OPERATING SYSTEMS

Virtual Memory & Demand Paging

Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY

Virtual Memory

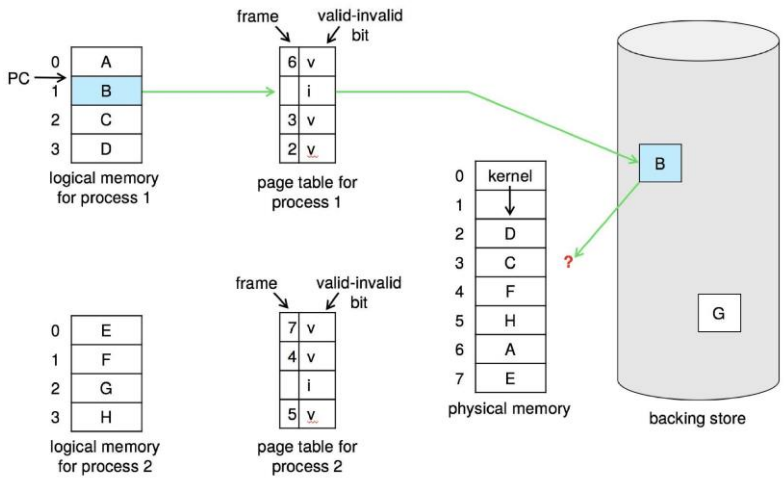
- **目录**
  - 虚拟内存
  - 请求调页
  - 请求调页的考虑因素
  - **页面置换算法**
    - **先进先出算法**
    - **最优算法**
    - **LRU算法**
    - **LRU近似算法**
    - **基于计数的算法**
    - **页面缓冲算法**
    - **清洗策略**
  - **组合分段和分页（段页式）**

Page Replacement Algorithms

- **页面置换**
  - 页面置换(page replacement)在内存中找到一些实际上没在使用的页面，将其移出。
    - 通过修改缺页中断服务例程以包括页面置换，防止内存过度分配。
    - 使用修改位（modify bit, dirty bit, 脏位）减少页面传输的开销–只有修改过的页面才会写回磁盘。
  - 页面置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提供较大的虚拟内存。

