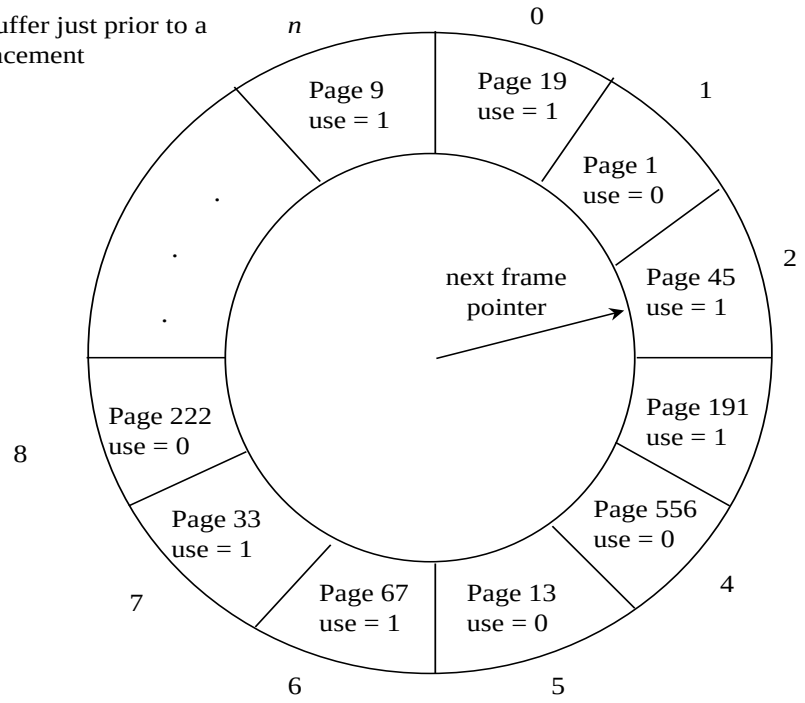
- A是FIFO队列中最旧的页面，且其放入队列后没有被再次访问，则A被立刻淘汰；否则如果放入队列后被访问过，则将A移到FIFO队列头，并且将访问标志位清除。
- 如果所有的页面都被访问过，则经过一次循环后就会按照FIFO的原则淘汰。

改进的FIFO算法— Clock

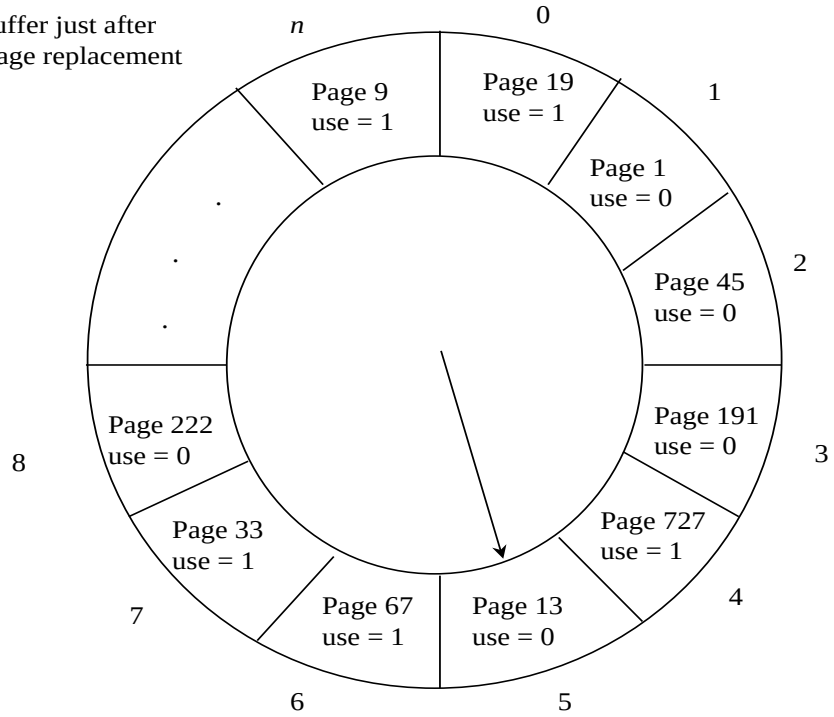
- Clock是Second Chance的改进版，也称最近未使用算法(NRU, Not Recently Used)。通过一个环形队列，避免将数据在FIFO队列中移动。算法如下：
  - 如果没有缺页错误，将相应的页面访问位置1，指针不动
  - 产生缺页错误时，当前指针指向C，如果C被访问过，则清除C的访问标志，并将指针指向D；
  - 如果C没有被访问过，则将新页面放入到C的位置，置访问标志，并将指针指向D。



State of buffer just prior to a page replacement



State of buffer just after the next page replacement



FIFO类算法对比

对比点	对比
命中率	Clock = Second Chance > FIFO
复杂度	Second Chance > Clock > FIFO
代价	Second Chance > Clock > FIFO

