操作系统　Operating System

第三章 内存管理(5)

沃天宇
woty@buaa.edu.cn
2024年3月27日

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

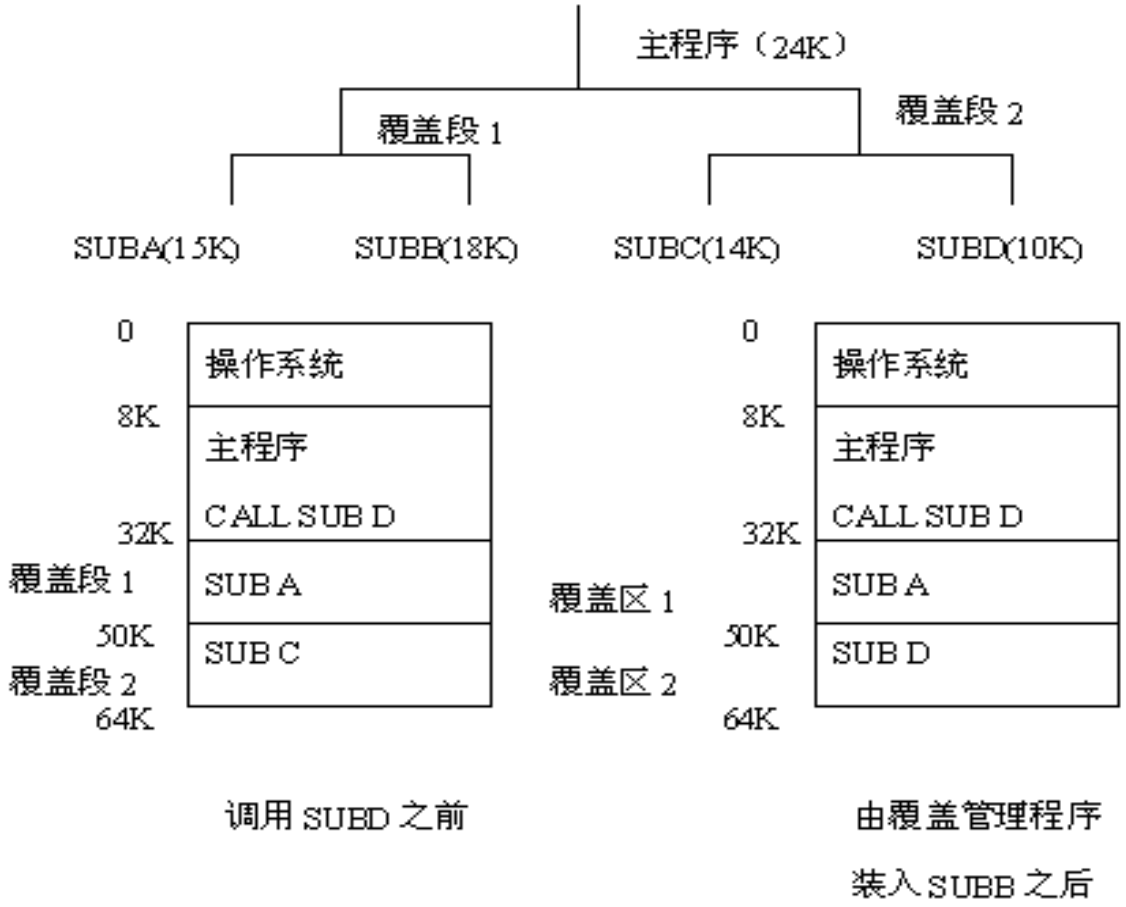
如何超越物理内存限制

- 覆盖（节约，时间上扩展）
  - 重用内存
  - 突破内存占用空间一致性
- 交换（借用，空间上扩展）
  - 辅助存储
  - 调度
  - 突破内存占用时间连续性

覆盖

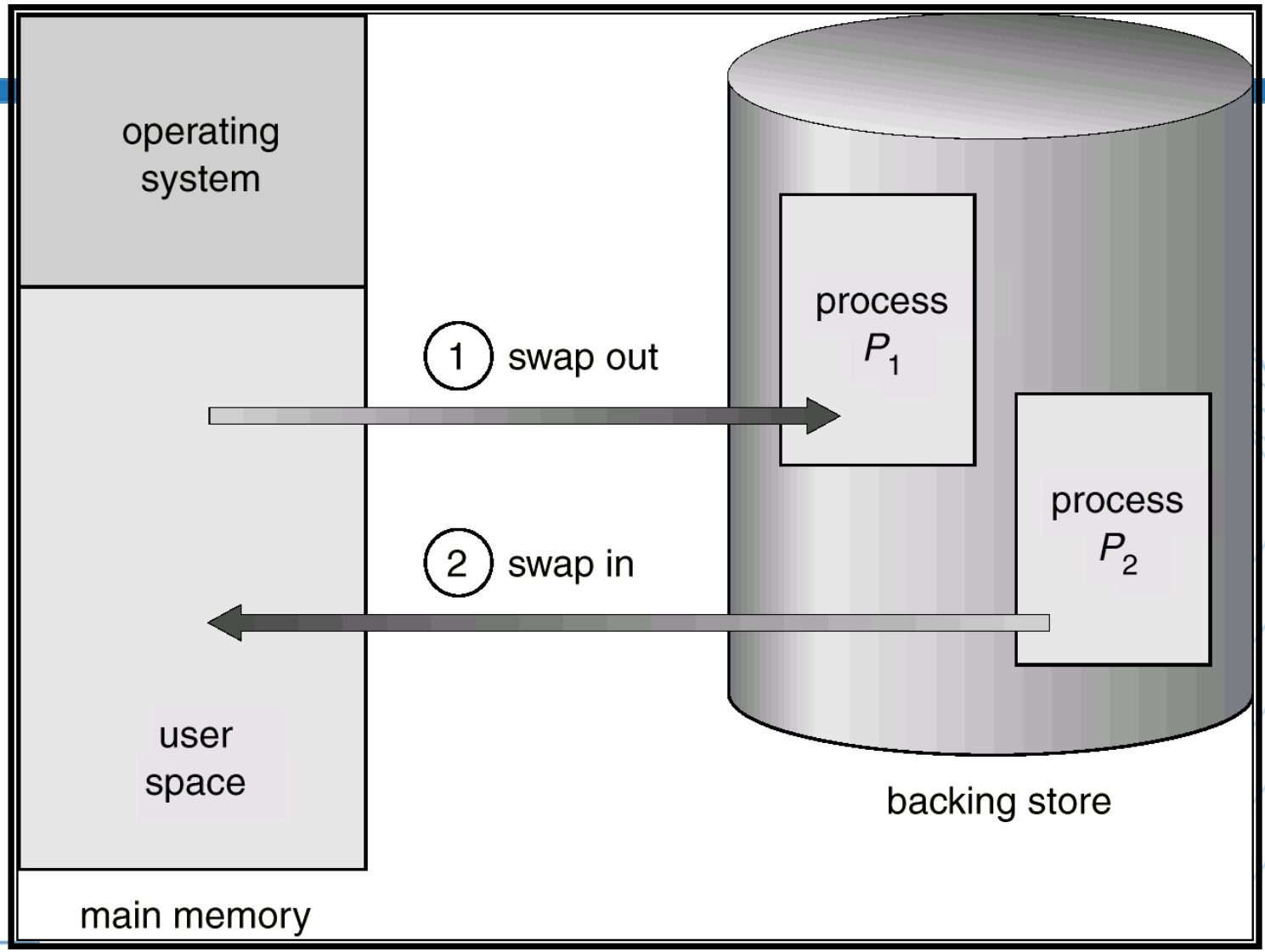
- 覆盖：“覆盖”管理，就是把一个大的程序划分成一系列的覆盖，每个覆盖是一个相对独立的程序单位。把程序执行时并不要求同时装入主存的覆盖组成一组，称其为覆盖段，这个覆盖段被分配到同一个存储区域。这个存储区域称之为覆盖区，它与覆盖段一一对应。
- 缺点：编程时必须划分程序模块和确定程序模块之间的覆盖关系，增加编程复杂度。从外存装入覆盖文件，以时间延长来换取空间节省。

