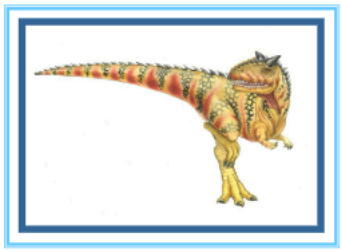
第 10 章：虚拟内存



第 10 章：虚拟内存

- 后台需求分页写入时复制
  页面替换帧分配抖动
-
-
-
-
-
- 其他注意事项

目标

- 描述虚拟内存及其优势。说明如何使用需求分页将页面加载到内存中。应用 FIFO、optimal 和 LRU 页面替换算法。描述进程的工作集，并解释它与程序位置的关系。

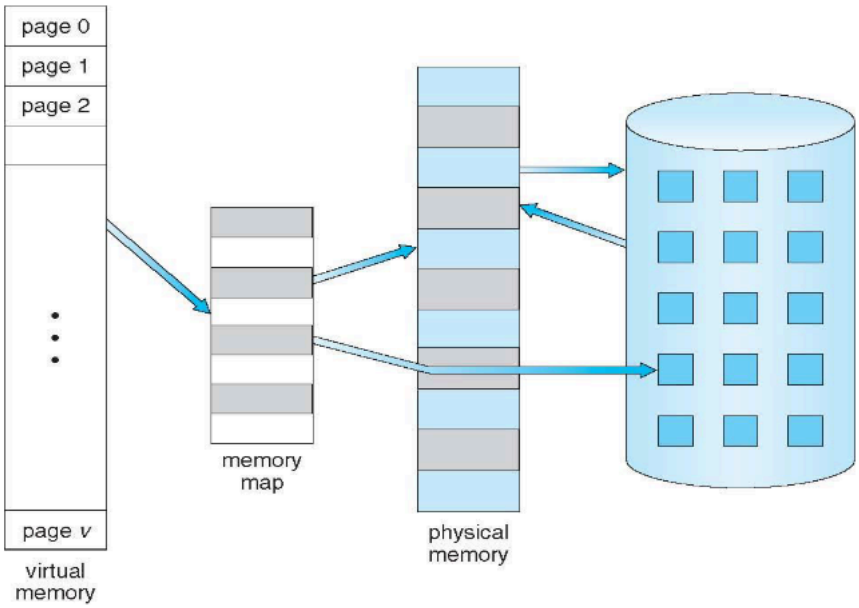
背景

- 代码需要在内存中才能执行，但不一定是整个程序
  - 错误代码、异常例程。在实践中很少（如果有的话）发生一些错误，这段代码几乎从未执行过
  - 大型数据结构（如数组、列表和表）分配的内存通常超过其需要的内存。例如，一个数组可以声明为 100x100 个元素，即使它很少大于 10x10
  - 程序的某些选项和功能可能很少使用
- 即使需要整个程序，也可能不是同时需要全部考虑执行部分加载程序的能力
  - 程序不再受物理内存限制的约束。可以使用非常大的虚拟内存地址编写程序，从而简化编程任务
  - 每个用户程序可以占用更少的物理内存，可以同时运行更多的程序，这会增加 CPU 利用率（多重编程的程度）和吞吐量将用户程序加载或交换到物理内存中所需的 I/O 更少，因此每个用户程序运行得更快。

背景

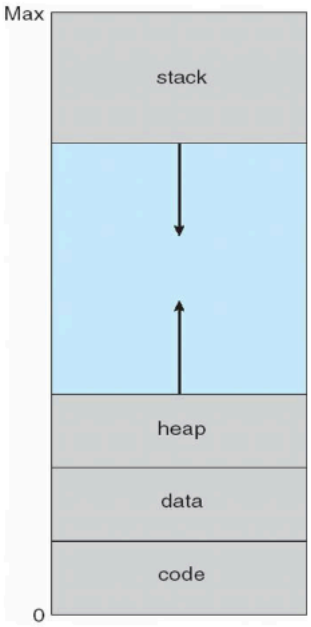
- 虚拟内存 – 程序员感知的用户逻辑内存（以前称为地址空间）与物理内存的分离
  - 程序只有一部分需要位于内存中才能执行 因此，逻辑地址空间可以比实际的物理地址空间大得多 允许多个进程共享地址空间。例如，系统库可以由多个进程共享
  -
  -
  - 允许更高效的流程创建，因为在流程创建期间可以共享页面，从而加快流程创建
  - 更多程序同时运行 – 提高多编程程度加载或交换进程所需的 I/O 更少
  -
- 虚拟内存可以通过以下方式实现：
  - 需求分页或需求细分 – 原则上，它们是相似的，但固定大小的框架/页面和可变大小的细分的细节不同