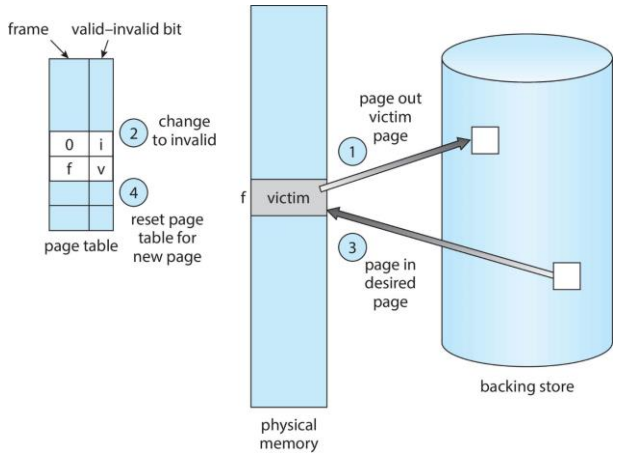
Page Replacement Algorithms

- **页面置换**
  - 页面置换需求



Page Replacement Algorithms

- **页面置换**
  - 基本页面置换



Page Replacement Algorithms

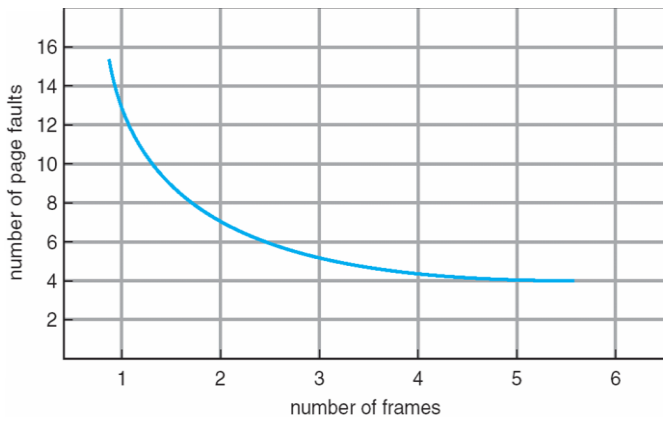
- **页面置换**
  - 基本页面置换
    - ① 在辅助存储器上查找所需页面的位置。
    - ② 查找空闲帧：
      - a) 如果有空闲帧，则使用它。
      - b) 如果没有空闲帧，则使用页面置换算法选择一个牺牲帧。
      - c) 将牺牲帧写入辅助存储器（如果脏）；相应地更改页面和帧表。
    - ③ 将所需页面读入新释放的帧；更改页面和帧表。
    - ④ 从发生缺页中断的位置继续此进程。
    - 注意：现在缺页中断可能会导致2次页面传输–增加EAT

Page Replacement Algorithms

- **页面置换**
  - 帧分配算法决定
    - 为每个进程提供多少帧
    - 要置换的帧。
  - 页面置换算法
    - 希望首次访问和再次访问的缺页中断率最低
  - 通过在特定的内存引用字符串上运行页面置换算法并计算该字符串上的缺页中断数来评估页面置换算法。
    - 字符串只包含页码，而不是完整地址。
    - 重复访问同一页面不会导致缺页中断。
    - 结果取决于可用的帧数。
  - 在我们所有的例子中，我们都使用了一些循环引用字符串。

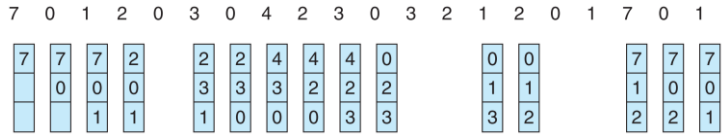
Page Replacement Algorithms

- **页面置换算法**
  - 缺页中断数 vs. 帧数
    - 期望曲线：随着帧数的增加，缺页中断的数量下降到最低水平。



先进先出(FIFO)算法

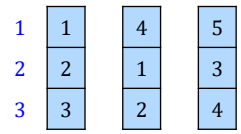
- 先进先出策略
  - 将分配给进程的页帧视为循环缓冲区。
    - 缓冲区已满时，通过先进先出，最旧的页将被置换。
      - 通常一个经常使用的页面是最旧的，因此它会被先进先出(FIFO)反复调出。
    - 易于实现
      - 只需要一个指针在进程的页帧中循环。
  - 实例
    - 引用字符串：7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
    - 3帧



15次缺页中断

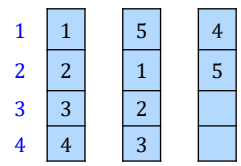
先进先出(FIFO)算法

- Belady异常的例子
  - 引用字符串：1 2 3 4 1 2 5 1 2 3 4 5
  - 3帧（每个进程一次可在内存中存储3页）：



9次缺页中断

  - 4帧：

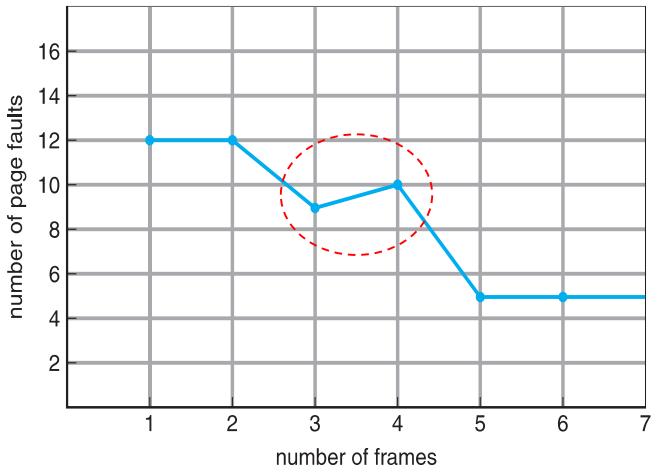


10次缺页中断

  - FIFO置换出现了Belady异常：对于某些页面置换算法，缺页中断率可能会随着分配的帧数的增加而增加。(*László BéLády,* 1969)