覆盖管理



交换

- 交换：广义的说，所谓交换就是把暂时不用的某个（或某些）程序及其数据的部分或全部从主存移到辅存中去，以便腾出必要的存储空间；接着把指定程序或数据从辅存读到相应的主存中，并将控制转给它，让其在系统上运行。
- 优点：增加并发运行的程序数目，并且给用户提供适当的响应时间；编写程序时不影响程序结构
- 缺点：对换入和换出的控制增加处理机开销；程序整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性。



交换技术的几个问题

- 选择原则，即将哪个进程换出/内存？
  - 等待I/O的进程
- 交换时机的确定，何时需发生交换？
  - 只要不用就换出（很少再用）；
  - 只在内存空间不够或有不够的危险时换出
- 交换时需要做哪些工作？
- 换入回内存时位置的确定

虚拟存储器

- 局部性原理
- 虚拟存储技术

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。

程序的局部性

- 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。

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

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？

测试结果

- bash -c "time ./matrix_locality_ijk" || exit 0
- real 0m11.201s
- user 0m11.192s
- sys 0m0.000s

- bash -c "time ./matrix_locality_kij" || exit 0
- real 0m5.010s
- user 0m5.004s
- sys 0m0.000s

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。

解决方法：

- 增加物理内存容量
- 从逻辑上扩充内存容量：覆盖、对换。

虚拟存储

虚拟存储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、连续的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