大于物理内存的虚拟内存

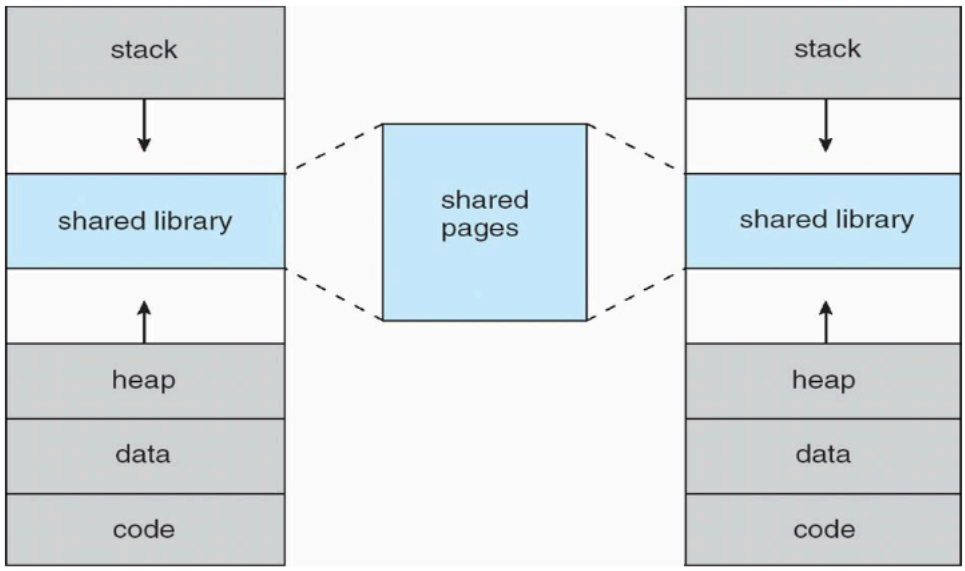


虚拟地址空间

- 虚拟地址空间 – 进程如何在内存中存储的逻辑视图
  - 通常从地址 0 开始，连续的地址直到地址空间结束
  - 同时，以帧 MMU 组织的物理内存必须将 logical 映射到 physical address
- 堆可以在内存中向上增长，用于动态内存分配。堆栈可以通过连续的函数调用在内存中向下增长
- 堆和堆栈之间的大空白空间（或空洞）是虚拟地址空间的一部分，但只有当堆或堆栈增长时，才需要实际的物理页面（空间）。
- 支持稀疏地址空间，并为增长留下漏洞、动态链接库等系统库可以通过映射到虚拟地址空间来共享
- 在进程创建期间，可以使用 fork（） 系统调用共享页面，从而加快进程创建速度