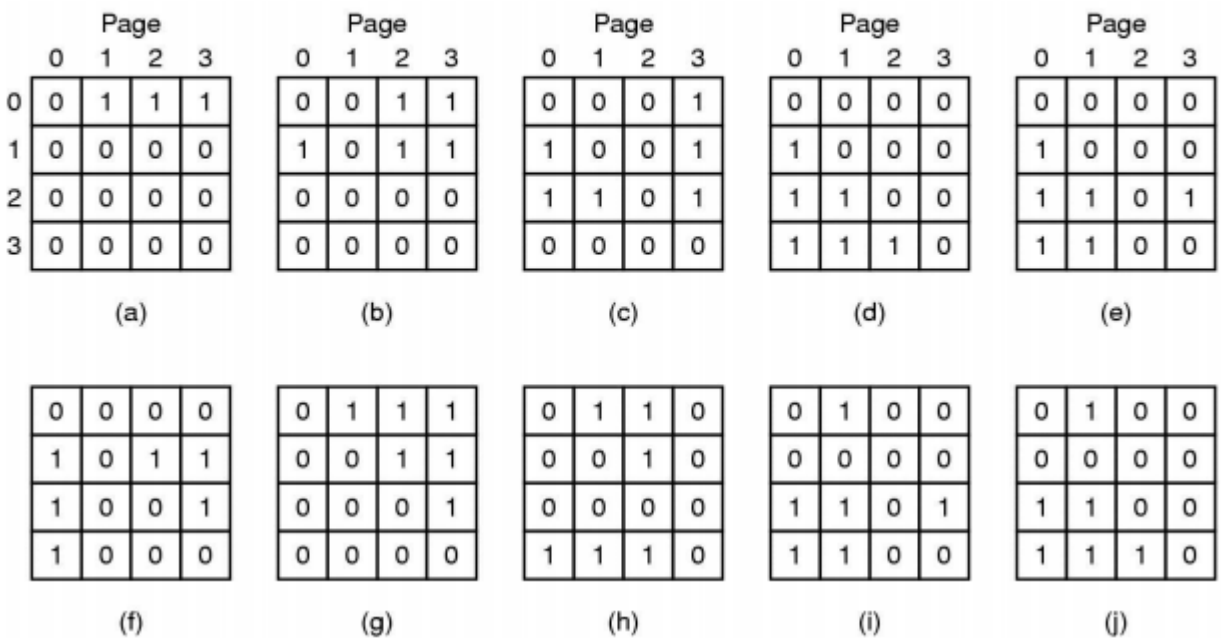
由于FIFO类算法命中率相比其他算法要低不少，因此实际应用中很少使用此类算法。

最近最少使用（Least recently used）

- LRU算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。
- 这是局部性原理的合理近似，性能接近最佳算法。但由于需要记录页面使用时间的先后关系，硬件开销太大。方法之一：
  - 设置一个特殊的栈，保存当前使用的各个页面的页面号。
  - 每当进程访问某页面时，便将该页面的页面号从栈中移出，将它压入栈顶。栈底始终是最近最久未使用页面的页面号。

LRU的一个硬件实现

- 页面访问顺序0, 1, 2, 3, 2, 1, 0, 3, 2, 3

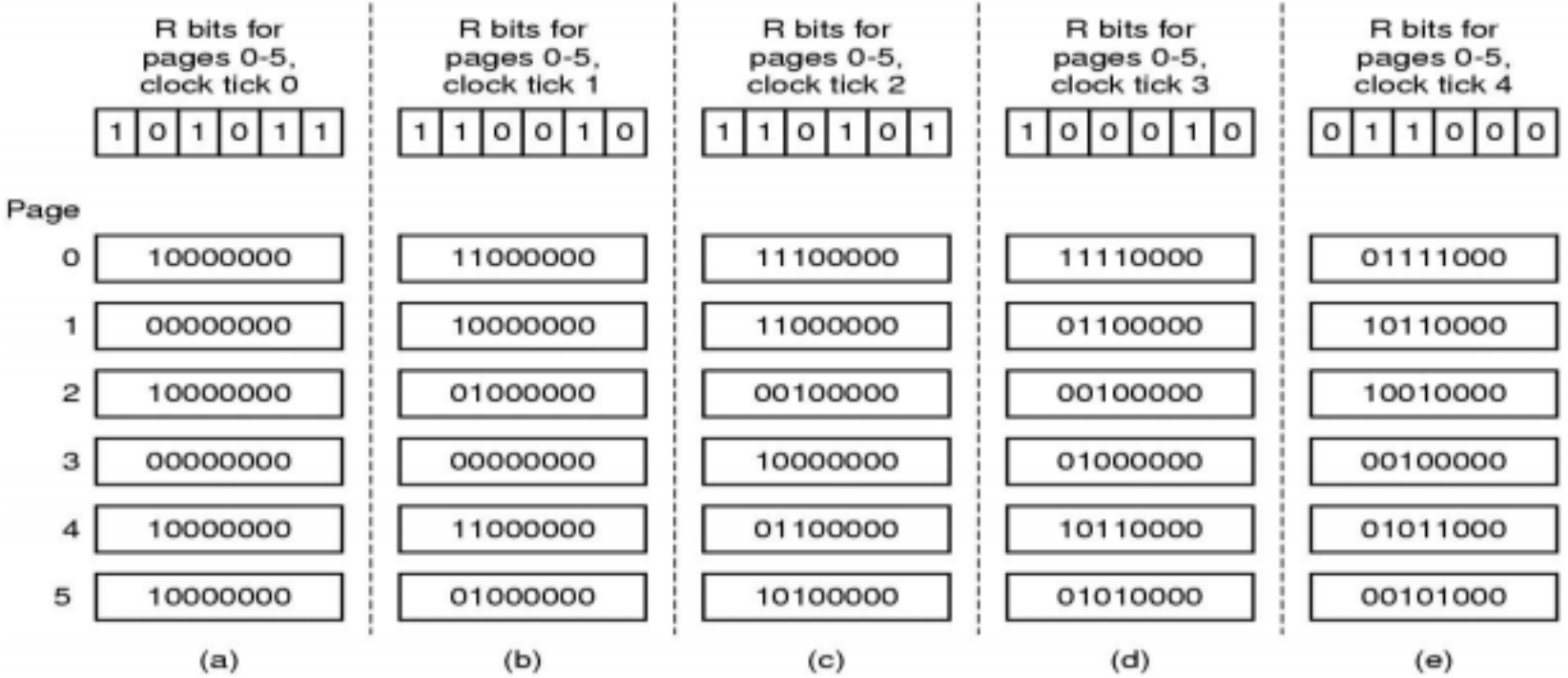


- 说明：访问第i页时先将页所在行置1，所在列清0。将行编号最小的那一页置换出去。本例中(j)处若发生页置换，则应置换第1页。

老化算法（AGING）

LRU算法开销很大，硬件很难实现。老化算法是LRU的简化，但性能接近LRU:

- 为每个页面设置一个移位寄存器，并设置一位访问位R，每隔一段时间，所有寄存器右移1位，并将R值从左移入。



算法举例(Opt, LRU, FIFO, Clock)

- 3个页框，5个工作页面

其它替换算法

页面替换算法还有很多：

- LRU类：LRU2，Two queues（2Q），Multi Queue（MQ）；
- LFU类：LFU（Least Frequently Used），LFU-Aging，LFU*-Aging，Window-LFU；
- 其它：Most Recently Used（MRU），Adaptive Replacement Cache（ARC），Working Set（WS），Working Set Clock（WSclock）

感兴趣的同学可以在网上查找相关算法并自学

替换算法总结

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

换出时的更新问题

在虚存系统中，主存作为辅存（磁盘）的高速缓存，保存了磁盘信息的副本。因此，当一个页面被换出时，为了保持主辅存信息的一致性，必要时需要信息更新：

- 若换出页面是file backed类型，且未被修改，则直接丢弃，因为磁盘上保存有相同的副本。
- 若换出页面是file backed的类型，但已被修改，则直接写回原有位置。
- 若换出页面是anonymous类型，若是第一次换出，则写入Swap区，若非第一次且未被修改则丢弃。
- 若换出页面是anonymous类型，且已被修改，则写入Swap区。