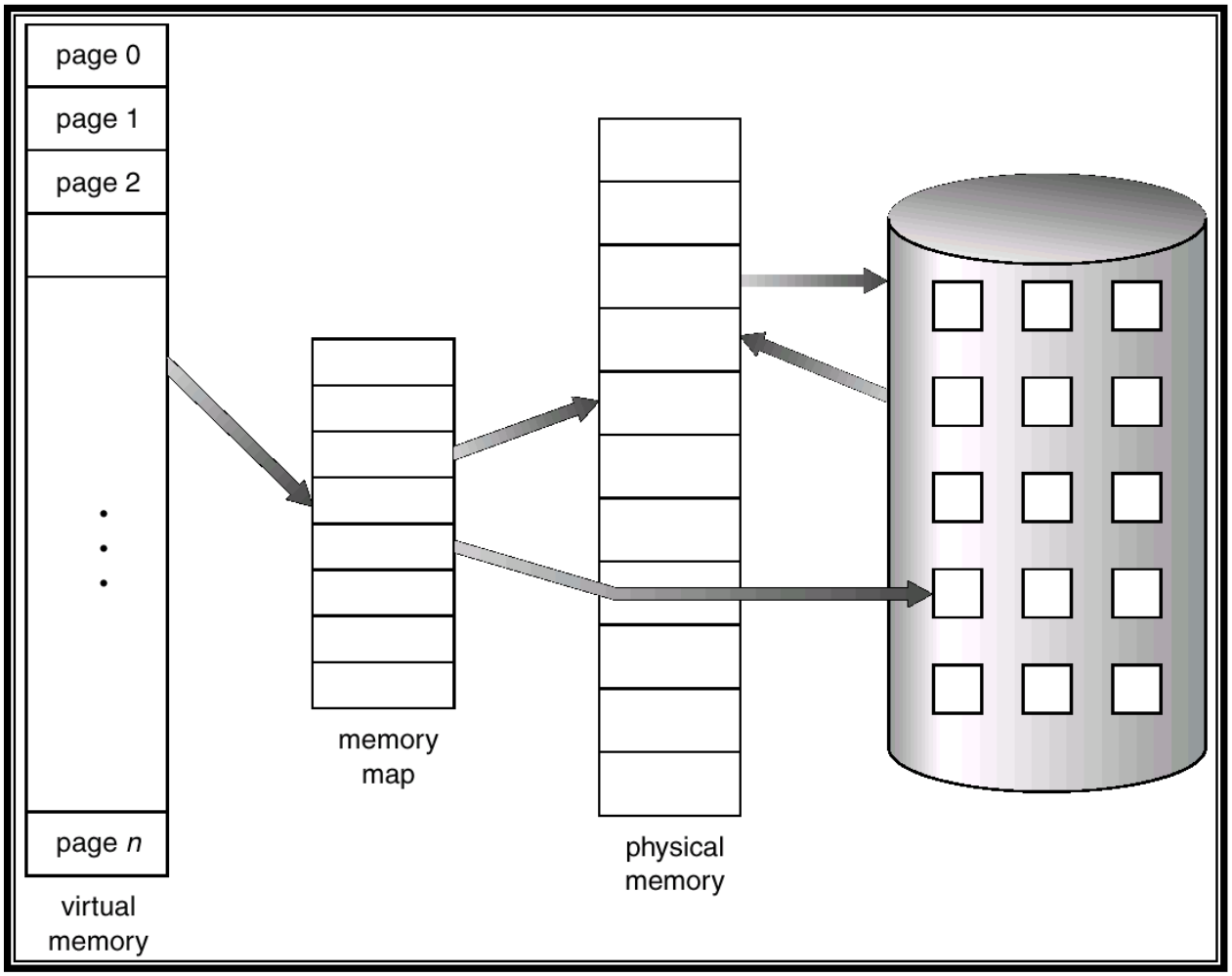
目标：

- 像覆盖技术那样，不是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。但做得更好，由操作系统自动来完成，无须程序员的干涉；
- 像交换技术那样，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。但做得更好，只对进程的部分内容(更小的粒度，如分页)在内存和外存之间进行交换。

虚拟存储的基本原理

- 按需装载：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到内存，就可让程序开始执行。
- 缺页调入：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。
- 不用调出：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(分时复用)：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：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

- 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：

- 可在较小的可用内存中执行较大的用户程序；
- 可在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）（对用户（编程人员）透明）
- 提供给用户可用的虚拟内存空间通常大于物理内存(real memory)

代价：

- 虚拟存储量的扩大是以牺牲 CPU 工作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由计算机的地址结构决定。如 32 位机器，虚拟存储器的最大容量就是 4G，再大 CPU 无法直接访问。

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方面。
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

不同点：

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。

虚拟内存

- 虚拟内存不只是“用磁盘空间来扩展物理内存”的意思——这只是扩充内存级别以使其包含硬盘驱动器而已。
- 把内存扩展到磁盘只是使用虚拟内存技术的一个结果，它的作用也可以通过覆盖或者把处于不活动状态的程序以及它们的数据全部交换到磁盘上等方式来实现。
- 对虚拟内存的定义是基于对地址空间的重定义的，即把地址空间定义为“连续（时间、空间）的虚拟内存地址”，以借此“欺骗”程序，使它们以为自己正在使用一大块的“连续”地址。

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 ）

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。
- 它允许只装入若干页(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

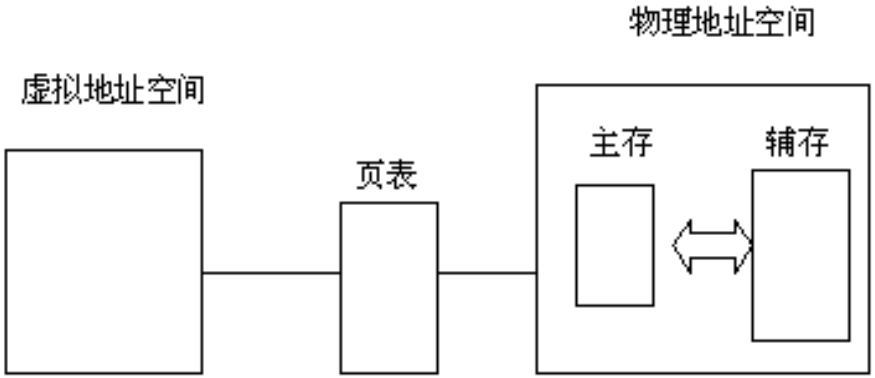
虚存机制要解决的关键问题

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：存储保护与程序再定位等问题

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。

请求式分页系统

- 在运行作业之前，只要求把当前需要的一部分页面装入主存。当需要其它的页时，可自动的选择一些页交换倒辅存去，同时把所需的页调入主存。
- 虚拟存储系统：控制自动页面交换而用户作业意识不到的那个机构，成为虚拟存储系统。