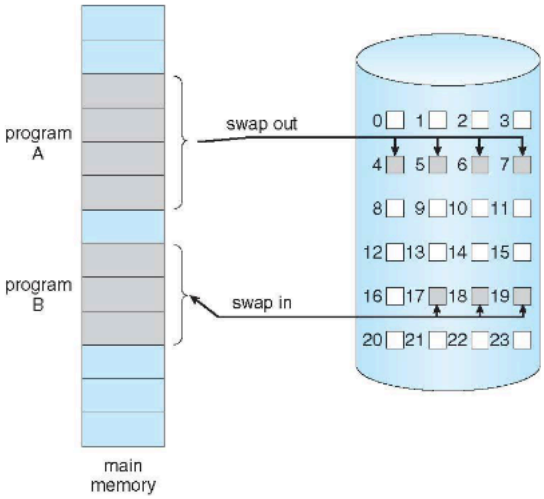


使用虚拟内存的共享库



需求分页

- 可以在加载时将整个进程带入内存
- 或者仅在需要或引用页面时将页面放入内存
  - 需要更少的 I/O，没有不必要的 I/O 需要更少的内存
  - 更快的响应 更多的用户正在运行
- 类似于带交换的分页系统（右图）
- 需要 Page = > 引用它
  - 无效引用（非法内存地址） => Abort
- not-in-memory => 带入内存Lazy swapper – 除非需要页面，否则从不将页面交换到内存中
  - 处理页面的 Swapper 是寻呼机

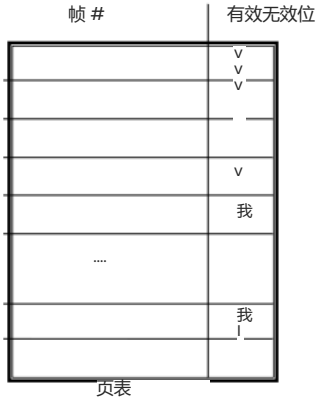


基本概念

- Pager 仅将那些 “需要” 的页面引入内存 如何确定带入内存的页面集?
  - 需要新的 MMU 功能来实现需求分页
- 如果所需的页面已经驻留在内存中
  - 与非按需分页 MMU 没有区别
- 如果所需的页面不是内存驻留的
  - 需要检测页面并将其从辅助存储设备加载到内存中
    - 无需更改程序行为 无需程序员更改代码 – 应用程序不知道这一点

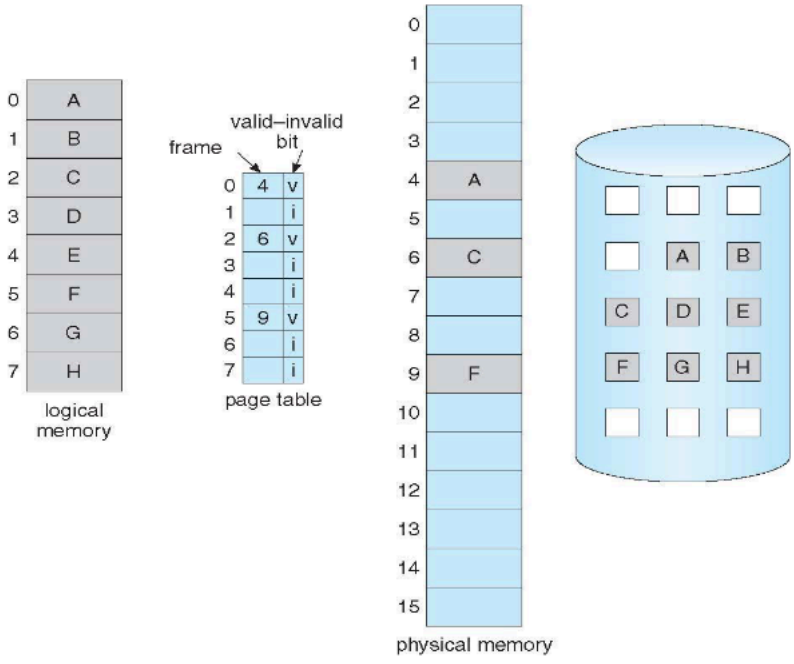
Valid-Invalid Bit

- 对于每个页表条目，都会关联一个有效 - 无效位（v ⊠ in-memory – memory resident，i > not-in-memory）最初有效 - 无效位在所有条目上都设置为 i
-
- 页表快照示例：