工作集与驻留集管理

前面我们从一个进程的角度，讨论了虚存管理中的相关问题，下面我们将从系统管理者的角度（OS的视角）讨论多个进程同时存在，虚存管理中的问题。

- 进程的工作集（working set）：当前正在使用的页面的集合。
- 进程的驻留集（Resident Set）：虚拟存储系统中，每个进程驻留在内存的页面集合，或进程分到的物理页框集合。
- 引入工作集的目的是依据进程在过去的一段时间内访问的页面来调整驻留集大小。

工作集

- 引入工作集的目的是依据进程在过去的一段时间内访问的页面来调整驻留集大小。
  - 工作集的定义：工作集是一个进程执行过程中所访问页面的集合，可用一个二元函数W(t, Δ)表示，其中：t是执行时刻；Δ是窗口尺寸；工作集是在[t - Δ, t]时间段内所访问的页面的集合，| W(t, Δ) | 指工作集大小即页面数目；
- 工作集大小的变化：进程开始执行后，随着访问新页面逐步建立较稳定的工作集。当内存访问的局部性区域的位置大致稳定时，工作集大小也大致稳定；局部性区域的位置改变时，工作集快速扩张和收缩过渡到下一个稳定值。

驻留集的管理

- 进程驻留集管理主要解决的问题是，系统应当为每个活跃进程分配多少个页框。
- 影响页框分配的主要因素：分配给每个活跃进程的页框数越少，同时驻留内存的活跃进程数就越多，进程调度程序能调度就绪进程的概率就越大。然而，这将导致进程发生缺页中断的概率较大；为进程分配过多的页框，并不能显著地降低其缺页中断率。

页面分配策略

固定分配策略：

- 为每个活跃进程分配固定数量的页框。即每个进程的驻留集尺寸在运行期间固定不变。为进程分配多少个页框是合理的呢？可以由系统根据进程的类型确定，也可以由编程人员或系统管理员指定。当进程执行过程中出现缺页时，只能从分给该进程的内存块中进行页面置换。

页面分配策略

可变分配策略：

- 为每个活跃进程分配的页框数在其生命周期内是可变的。即系统首先给进程分配一定数量的页框，运行期间可以增/减页框。系统可以根据进程的缺页率调整进程的驻留集。当进程的缺页率很高时，驻留集太小，需要增加页框；当缺页率一段时间内都保持很低时，可以在不会明显增加进程缺页率的前提下，回收其一部分页框，减小进程的驻留集。

页面分配策略

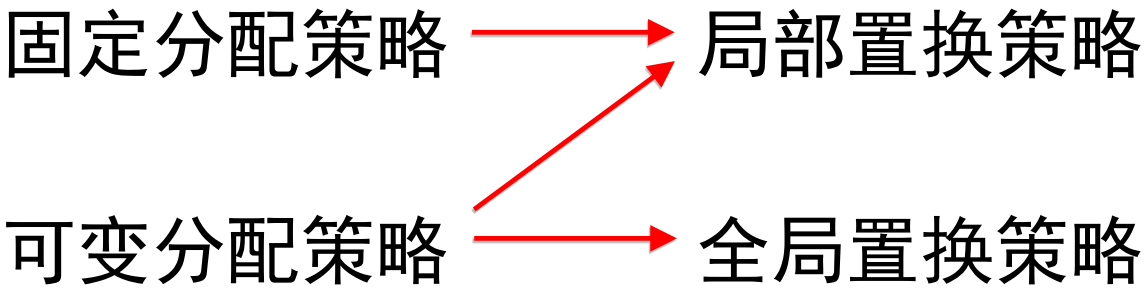
两种策略的评价：

- 可变分配策略比固定分配策略更灵活，既可以提高系统的吞吐量，又能保证内存的有效利用。
- 可变分配要求统计进程的缺页率，增加系统额外开销。而准确判断进程缺页率的高低，确定缺页率的阈值是非常困难的。
- 可变分配策略不仅需要操作系统软件专门的支持，而且，还需要处理机平台提供的硬件支持。

页面置换策略

- 当发生缺页中断且无足够的内存空间时，需要置换已有的某些（个）页面。应该解决的基本问题：
  - 当系统欲把一个页面装入内存时，应当在什么范围内判断已经没有空闲页框供分配?
  - 当指定的范围内没有空闲页框时，应当从哪里选择哪个页面移出内存?
- 局部置换：系统在进程自身的驻留集中判断当前是否存在空闲页框，并在其中进行置换。
- 全局置换策略：在整个内存空间内判断有无空闲页框，并允许从其它进程的驻留集中选择一个页面换出内存。

分配模式与置换模式的搭配



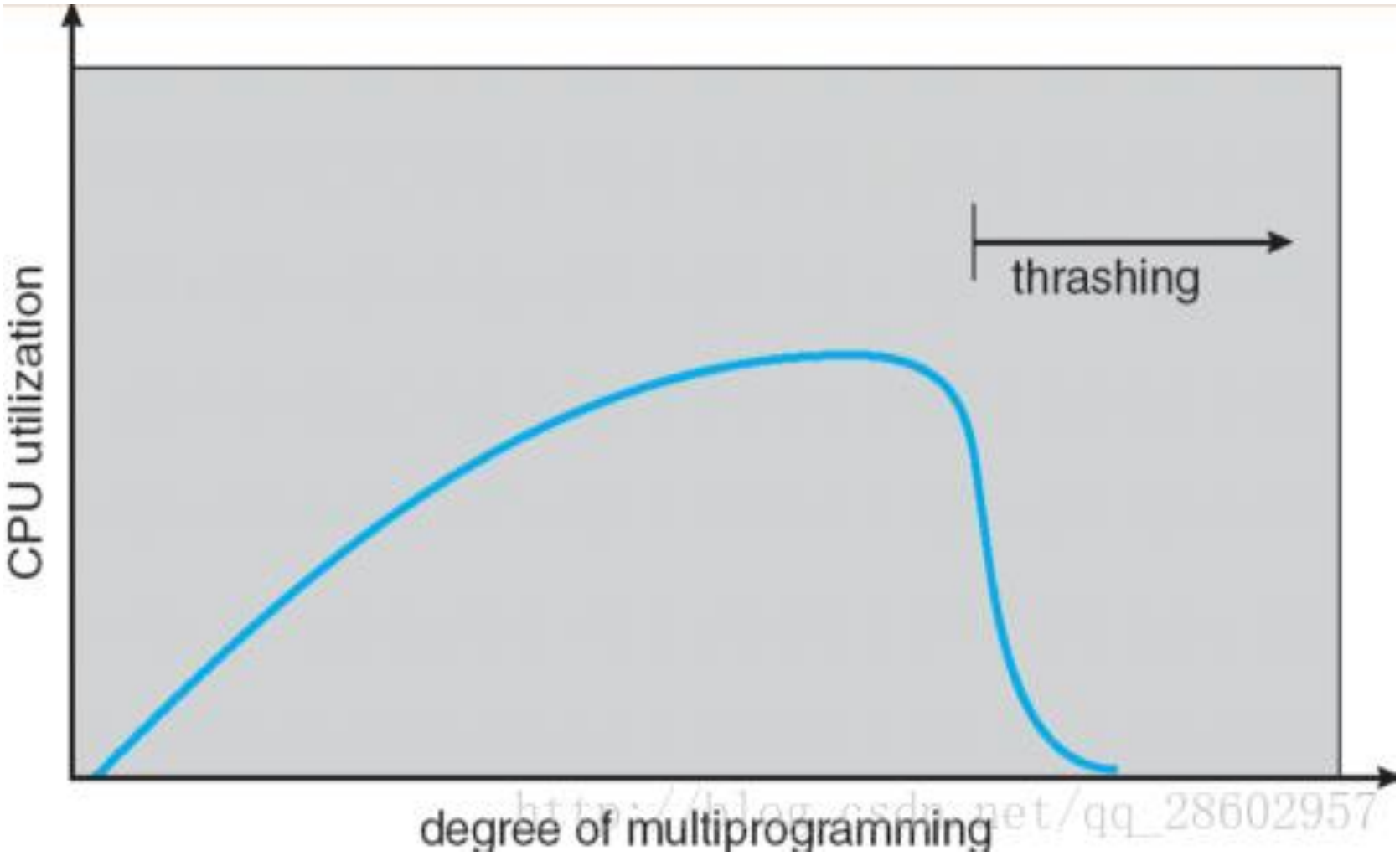
- 全局置换算法存在的一个问题是，程序无法控制自己的缺页率。一个进程在内存中的一组页面不仅取决于该进程的页面走向，而且也取决于其他进程的页面走向。因此，相同程序由于外界环境不同会造成执行上的很大差别。使用局部置换算法就不会出现这种情况，一个进程在内存中的页面仅受本进程页面走向的影响。
- 可变分配策略+局部置换策略：可增加或减少分配给每个活跃进程的页框数；当进程的页框全部用完，而需要装入一个新的页面时，系统将在该进程的当前驻留集中选择一个页面换出内存。