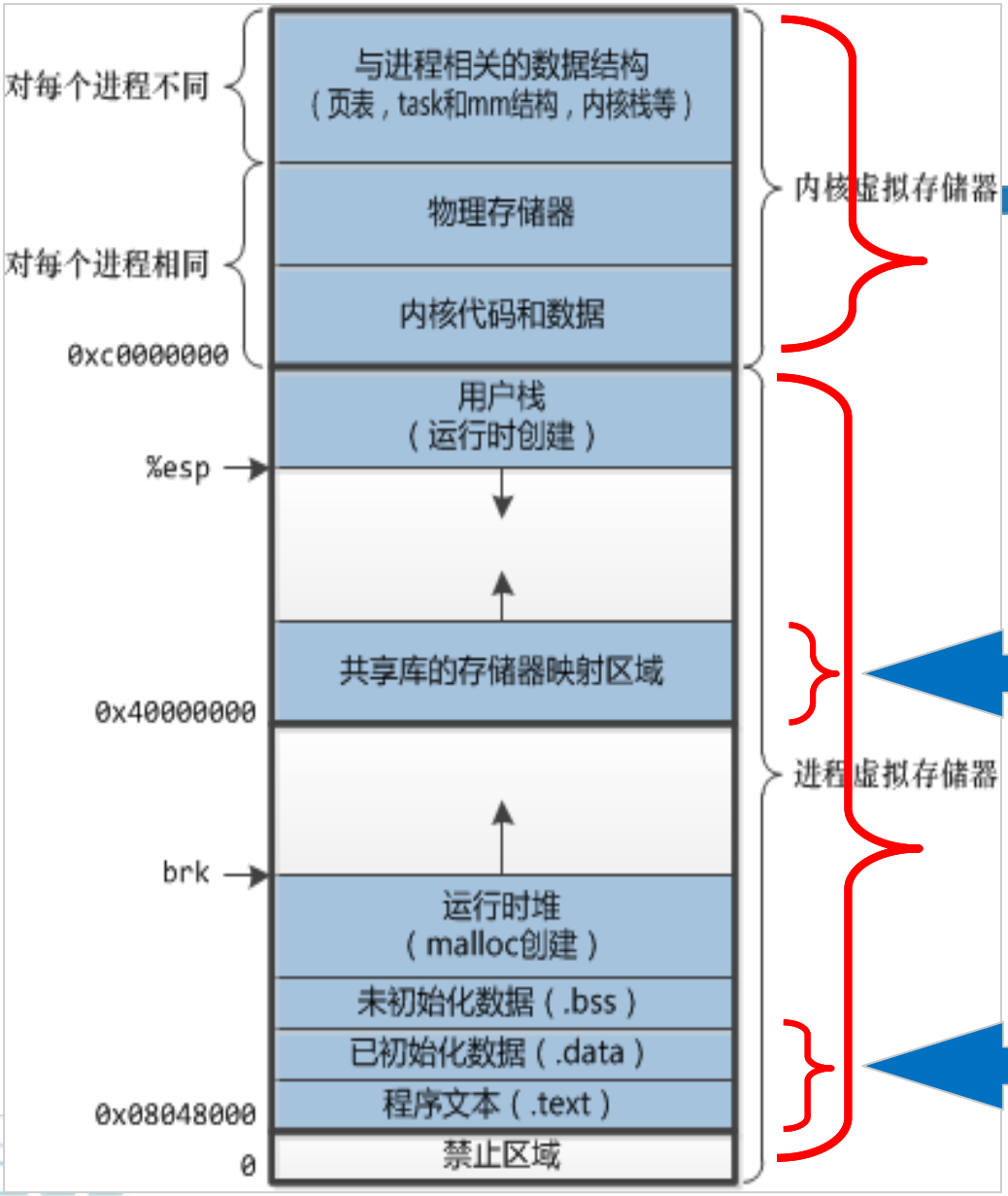


基本概念1-进程的逻辑空间（虚拟空间）

- 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。后面主要讨论用户可见部分。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。