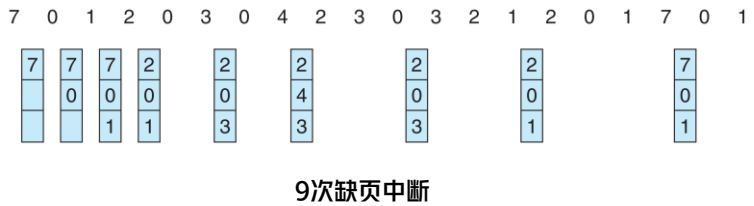
Page Replacement Algorithms

- **先进先出(FIFO)算法**
  - 出现Belady异常的FIFO



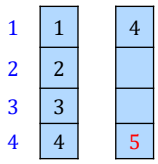
Page Replacement Algorithms

- **最优(OPT)算法**
  - 最优页面置换
    - 最优策略选择在最长时间内不使用的页面进行置换。
      - 不可能实现（需要知道未来），但可以作为与我们将要研究的其他算法进行比较的标准。
    - 实例
      - 引用字符串：7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
      - 3帧
      - 本例的最优选择，有9次缺页中断



最优(OPT)算法

- 实例
  - 引用字符串：1 2 3 4 1 2 5 1 2 3 4 5
  - 4帧：

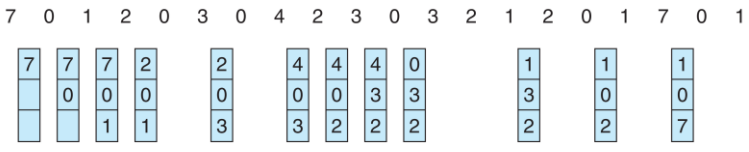


6次缺页中断

  - 你如何知道未来的用途？你不知道！
  - 用于测量算法的性能。

最近最少使用(LRU)算法

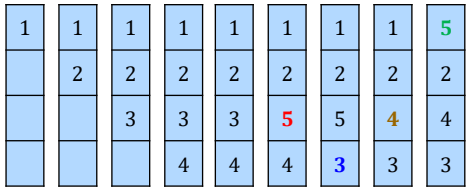
- LRU策略
  - 置换最长时间未被引用的页面。
    - 使用过去的知识而不是未来
    - 将上次使用的时间与每页关联
    - 根据局部性原则，这应该是在不久的将来最不可能被引用的页面。
    - 性能几乎与最优策略一样好。
    - 例子中出现12次缺页中断–比FIFO好，比最优算法差
    - 一般来说这是很好的算法，经常使用



12次缺页中断

Page Replacement Algorithms

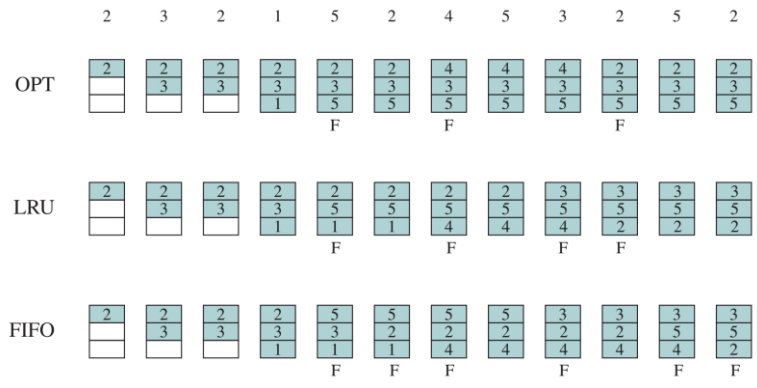
- **最近最少使用(LRU)算法**
  - 实例
    - 引用字符串：1 2 3 4 1 2 5 1 2 3 4 5
    - 4帧：



8次缺页中断

Page Replacement Algorithms

- **最近最少使用(LRU)算法**
  - OPT、LRU和FIFO的比较
    - 示例：一个包含5个页面的进程，操作系统将常驻集大小固定为3。
    - 引用字符串：2 3 2 1 5 2 4 5 3 2 5 2