内存块初始分配方法

1. 等分法：为每个进程分配存储块的最简单的办法是平分，即若有m块、n个进程，则每个进程分m/n块（其值向下取整）。
2. 比例法：分给进程的块数=进程地址空间大小 / 全部进程的总地址空间 * 可用块总数
3. 优先权法：为加速高优先级进程的执行，可以给高优先级进程分较多内存。如使用比例分配法时，分给进程的块数不仅取决于程序的相对大小，而且也取决于优先级的高低。

抖动问题(thrashing)

- 随着驻留内存的进程数目增加，或者说进程并发水平(multiprogramming level)的上升，处理器利用率先是上升，然后下降。这里处理器利用率下降的原因通常称为虚拟存储器发生“抖动”，也就是：每个进程的驻留集不断减小，当驻留集小于工作集后，缺页率急剧上升频繁调页使得调页开销增大。OS要选择一个适当的进程数目，以在并发水平和缺页率之间达到一个平衡。



抖动的消除与预防

- 局部置换策略：如果一个进程出现抖动，它不能从另外的进程那里夺取内存块，从而不会引发其他进程出现抖动，使抖动局限于一个小的范围内。 然而这种方法并未消除抖动的发生。（微观层面）
- 引入工作集算法（微观）
- 预留部分页面（微观或宏观）
- 挂起若干进程：当出现CPU利用率、而磁盘I/O非常频繁的情况时，就可能因为多道程序度太高而造成抖动。为此，可挂起一个或几个进程，以便 腾出内存空间供抖动进程使用，从而消除抖动现象。（宏观）

问题：如何从宏观层面预防抖动？

负载控制

- 多道程序系统允许多个进程同时驻留内存，以提高系统吞吐量和资源利用率。然而，如果同时驻留的进程数量太多，每个进程都竞争各自需要的资源，反而会降低系统效率。
  - 如果内存中进程太多，将导致每个进程的驻留集太小，发生缺页中断的概率很大，系统发生抖动的可能性就会很大。
  - 如果在内存中保持太少的活动进程，那么所有活动进程同时处于阻塞状态的可能性就会很大，从而降低处理机的利用率。
- 负载控制主要解决系统应当保持多少个活动进程驻留在内存的问题，即控制多道程序系统的度。当内存中的活动进程数太少时，负载控制将增加新进程或激活一些挂起进程进入内存；反之，当内存中的进程数太多时，负载控制将暂时挂起一些进程，减少内存中的活动进程数。

系统负载的判断

L=S准则

- 通过调整多道程序的度，使发生两次缺页之间的平均时间（L）等于处理一次缺页所需要的平均时间（S）。即平均地，一次缺页处理完毕，再发生下一次缺页。此时，处理机的利用率将达到最大。

50%准则

- 当分页单元的利用率保持在50%左右时，处理机的利用率将达到最大。如果系统使用基于CLOCK置换算法的全局置换策略，可通过监视扫描指针的移动速率来调整系统负载。如果移动速率低于某个给定的阀值，则意味着近期页面失败的次数较少，此时可以增加驻留内存的活动进程数。如果超出阀值，则近期页面失败的次数较多，此时，系统应当减少驻留内存的活动进程数。

可挂起的进程

- 优先级最低的进程；
- 缺页进程：因为发生缺页中断的进程处于阻塞状态，暂时不需要竞争处理机的使用权。而且，挂起一个缺页进程时，挂起和激活操作需要的数据交换将消除页替换的开销；
- 最后一个被激活的进程：因为为此类进程装入的页面较少。将其挂起的开销较小；
- 驻留集最小的进程：将该类进程挂起以后，激活所需的开销较小；
- 最大的进程：挂起最大的进程将获得最多的内存空间，可以满足内存中的进程申请空闲页框的需要，使它们能尽快执行完成。

页面清除策略

- 页面清除是指，将由置换算法选出的置换页面保存到外存。页面清除策略需要决定系统何时把被置换页面写回外存。
- 若被选中的置换页面被修改过，则需要将该页面内容写回外存，然后装入新页内容。可见，此时写出一个页面与读入一个新页的操作是成对出现的，而且，写出操作先于读入操作。所以，当正在执行的进程发生缺页中断时，需要阻塞等待一个页面的写出和另一个页面的读入，这可能降低处理机的使用效率。
- 一种有效的页面清除策略是结合页缓冲（Page Buffering）技术。当发生缺页中断时，不必首先写出置换页，而是将被选中的置换页暂时保留在内存的一个缓冲区，在以后某个合适的时候将这些被置换页批量写出到外存。减少磁盘I/O的次数，提高处理机的效率。