- 在地址转换期间，如果页表项中的 valid-invalid bit 为 i ⊠ page fault

当某些页面不在主内存中时，页表

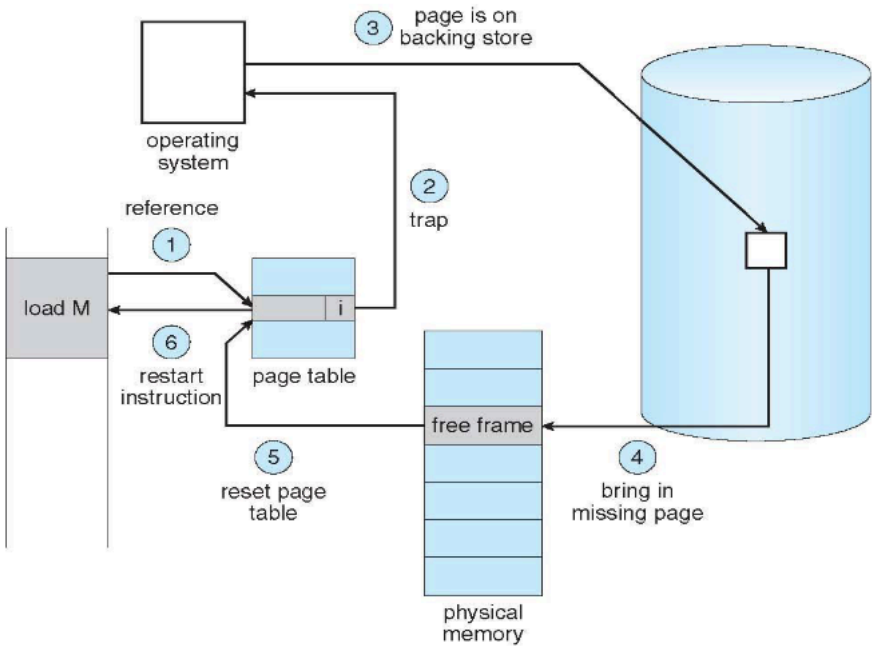


页面错误

- 如果存在对某个页面的引用，则对该页面的第一次引用将捕获到操作系统：
  页面错误

1. 操作系统会查看相应的页表条目来决定：
   - 无效的引用（非法地址）⊠ abort Just not in memory
   -
2. 获取空帧（如果有） - 操作系统维护空闲帧列表
3. 通过计划的磁盘操作将页面（在二级存储上）交换到框架中
4. 更新 page table 中的相应条目
5. 重置页表以指示此页现在位于内存中，设置验证位 = v
6. 重新启动导致页面错误的指令（取决于 CPU 调度）

处理页面错误的步骤



需求分页的各个方面

- 极端情况 – 启动内存中没有页面的进程 OS 将指令指针设置为进程的第一条指令，非内存驻留 -> 页面错误，并且对于第一次访问的所有其他进程页面
  - 这称为纯需求分页
- 给定的指令可以访问多个页面>从而导致多个页面错误
  - 考虑 fetch 和 decode 指令，它从内存中添加两个数字并将结果存储回内存
  - 由于内存引用的局部性，在进程开始运行一段时间后，页面错误引起的痛苦会减少
- 通常，为了最大程度地减少初始和潜在的高页错误，OS 会在进程执行开始之前将一些页面预先分页到内存中。
- 需求分页需要硬件支持
  - 以有效/无效位为指示的页表用于 Page In 和 Page Out 的辅助存储器（具有交换空间的交换设备）指令重启

自由帧列表

- 发生页面错误时，操作系统必须将所需的页面从辅助存储导入主内存。
- 大多数操作系统都维护一个空闲帧列表 —— 用于满足此类请求的空闲帧池（只有操作系统才能访问的内核数据结构）

- 操作系统通常使用一种称为按需零填充的技术来分配空闲帧，即帧的内容在（重新）分配之前清零。
  - 在页面可供进程使用之前将 0 写入页面的技术 - 从而擦除其先前的内容或阻止任何旧数据对进程可用考虑在重新分配帧之前不清除帧内容的潜在安全隐患。
- 当系统启动时，所有可用内存都放在空闲帧列表中。