Page Replacement Algorithms

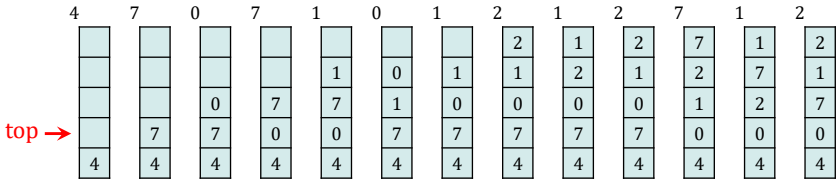
- **最近最少使用(LRU)算法**
  - LRU策略的实现
    - 每个页面都可以用每次内存引用的时间标记（在页表条目中）
    - LRU页面是时间值最早的页面（需要在每个缺页中断处搜索）
    - 这将需要昂贵的硬件和大量的开销。
      - 因此，很少有计算机系统为真正的LRU置换策略提供足够的硬件支持。
      - 取而代之的是其他算法。

Page Replacement Algorithms

- **最近最少使用(LRU)算法**
  - LRU实现
    - 计数器实现
      - 每个页面条目都有一个计数器，用于保存页面引用的时间
        - 每次通过此页面条目引用页面时，将时钟(Clock)复制到计数器中。
      - 当需要置换页面时，查看计数器以确定要置换的页面。

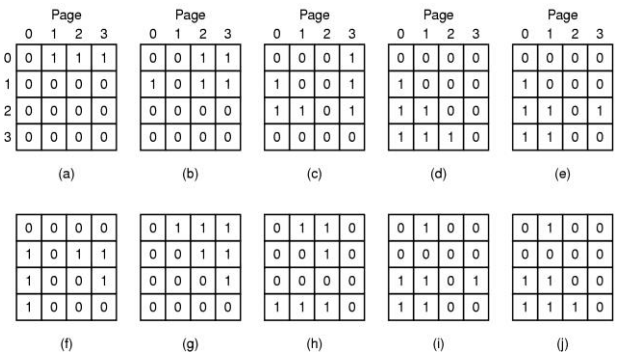
Page Replacement Algorithms

- **最近最少使用(LRU)算法**
  - LRU实现
    - 堆栈实现
      - 以双链接形式保存一个页码堆栈
      - 当引用页面时
        - 把它移到顶端
        - 需要更改6个指针（2指针 × 3个节点）
      - 置换不需要搜索。
      - LRU和OPT是没有Belady异常的堆栈算法的例子。
      - 实例
        - 引用字符串：4 7 0 7 1 0 1 2 1 2 7 1 2



Page Replacement Algorithms

- **最近最少使用(LRU)算法**
  - LRU实现
    - 硬件矩阵LRU实现
      - 页面的引用顺序为0, 1, 2, 3, 2, 1, 0, 3, 2, 3.
      - 当引用第i页时，
        (1) 将第i行的元素设置为1
        (2) 将第i列的元素设置为0
      - 将置换行累加和最小的页面。