页面缓冲算法(page buffering)

- 它是对FIFO算法的发展，通过被置换页面的缓冲，有机会找回刚被置换的页面；
- 被置换页面的选择和处理：用FIFO算法选择被置换页，把被置换的页面放入两个链表之一。即：如果页面未被修改，就将其归入到空闲页面链表的末尾，否则将其归入到已修改页面链表。

改善时间性能的途径

- 降低缺页率：缺页率越低，虚拟存储器的平均访问时间延长得越小；
- 提高外存的访问速度：外存和内存的访问时间比值越大，则达到同样的时间延长比例，所要求的缺页率就越低；
- 提高高速缓存命中率。

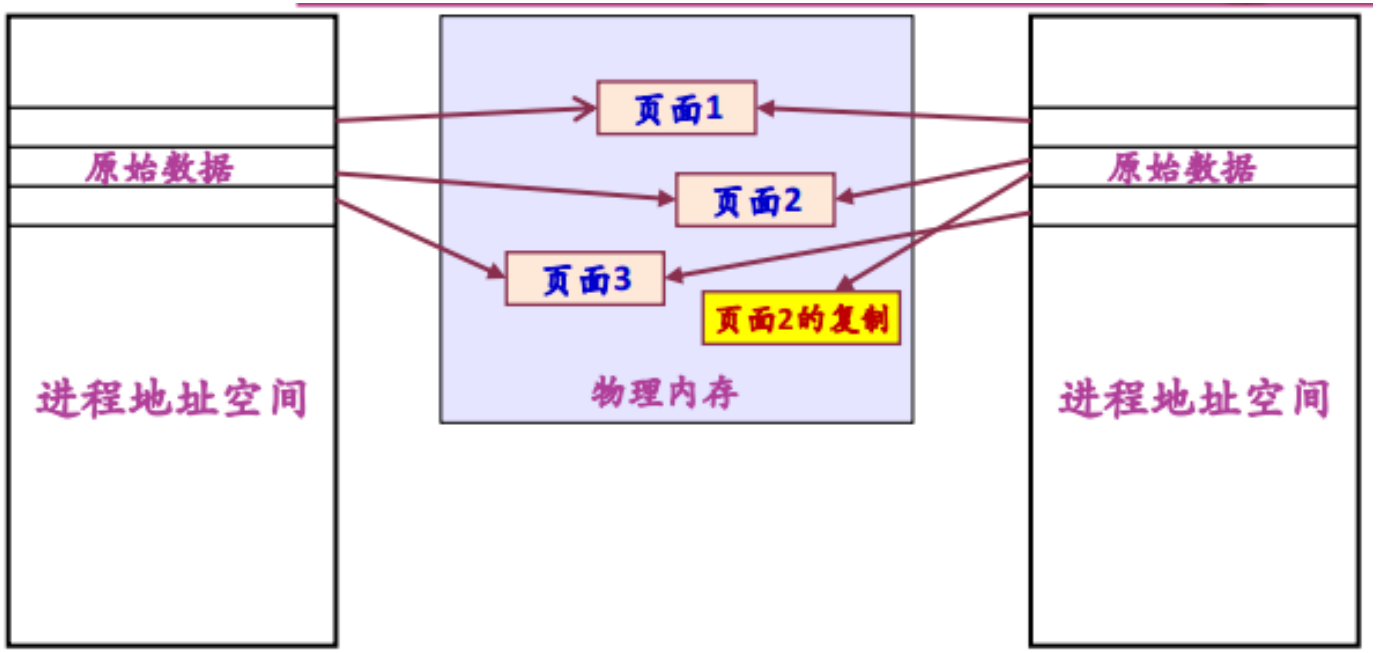
请求页式系统的性能

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

$$\begin{aligned} EAT = (1 - p) \times & \text{ memory access} \\ & + p \text{ (page fault overhead} \\ & + [\text{swap page out}] \\ & + \text{swap page in} \\ & + \text{restart overhead)} \end{aligned}$$

写时复制技术



- 两个进程共享同一块物理内存，每个页面都被标志成了写时复制。共享的物理内存中每个页面都是只读的。如果某个进程想改变某个页面时，就会与只读标记冲突，而系统在检测出页面是写时复制的，则会在内存中复制一个页面，然后进行写操作。新复制的页面对执行写操作的进程是私有的，对其他共享写时复制页面的进程是不可见的。

写时复制技术(copy-on-write)