需求分页的性能

- 需求分页阶段 – 处理页面错误

1. Trap 到操作系统
2. 保存用户寄存器和进程状态
3. 确定中断是页面错误
4. 检查页面引用是否合法，并确定页面在磁盘上的位置
5. 发出从磁盘到物理内存中的空闲帧（如果可用）的读取：
    1. 在队列中等待此设备，直到处理读取请求
    2. 等待设备寻道和/或延迟时间
    3. 开始将页面传输到自由框架
6. 等待时，将 CPU 核心分配给其他一些进程
7. 从磁盘 I/O 子系统接收中断（I/O 已完成）
8. 保存另一个进程的寄存器和进程状态（取决于 CPU 调度）
9. 确定中断来自磁盘
10. 更新页表和其他表以显示页现在在内存中
11. 等待 CPU 再次分配给此进程
12. 恢复用户寄存器、进程状态和新的页表，然后恢复中断的指令

需求分页性能（续）

- page-fault 服务时间有三个主要任务：
    - 为中断提供服务 – 仔细编码可能会导致几百条指令 阅读页面 – 大量时间（访问辅助存储，通常是硬盘） 重新启动进程 – 少量时间
    -
    -
- 页面切换时间可能接近 8 毫秒（对于典型的硬盘）
- 页面错误率 0 ⊴ p ⊴ 1
    - 如果 p = 0，则无页面错误如果 p = 1，则每个引用都是错误
    -
- 有效访问时间 （EAT）

EAT = （1 – p） x 内存访问 + p（页面错误开销

+ 交换页面输出 + 交换页面输入 + 重新启动开销）

需求分页示例

- 内存访问时间 = 200 纳秒平均页面错误服务时间 = 8 毫秒
- 吃 = （1 – p） x 200 + p （8 毫秒） = （1 – p） x 200 + p x 8,000,000 = 200 + p x 7,999,800
- 如果 1,000 次访问中有一次导致页面错误，则

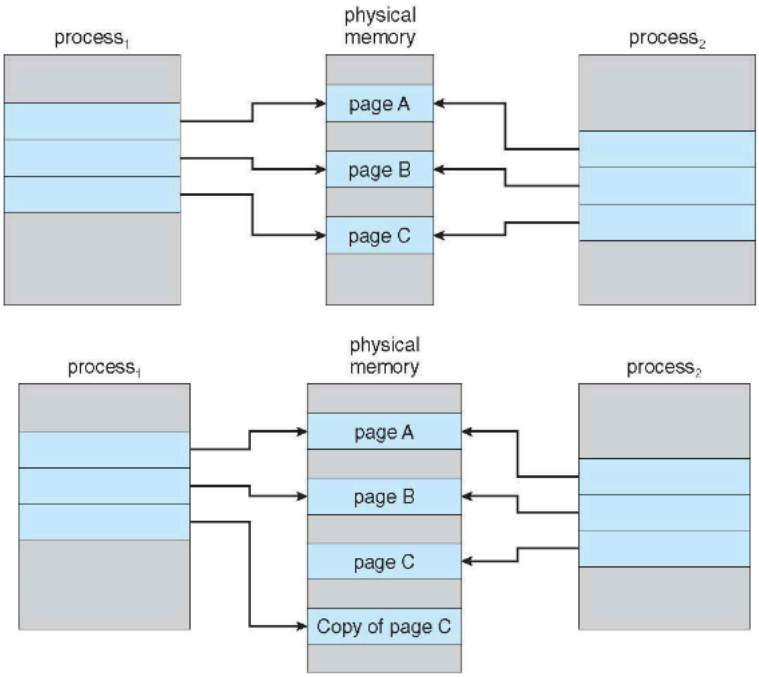
  EAT = 8.2 微秒。这慢了 40 倍！！如果希望性能下降< 10%
-
  - 220 > 200 + 7,999,800 x p 20 > 7,999,800 x p p < .0000025
  -

  <每 400,000 次内存访问中有一个页面错误幸运的是，内存局部性通常满足这一点，因为每次内存丢失都会带来一整页
-

写入时复制

- 传统上，fork （） 的工作原理是为子节点创建父级地址空间的副本，复制属于父级的页面
- Copy-on-Write （COW） 允许父进程和子进程最初共享内存中的相同页面
  - 这些共享页被标记为写入时复制页，这意味着如果任一进程写入共享页，则必须为该进程创建共享页的副本。在这种情况下，未修改的页面将留给另一个进程，没有共享
- COW 允许更高效地创建流程，因为仅复制或复制修改后的页面 此技术提供快速流程创建，并最大限度地减少必须分配给新创建流程的新页面数量
-
  - 考虑到许多子进程在创建后立即调用 exec （） 系统调用，因此无论如何都不需要复制父进程的地址空间