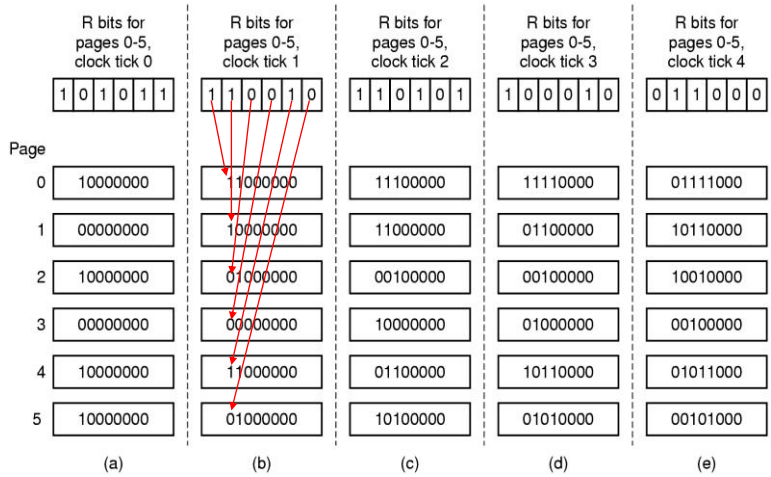


Page Replacement Algorithms

- **LRU近似算法**
  - LRU需要特殊的硬件，速度仍然很慢。
  - 引用位
    - 与每个页面关联的一位，初始值为0.
    - 引用页面时，设置为1。
    - 置换引用位为0的页面（如果存在）
      - 然而，我们不知道真正的使用顺序
  - 引用字节
    - 定期记录引用位；为每页保留一个字节的引用位。
    - 以固定的间隔（例如每20毫秒）定时中断将控制传递到操作系统。操作系统将每个页面的8位字节右移1位，丢弃低位，将引用位移到最高一位。
    - 每个引用字节保存过去八个时间间隔的页面使用（老化）历史记录。
    - 如果我们将引用字节解释为无符号整数，则数字最小的页面就是LRU页面。

Page Replacement Algorithms

- **LRU近似算法**
  - 引用字节
    - 实例：各页的引用位记录到引用字节的最高位

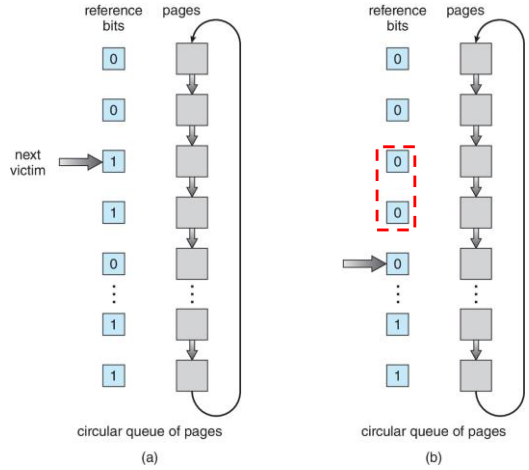


Page Replacement Algorithms

- **LRU近似算法**
  - 第二次机会算法
    - 候选置换帧集被视为循环缓冲区（有时称为时钟算法）。
    - 每当以下情况，每帧的硬件引用位都设置为1：
      - 页面首先加载到帧中，或
      - 相应的页面被引用。
    - 第二次机会置换的基本算法是FIFO。检查页面时，将检查其引用位。
    - 第二次机会策略如下：
      - 如果引用位为0，则选择置换此页面。下一帧将在下一次置换时检查。
      - 否则，该位将被清除为0，页面的到达时间将重置为当前时间（即，页面将有第二次机会）。我们继续检查下一个FIFO页面。因此，在所有其他页面都被置换（或被赋予第二次机会）之前，不会置换被赋予第二次机会的页面。
      - 此外，如果一个页面的使用频率足以保持其引用位的设置，它将永远不会被置换。