进程的逻辑空间



用不户可见的进程空间，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）：

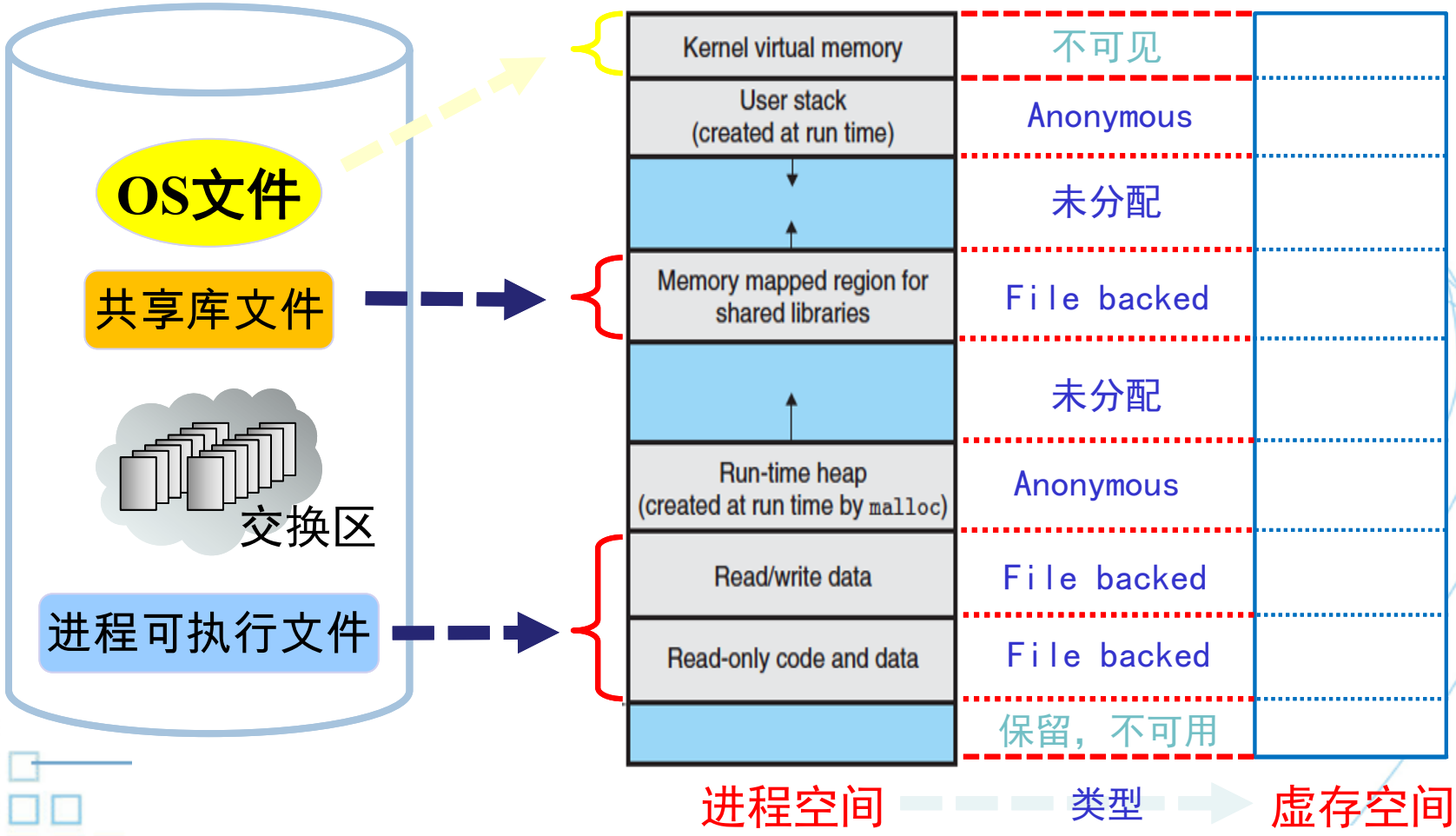
  If (M < 2G) Swap = 2*M

  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。（经验）

地址映射问题（以32位Linux为例）

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）

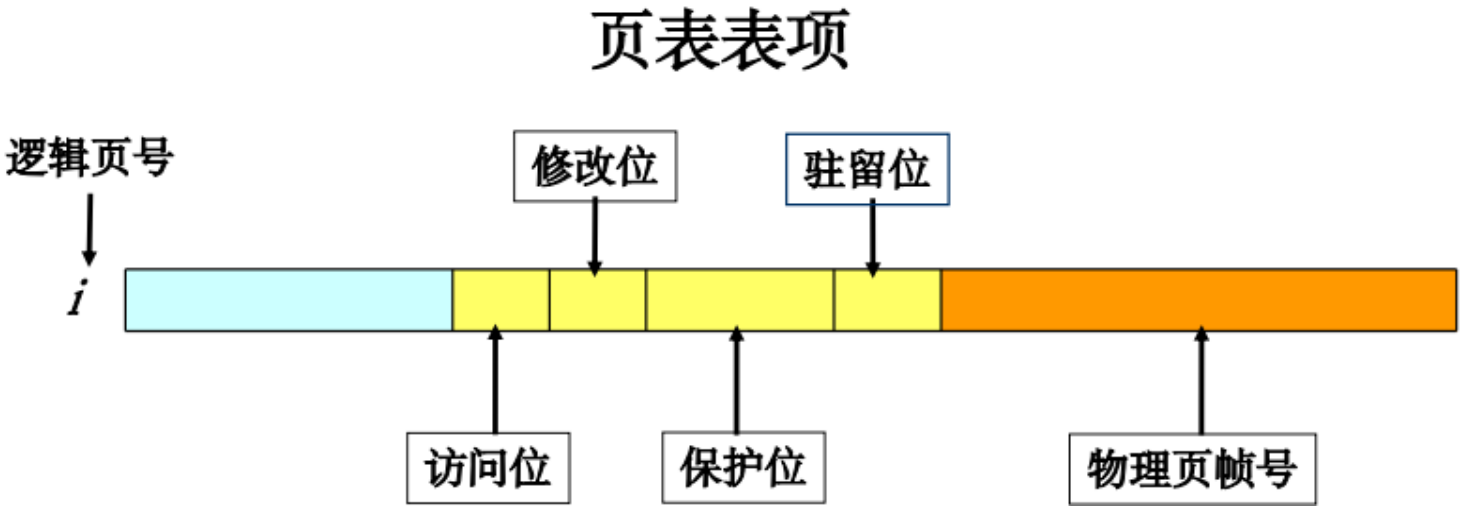
- 在程序装入时，由装载器（Loader）完成。
- 分配是以段为单位（需页对齐）进行的。
- 事实上，在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局，实际上并不立即就把虚拟内存对应位置的程序数据和代码（如.text .data段）拷贝到物理内存中，只是建立好虚拟内存和磁盘文件之间的映射（叫做存储器映射），等到运行到对应的程序时，才会通过**缺页异常**，来拷贝数据。（延迟装载。好处？代价？）

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



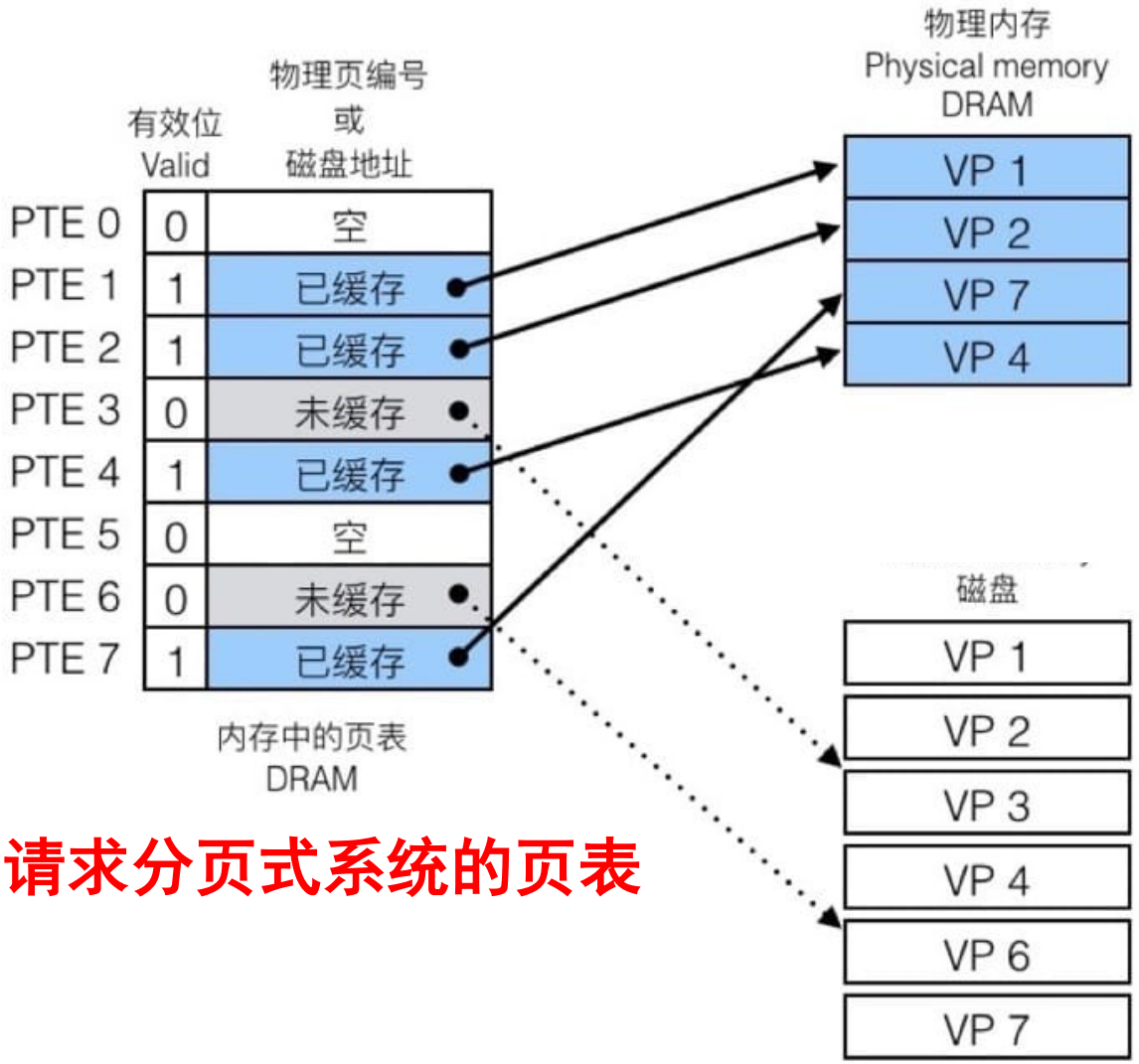
- 驻留位：1表示该页位于内存当中，0，表示该页当前还在外存当中。
- 保护位：只读、可写、可执行。
- 修改位：表明此页在内存中是否被修改过。
- 访问（统计）位：用于页面置换算法。