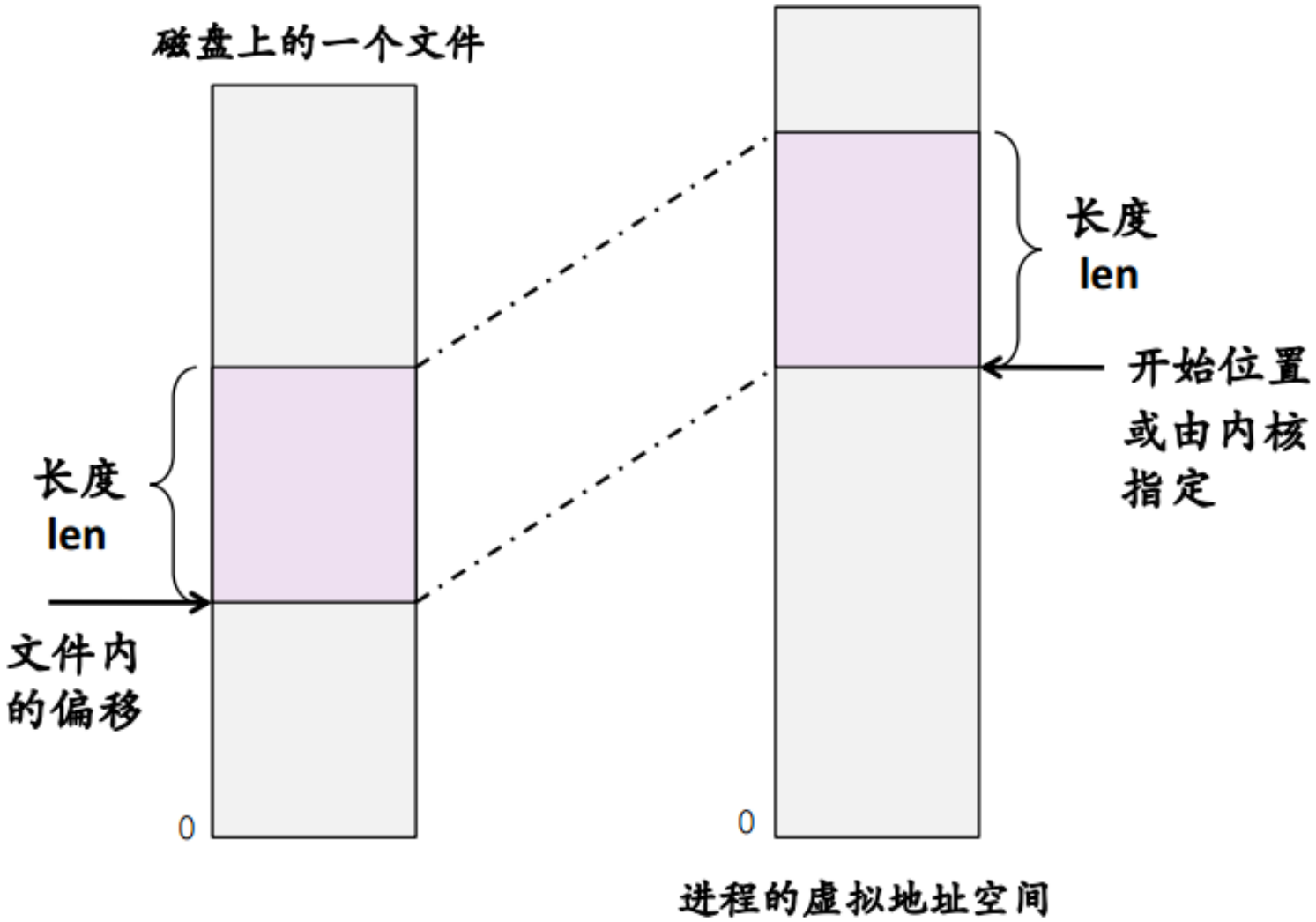
写时复制的优点

- 传统的fork()系统调用直接把所有的资源复制给新创建的进程。这种实现过于简单而效率低下，因为它拷贝的数据也许并不共享，如果新进程打算立即执行一个新的映像，那么所有的拷贝都将前功尽弃。Linux的fork()使用写时拷贝(copy-on-write)实现，它可以推迟甚至免除拷贝数据的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会复制，从而使各个进程都拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行。

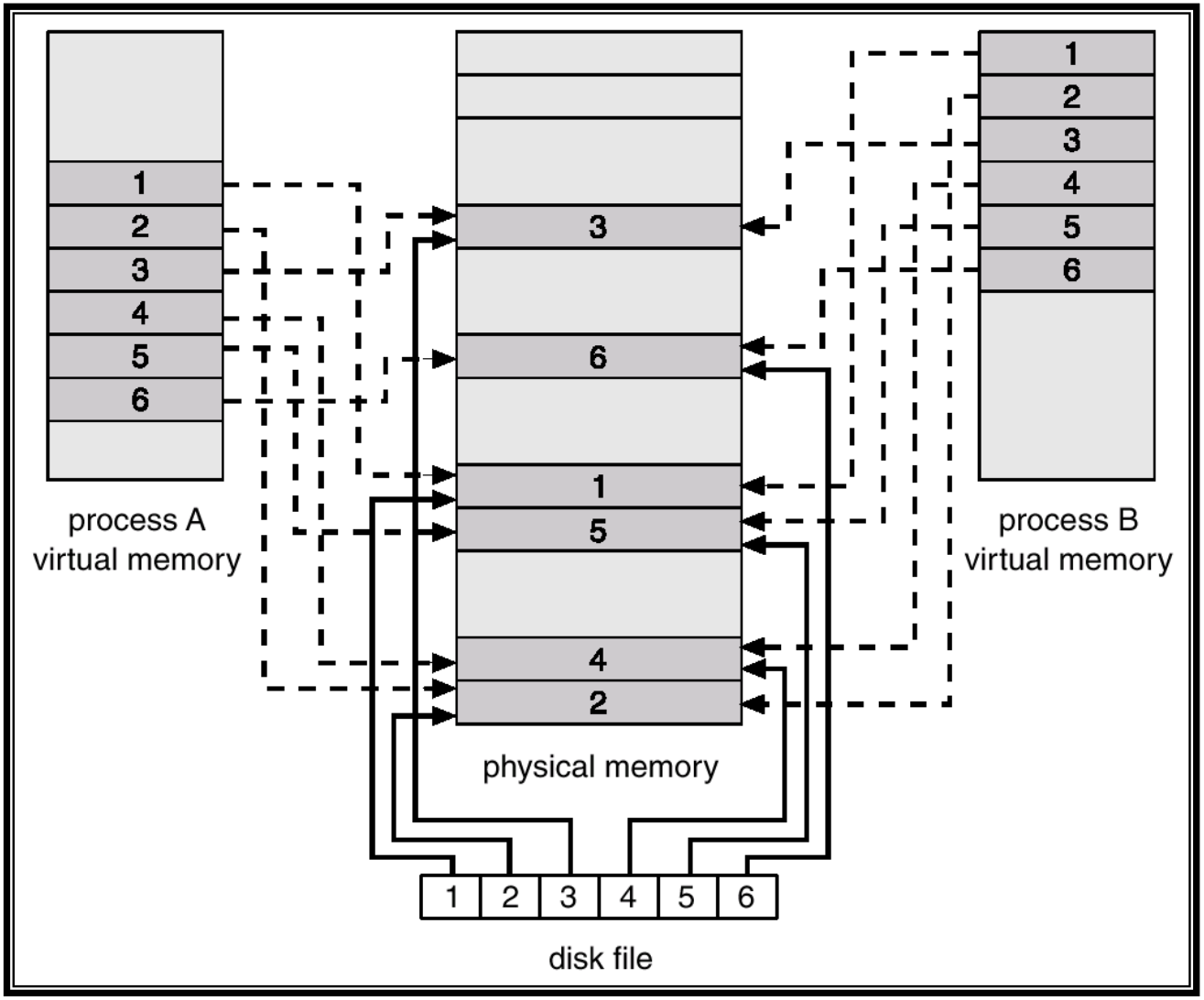
内存映射文件(Mem-Mapped File)

- 基本思想：进程通过一个系统调用（mmap）将一个文件（或部分）映射到其虚拟地址空间的一部分，访问这个文件就像访问内存中的一个大数组，而不是对文件进行读写。
- 在多数实现中，在映射共享的页面时不会实际读入页面的内容，而是在访问页面时，页面才会被每次一页的读入，磁盘文件则被当作后备存储。
- 当进程退出或显式地解除文件映射时，所有被修改页面会写回文件。
- 采用内存映射方式，可方便地让多个进程共享一个文件。