进程 1 修改页面 C 之前和之后



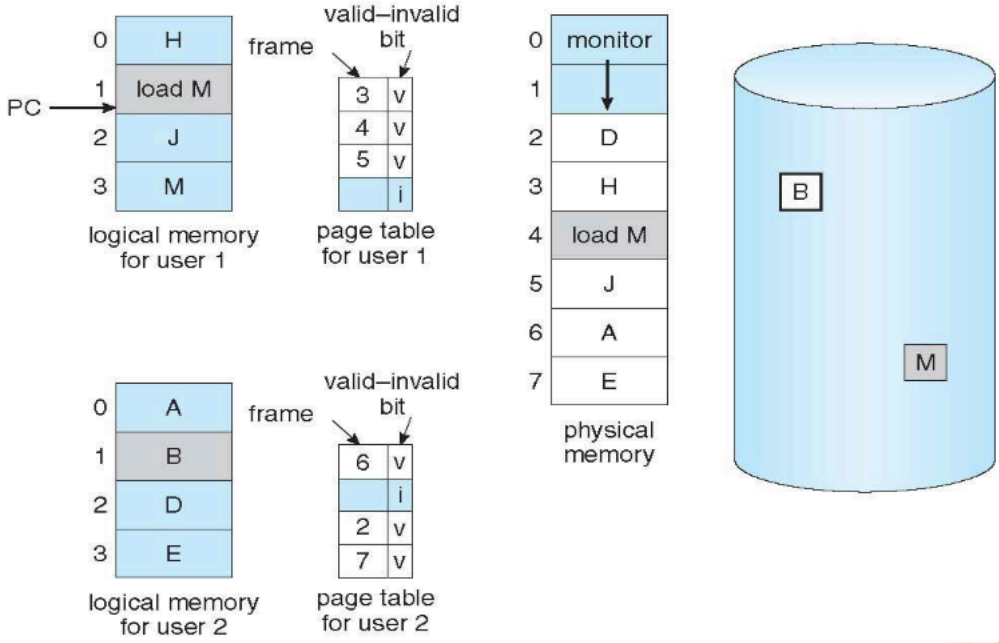
如果没有自由帧会怎样?

- 被进程页用完 内核、I/O 缓冲区等也需要。要为每个进程分配多少内存? - 帧分配算法
- 页面替换 – 在内存中找到一些页面，但并未真正使用，将其分页出来
  - 算法 – 终止进程? 换掉整个流程映像? 替换页面?
  - 性能 – 想要一种将导致最少页面错误数的算法
  - 这需要对流程或程序执行透明
- 注意到同一页可能不可避免地多次进入内存

页面替换

- 通过修改页面错误服务例程以包含页面替换来防止内存过度分配
- 使用修改 （脏） 位来减少页面传输的开销 – 只有修改后的页面才会写回磁盘
- 页面替换完成了逻辑内存和物理内存之间的分离 – 可以在较小的物理内存上支持大型虚拟内存