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



请求分页式系统的页表

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。

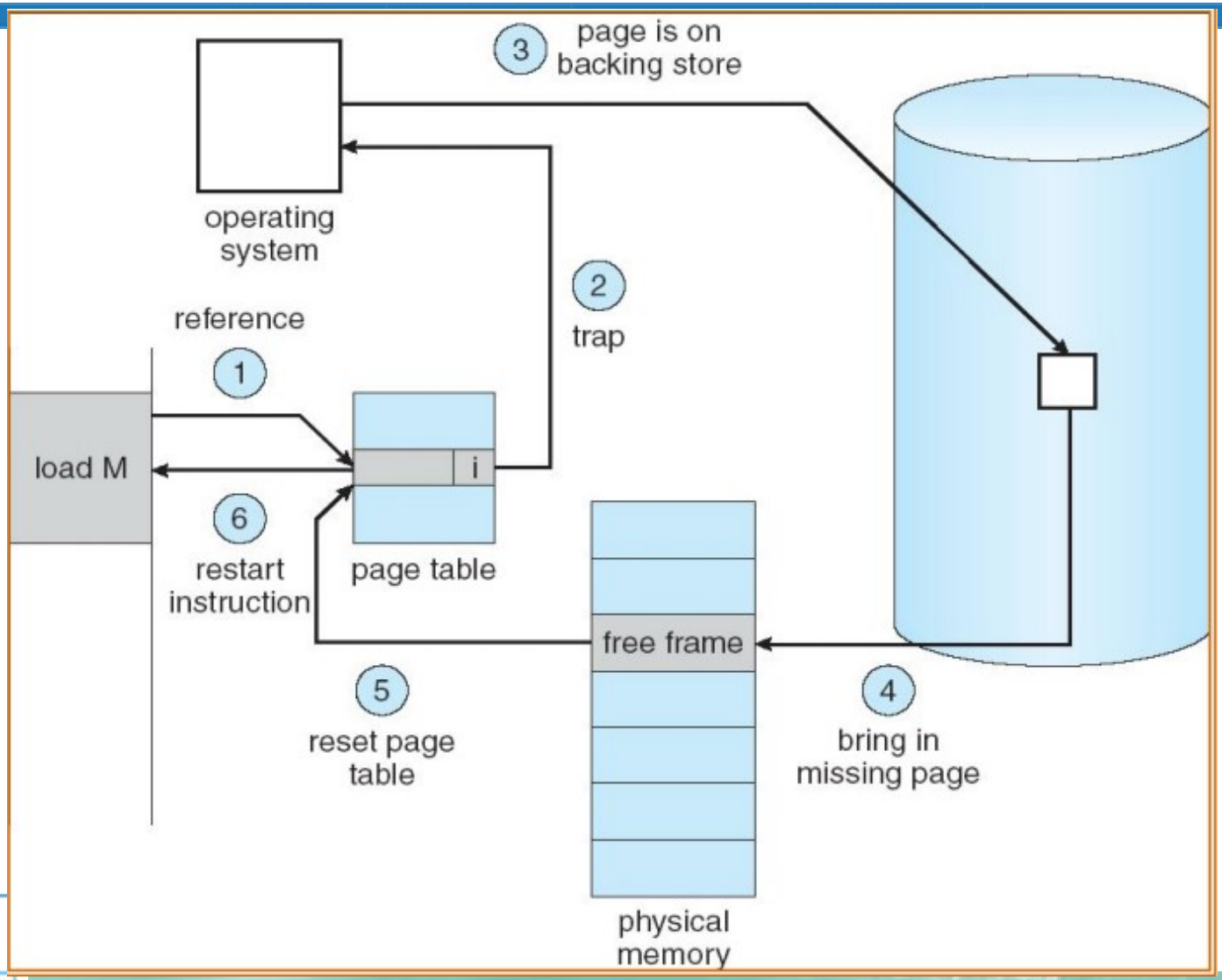
预调页（pre-paging）

- 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）（旧页面写回）
6. 页框“干净”后，操作系统将保存在磁盘上的页面内容复制到该页框中[2]。（新页面调入（2））