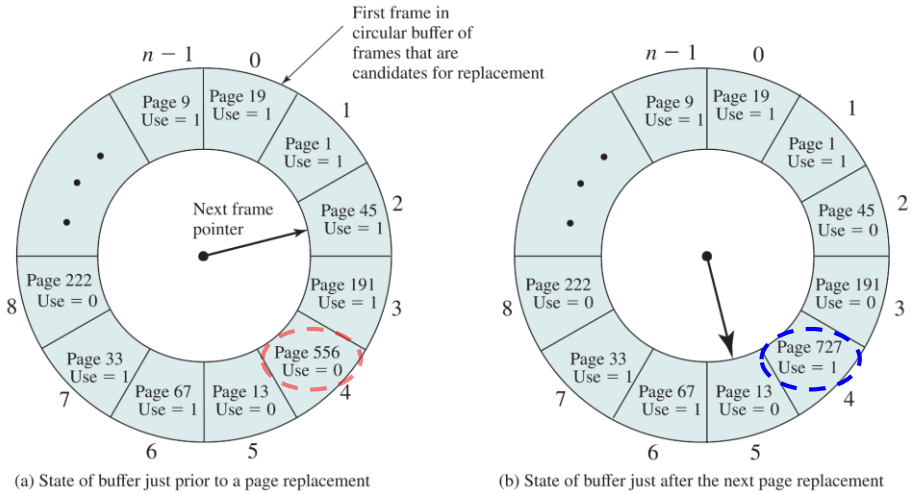
Page Replacement Algorithms

- **LRU近似算法**
  - 第二次机会算法
    - 实例



Page Replacement Algorithms

- **LRU近似算法**
  - 第二次机会算法
    - 时钟策略操作的另一个示例。



(a) State of buffer just prior to a page replacement

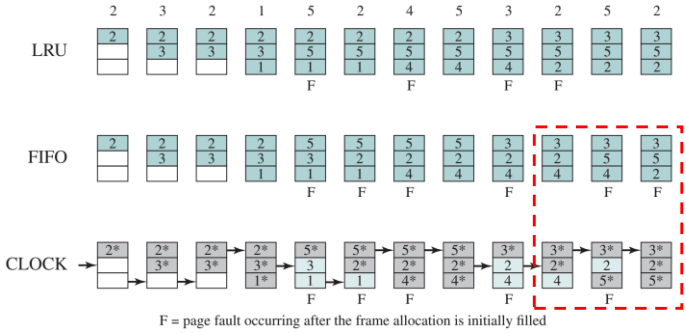
(b) State of buffer just after the next page replacement

Page Replacement Algorithms

- **LRU近似算法**
  - 增强型第二次机会算法
    - 通过将引用位和修改位（如果可用）作为有序对来改进算法
      <引用位，修改位>
    - 有以下四级的类别：
      - 类别0：<0, 0>最近未使用且未修改的 – 要置换的最优页面
      - 类别1：<0, 1>最近未使用但已修改 – 不太好，需要在置换之前把页面写入外存。
      - 类别2：<1, 0>最近使用但未修改 – 可能很快会再次使用。
      - 类别3：<1, 1>最近使用且修改过 – 可能很快会再次使用，需要在置换前写入外存。
    - 当需要页面置换时，使用时钟方案， 置换非空的最低类别中的第一页
      - 可能需要搜索循环队列多次。
    - 优点
      - 为已修改页面赋予更高类别，减少了所需I/O数量

Page Replacement Algorithms

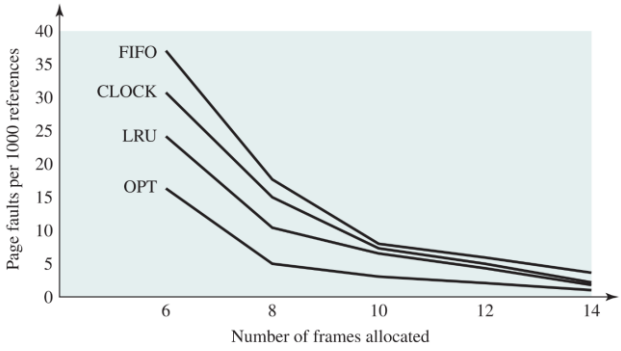
- **LRU近似算法**
  - 时钟与FIFO、LRU的比较
    - 示例：一个包含5个页面的进程，操作系统将常驻集大小固定为3。引用字符串：2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2.



    - 在时钟方案中
      - 星号表示相应的使用位设置为1
      - 箭头指示指针的当前位置
      - 请注意，时钟策略擅长于防止帧2和5被置换（使用位）

Page Replacement Algorithms

- **LRU近似算法**
  - 时钟与FIFO、LRU的比较
    - 固定分配，局部页面置换。
      - 假定分配给进程的页面帧数是固定的。



Page Replacement Algorithms

- **基于计数的算法**
  - 让每一页保留一个计数器，作为该页引用数的累加器。
  - 最不常用(LFU)
    - LFU算法置换最小计数的页面。
      - 其他的过去和将来都会被更多地使用。
  - 最常用(MFU)
    - MFU算法基于这样一个论点，即计数最小的页面可能刚刚被带进主内存，尚未使用。
    - 本算法不太常用。