内存映射文件

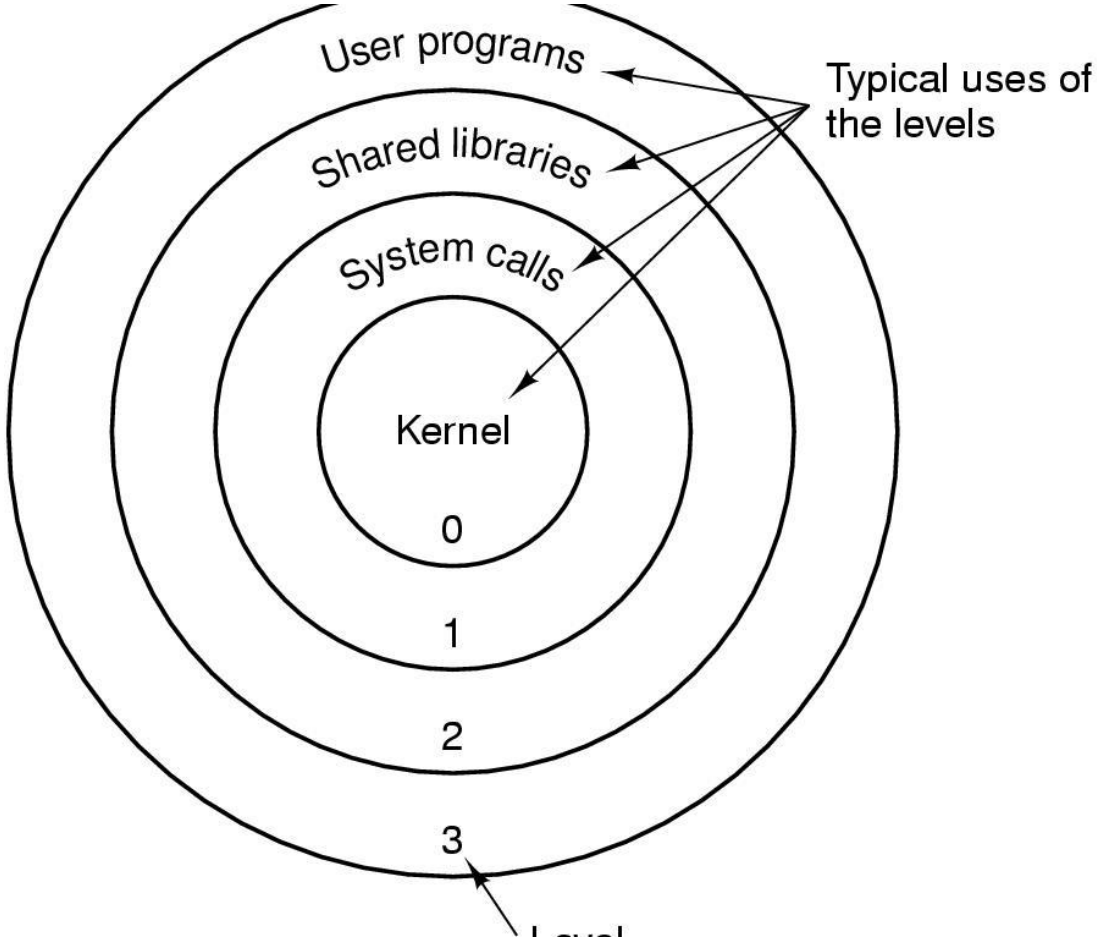


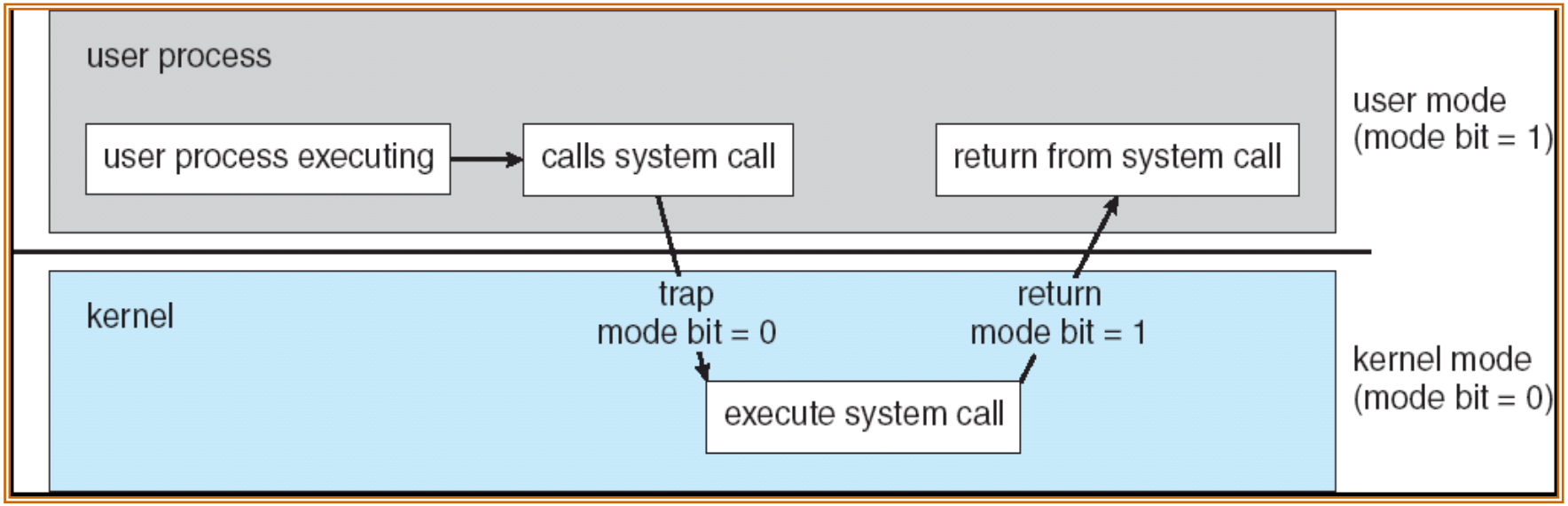
内存映射方式的文件共享



存储保护

- 界限保护（上下界界限地址寄存器）：所有访问地址必须在上下界之间；
- 用户态与内核态
- 存取控制检查
- 环保护：处理器状态分为多个环(ring)，分别具有不同的存储访问特权级别(privilege)，通常是级别高的在内环，编号小（如0环）级别最高；可访问同环或更低级别环的数据；可调用同环或更高级别环的服务。


User programs
Shared libraries
System calls
Kernel
0
1
2
3
Typical uses of
the levels


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)