12 此时会引起一个磁盘读写调用，发生上下文切换（在等待磁盘读写的过程中让其它进程运行）。

缺页错误处理过程

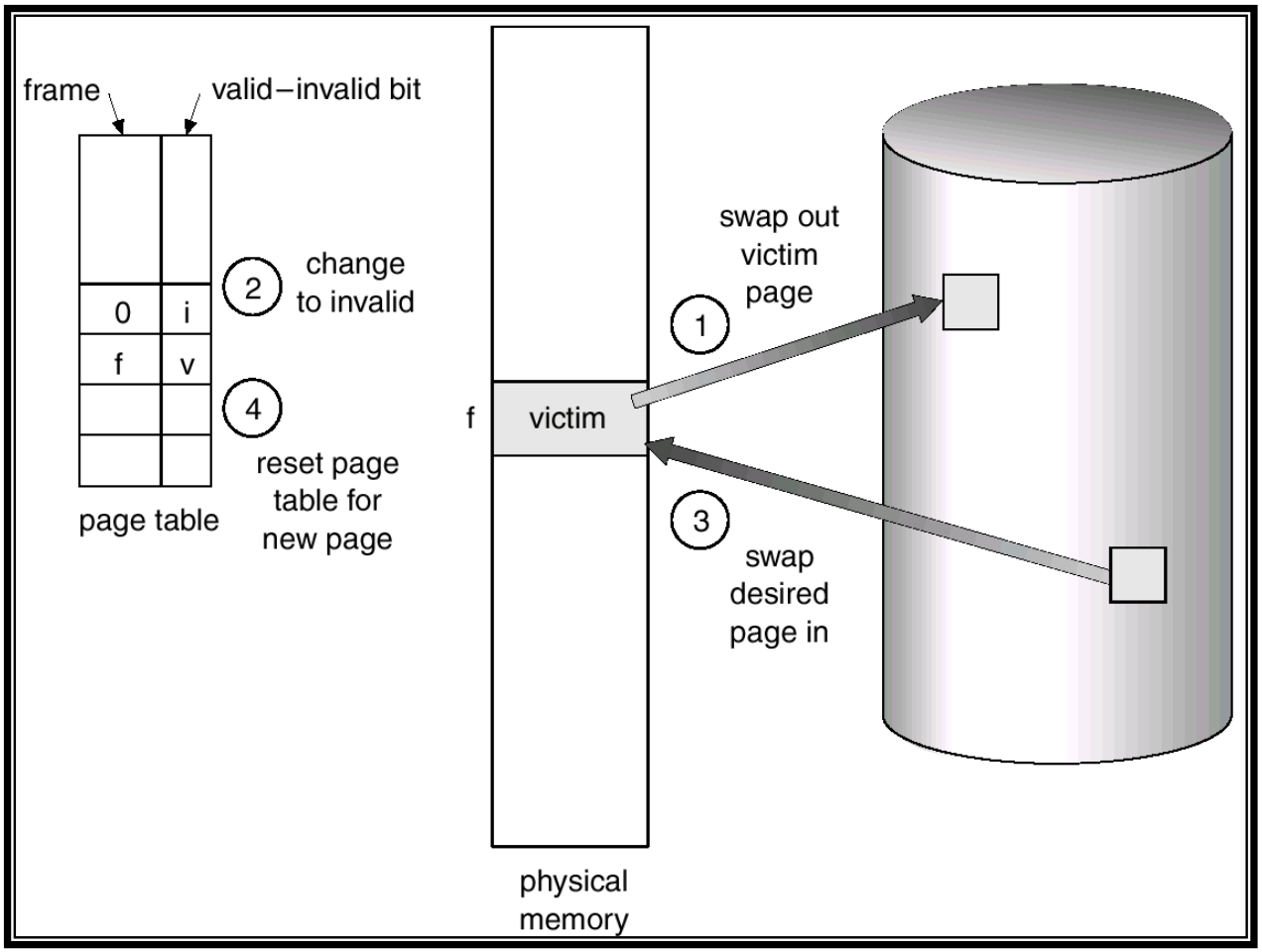
7. 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态。（更新页表）
8. 恢复缺页中断发生前的状态，将程序指针重新指向引起缺页中断的指令。（恢复现场）
9. 程序重新执行引发缺页中断的指令，进行存储访问。（继续执行）

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？
  - 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



关于实验

- 函数调用规范（ABI）
  - 《See MIPS Run Linux》第11章
- 结构体内存布局
- 什么是栈桢（Stack Frame）？
- 函数调用时堆栈的变化？
- 什么是LEAF函数？NESTED函数？
- C语言函数调用传参顺序？
- 如何处理可变长函数参数表？（printf）

11.1.5 Memory Layout of Structure and Array Types and Alignment

- 结构体的内存布局
  - 右侧程序的执行结果？

s1

7a	00
10	00
78	56
34	12

s2

7a	00
00	00
78	56
34	12
10	00
00	00

```
// a.c
#include <stdio.h>

struct thing1 {
        char letter;
        short count;
        int value;
};
struct thing2 {
        char letter;
        int value;
        short count;
};
struct thing1 s1={'z', 16, 0x12345678};
struct thing2 s2={'z', 0x12345678, 16};

int main() {
        printf("%ld\n", sizeof(struct thing1));
        printf("%ld\n", sizeof(struct thing2));
}
```

执行结果

- gcc a.c
- ./a.out
  - 8
  - 12
- objdump -s a.out

s1

7a	00
10	00
78	56
34	12

s2

7a	00
00	00
78	56
34	12
10	00
00	00

```
Contents of section .data:
 4000 00000000 00000000 08400000 00000000  .........@......
 4010 7a001000 78563412 7a000000 78563412  z...xV4.z...xV4.
 4020 10000000                             ....
```