需要更换页面



基本页面替换

1. 在磁盘上找到所需页面的位置

2. 查找免费框架：

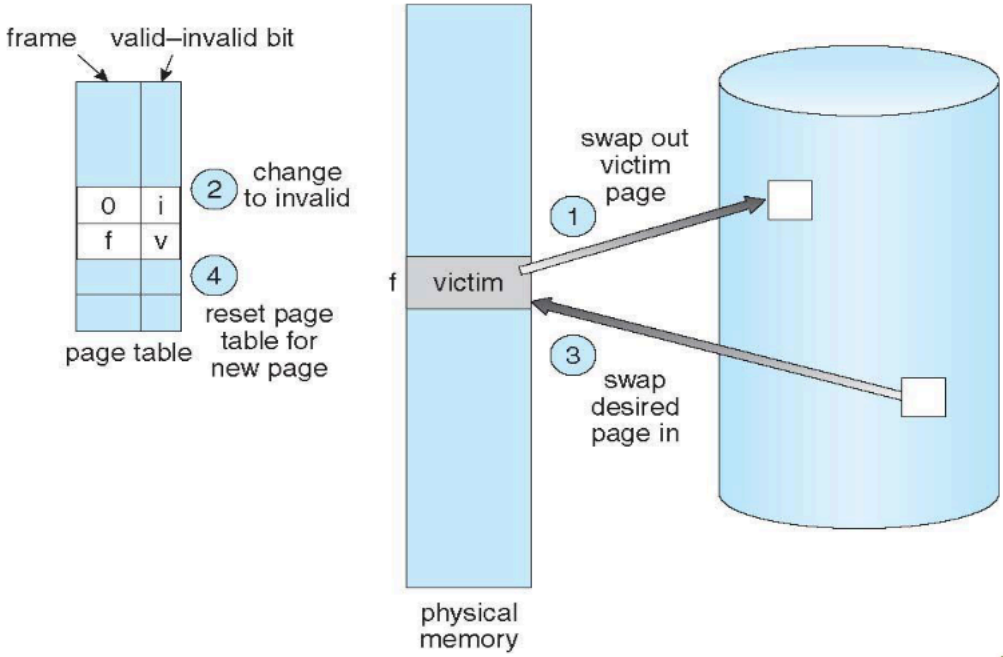
   - 如果有空闲帧，则使用它 - 如果没有空闲帧，请使用页面替换算法选择受害者帧 - 如果"脏"（自上次将其引入内存）

3. 将所需的页面放入（新）自由框架中;更新页表

4. 通过重新启动导致陷阱的指令来继续该过程

请注意，现在可能会因页面错误而进行两次页面传输 - 这可能会显着增加 EAT

页面替换



页面和框架替换算法

- 页面替换算法
  - 决定要替换的帧 – 目标是最大限度地降低页面错误率
- 帧分配算法确定 - 分配给每个进程的帧数 - 这取决于进程如何访问内存 - 位置

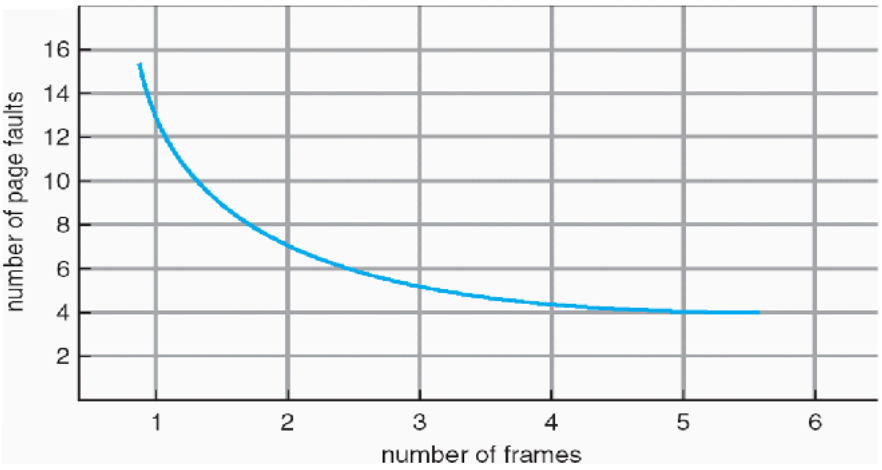
页面和框架替换算法

- 通过在特定的内存引用 - 引用字符串上运行并计算该字符串上的页面错误数来评估算法
  - 首先，对于给定的页面大小，我们只需要考虑页码，而不是完整的物理内存地址。
  - 如果我们引用了页面 p，那么紧随其后的任何对页面 p 的引用都不会导致页面错误。页面 p 将在第一个引用之后位于内存中，因此紧随其后的引用不会有页面错误。
- 例如，对于特定进程，我们可能会记录以下地址序列：

  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- 在每页 100 字节时，此序列将缩减为以下引用字符串：

  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
- 在我们所有的示例中，引用字符串都是

  7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1



-
-

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

-
  -
-
  -