内容提要

- 存储管理基础
- 页式内存管理
- 段式内存管理
- 虚拟存储管理
- 存储管理实例
  - Unix
  - Windows NT

Unix中的存储管理

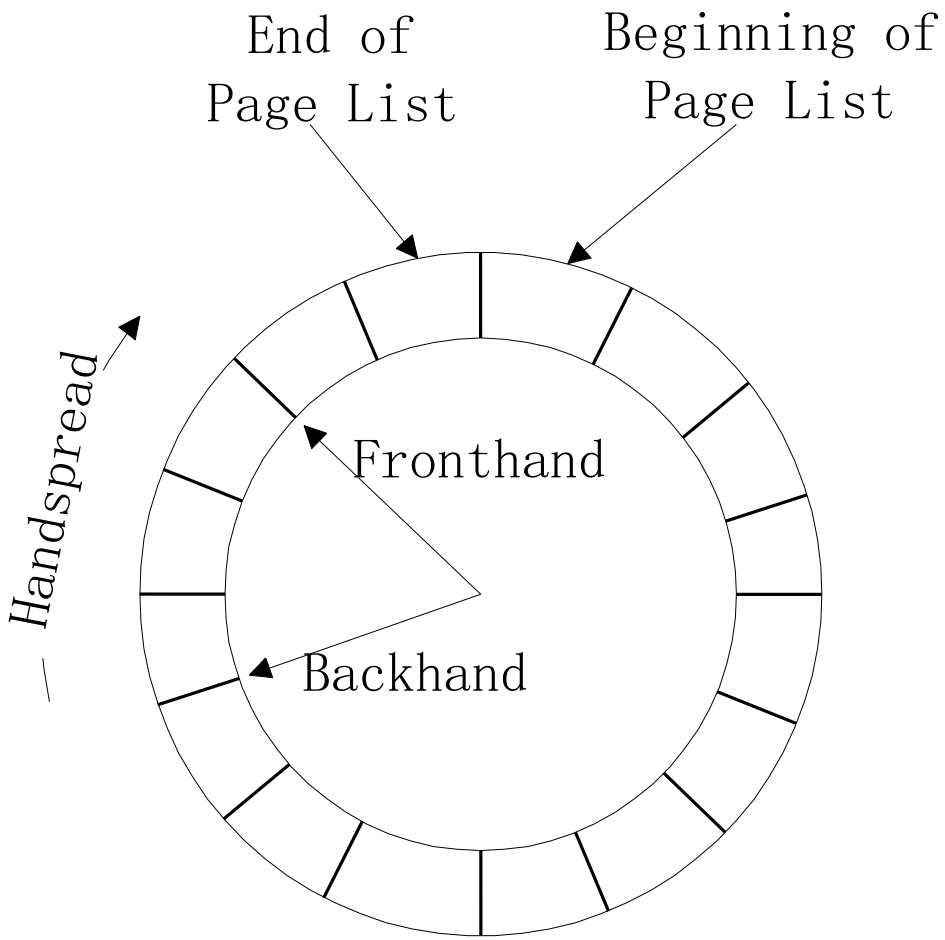
- 由于UNIX可在多种平台上运行，其存储管理差异十分大。这里我们介绍Solaris 2.x的存储管理系统。Solaris的存储管理系统分成两部分：对换系统（paging system）和内核存储分配器（kernel memory allocator）。
  - 对换系统：为进程和磁盘缓存提供页式虚拟存储管理；
  - 内核存储分配器：为内核提供内存分配服务；

对换系统

- 对换系统的数据结构
  - 页表(page table)：每个进程有一个页表，进程逻辑空间的每页对应页表中的一个表项；
  - 硬盘块描述表(disk block descriptor)：该表表项与页表表项一一对应，描述该页在磁盘上的副本；
  - 页面表(page frame date table)：按物理页面号排序，描述每个物理页面；
  - 对换表(swap-use table)：每个磁盘对换区有一个对换表，该对换区上的每个页副本对应一个表项；

双指针轮转置换算法(Two-Handed Clock Page-Replacement Algorithm)

- 该算法利用页面表上的几个指针维护一个空闲页面表。当空闲页面数少于一定阈值时，该算法置换一些页面加入空闲页面表。


End of
Page List
Beginning of
Page List
Handspread
Fronthand
Backhand

内核存储分配器

- 内核存储分配特征
  - 内核需要分配和释放小内存块，用于内核的数据结构和缓冲区；
  - 这些内存块通常比物理页面要小许多；
  - 这些内存块的分配和释放要求快速进行；
  - 分配和释放的内存块的大小变化是缓慢的。

惰性动态分区算法（lazy buddy system）

- 惰性动态分区算法：该算法是一种动态分区算法。
  - 分配的分区大小可变，但只能为2^K。
  - 释放的分区并不马上合并，而是维持一定数目的各种大小的空闲分区；当空闲分区超过一定数目时，才合并。

- 该算法维护的参数：
  - N_i表示大小为2^i的分区数目；
  - A_i表示大小为2^i的已分配分区数目；
  - G_i表示大小为2^i的适合合并空闲（globally free）分区数目；
  - L_i表示大小为2^i的不适合合并空闲（locally free）分区数目；
  - 我们有如下关系：
    $N_i = A_i + G_i + L_i$；
- 合并的判断条件：
  - 要求Li ≤ Ai；依据它们的差来决定是否合并：差为0或1时进行合并，大于1时不合并。

Windows NT中的存储管理

内存管理方式

- 保留与提交
- 内存映射文件
- 内存堆

页面调度策略

- 页面调度策略包括取页策略、置页策略和淘汰策略。
  - 取页策略：NT采用按进程需要进行的请求取页和按集群方法进行的提前取页。集群方法是指在发生缺页时，不仅装入所需的页，而且装入该页附近的一些页。
  - 置页策略：在线性存储结构中，简单地把装入的页放在未分配的物理页面即可。
  - 淘汰策略：采用局部FIFO置换算法。在本进程范围内进行局部置换，利用FIFO算法把驻留时间最长的页面淘汰出去。

工作集策略

- NT根据内存负荷和进程缺页情况自动调整工作集。
  - 进程创建时，指定一个最小工作集（可用SetProcessWorkingSetSize函数指定）；
  - 当内存负荷不太大时，允许进程拥有尽可能多的页面；
  - 系统通过自动调整保证内存中有一定的空闲页面存在；

内存管理小结

- 存储组织（层次），存储管理功能
- 重定位和装入
- 存储管理方式：单一连续区管理，分区管理（静态和动态分区）
- 覆盖，交换
- 页式和段式存储管理：原理，优缺点，数据结构，地址变换，分段的意义，两者比较

虚拟存储器

- 局部性原理，虚拟存储器的原理
- 种类（虚拟页式、段式、段页式），缺页中断
- 存储保护和共享
- 调入策略、分配策略和清除策略
- 置换策略
- 工作集策略