11.2.9什么是栈桢（Stack Frame）

- 运行时动态分配内存
  - 栈桢结构**通常**是编译时可确定的

- 存储函数参数
- 存储临时变量
- 存储返回地址

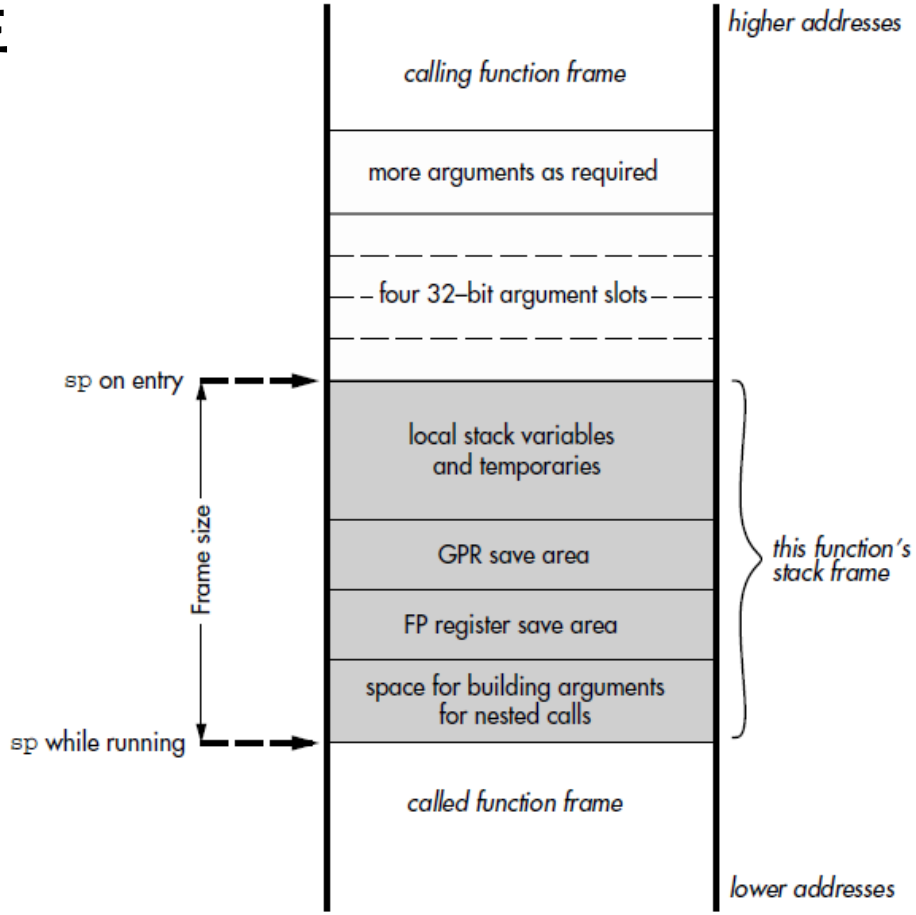


FIGURE 11.8 Stack frame for a nonleaf function.

11.2.9什么是栈桢（Stack Frame）

- 编译文件
  - mips_4KC-gcc -mno-abicalls -O -c a.c
- 观察编译结果
  - mips_4KC-objdump -D a.o

```
00000008 <nonleaf>:
   8:   27bdffd0        addiu   sp,sp,-48
   c:   afbf0028        sw      ra,40(sp)
  10:   afa70010        sw      a3,16(sp)
  14:   afa50014        sw      a1,20(sp)
  18:   afa60018        sw      a2,24(sp)
  1c:   afa4001c        sw      a0,28(sp)
  20:   27a20040        addiu   v0,sp,64
  24:   afa20020        sw      v0,32(sp)
  28:   24040001        li      a0,1
  2c:   24050002        li      a1,2
  30:   24060003        li      a2,3
  34:   0c000000        jal     0 <nested>
  38:   24070004        li      a3,4
  3c:   8fbf0028        lw      ra,40(sp)
  40:   03e00008        jr      ra
  44:   27bd0030        addiu   sp,sp,48
```

```
00000048 <main>:
  48:   27bdffe0        addiu   sp,sp,-32
  4c:   afbf0018        sw      ra,24(sp)
  50:   2402000f        li      v0,15
  54:   afa20010        sw      v0,16(sp)
  58:   2404000b        li      a0,11
  5c:   2405000c        li      a1,12
  60:   2406000d        li      a2,13
  64:   0c000000        jal     0 <nested>
  68:   2407000e        li      a3,14
  6c:   8fbf0018        lw      ra,24(sp)
  70:   03e00008        jr      ra
  74:   27bd0020        addiu   sp,sp,32
```

```
// a.c
int nested(int a, int b, int c, int d, int e, int f,
int g, int h, int *i)
{
        return 0;
}

int nonleaf(int a, int b, int c, int d, int e)
{
        return nested(1, 2, 3, 4, d, b, c, a, &e);
}

int main() {
        int r = nonleaf(11, 12, 13, 14, 15);
        return r;
}
```

```
00000000 <nested>:
   0:   03e00008        jr      ra
   4:   00001021        move    v0,zero
```

11.2.9什么是栈桢（Stack Frame）

- 参考右边栈桢格式画出每次函数调用前后（sp指针变化前后）的堆栈布局

TABLE 11.4 Stack Layout for `nonleaf()`

		48	e
		44	(reserved for c/**a3**)
		40	(reserved for b/**a2**)
		36	(reserved for a/**a1**)
sp on entry	⇒	32	(reserved for a/**a0**)
		28	(pad to 8 bytes)
		24	saved **ra**
		20	(pad to 8 bytes)
		16	&e
		12	(reserved for a/**a3**)
		8	(reserved for c/**a2**)
		4	(reserved for b/**a1**)
sp running	⇒	0	(reserved for d/**a0**)

```
extern nested (int a, int b, int c, int d, int *e);

extern nonleaf (int a, int b, int c, int d, int e)
{
  nested(d, b, c, a, &e);
}
```

什么是LEAF函数？NESTED函数？

- LEAF函数：不调用其他函数的函数
  - 直接通过j ra返回
  - 通常不需要操纵栈
- NESTED函数
  - 需要保存ra
  - 需要操作栈

```
#include <mips/asm.h>
#include <mips/regdef.h>

LEAF(myleaf)
    ...
    <your code goes here>
    ...
    j       ra
END(myleaf)
```

C语言函数调用传参顺序

- 从右向左依次进栈

TABLE 11.4 Stack Layout for `nonleaf()`

		48	e
		44	(reserved for c/**a3**)
		40	(reserved for b/**a2**)
		36	(reserved for a/**a1**)
sp on entry	⇒	32	(reserved for a/**a0**)
		28	(pad to 8 bytes)
		24	saved **ra**
		20	(pad to 8 bytes)
		16	&e
		12	(reserved for a/**a3**)
		8	(reserved for c/**a2**)
		4	(reserved for b/**a1**)
sp running	⇒	0	(reserved for d/**a0**)

```
extern nested (int a, int b, int c, int d, int *e);

extern nonleaf (int a, int b, int c, int d, int e)
{
  nested(d, b, c, a, &e);
}
```

11.2.10 可变长函数参数表

- 特殊宏
  - va_list
  - va_start
  - va_arg
  - va_end
- 实际上是对栈进行操作
- 如果fmt与实际传参不匹配，会怎样？

```
/* lib/printf.c */

void printf(char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    lp_Print(myoutput, 0, fmt, ap);
    va_end(ap);
}


/* init/main.c */
int main() {
     printf("%d,%d,%d\n", 1, 2, 3);
     printf("%d,%x\n", 1, 0x12);
}
```