Page Replacement Algorithms

- **页面缓冲算法**
  - 要置换的页面在主内存中保留一段时间，以防止性能不佳的置换算法，如FIFO。
  - 维护两个指针列表是，若再次被引用则可快速响应
    - 空闲页列表（未修改列表）：已选择要被置换，但自引入后未修改的帧。
      - 不需要换出；修改页列表：已选择要被置换，且修改过的帧。
      - 需要（空闲时）将其写入外存；
  - 由FIFO等算法选择的帧，根据其修改位，把指向其地址的指针追加到以上一个列表的尾部，相应页表条目中的存在位被清除；但页面仍保留在相同的物理内存帧中。
    - 当缺页中断时，引用页不一定会调入覆盖所选页，而是调入到空闲页列表中的第一帧中（请参阅下一张幻灯片）。

页面缓冲算法

- 在每个缺页中断时，首先检查两个列表，以查看所需的页面是否仍在主存中。
  - 如果是，我们只需要在相应的页面表条目中设置存在位，并删除相关列表（空闲页列表或修改页列表）中的匹配条目。
  - 如果不是，则引入所需的页面。它被放置在空闲页列表头部指向的帧中（覆盖该页面）；空闲帧列表的头部将移动到下一个条目。
  - 页表条目中的帧码可用于扫描两个列表，或者每个列表条目可包含进程ID和占用帧的页码。
- 修改页列表还用于聚簇而不是单独写出修改页到外存，并重置修改位，将条目移动到空闲页列表中。

清洗策略

- 修改后的页面应在何时写入磁盘？
- 请求清洗
  - 仅当页帧被选定要置换时，才会写出该页面。
    - 但是出现缺页中断的进程可能必须等待2次页面传输。
- 预先清洗
  - 修改后的多个页面在需要使用其物理帧之前写入，以便可以成批写入。
    - 写回辅存时，对应页可能仍留在主存中。
    - 但是，如果大部分页面在被置换之前会被再次修改，那么写这么多页面就没有什么意义了。

Page Replacement Algorithms

- **清洗策略**
  - 使用页面缓冲可以实现很好的折衷。
    - 回想一下，选择用于置换的页面会保留在空闲（未修改）页列表或修改页列表上。
    - 修改页列表中的页面可以周期性地批量写出，并移动到空闲页列表中。
    - 这是一个很好的折衷方案，因为：
      - 并不是所有的脏页都会被写出，只有那些被选择置换的页面才会被写出。且写出与缺页中断不是同时发生。
      - 写出是聚簇完成的。

Demand Segmentation

- **组合分段和分页（段页式）**
  - 分页和分段的比较。

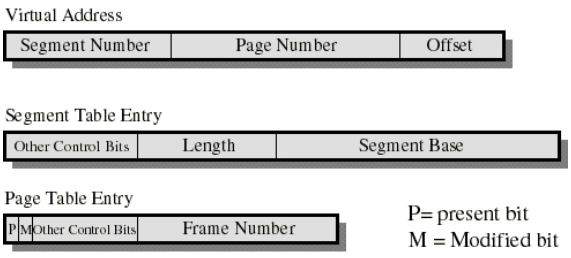
Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Demand Segmentation

- **组合分段和分页（段页式）**
  - 为了结合他们的优势，一些OS对段进行分页。
  - 存在多种组合 – 假设每个进程具有：
    - 一个段表。
    - 多个页表：每个段一个页表
  - 虚拟地址包括：
    - 一个段号：用于索引段表，段表条目给出该段的页表起始地址
    - 一个页码：用于索引页表以获得相应的帧码
    - 一个偏移量：用于在帧内定位单字

Demand Segmentation

- **简单的段页式**
  - 段基址是该段的页表的物理地址。
  - 存在位、修改位仅存在于页表条目中。
  - 保护和共享信息自然存在于段表条目中。
    - 例如：一个只读/读写位，一个内核/用户位……



Demand Segmentation

- **简单的段页式**
  - 段页式的地址转换