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
    - 基本概念
    - 缺页错误和页面置换
    - 请求调页的几个方面
    - 备份/交换存储
    - 转换表(查找)缓冲区(TLB)
    - 请求调页的步骤
    - 请求调页的性能
  - 请求调页的考虑因素
    - 局部性与抖动
    - 内存映射文件
    - 伙伴系统分配器
    - Slab分配器
    - 写时复制
    - 其他问题
  - 页面置换算法
  - 组合分段和分页

Virtual Memory

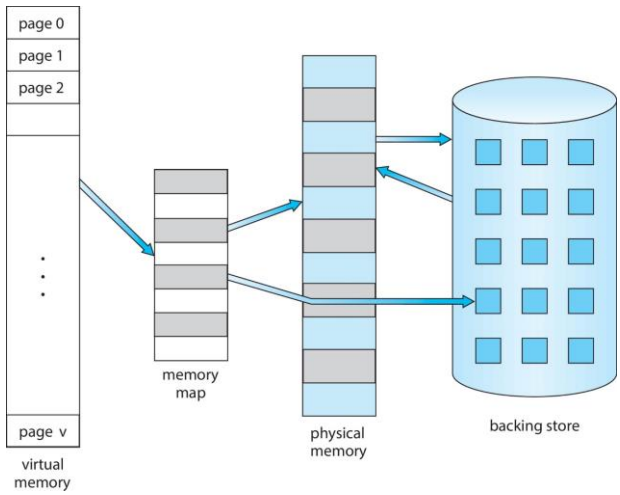
- **虚拟内存**
  - 真正的内存管理将多个进程同时保存在内存中，以允许多道程序。
    - 整个进程应该在内存中才能执行。
  - 虚拟内存是一种允许执行不完全在内存中的进程的技术。
    - 只需要程序的一部分在内存中就能执行。程序可以大于可用的物理内存。
      - 它们主要维持在辅助内存（磁盘）上。
      - 允许更多的程序同时运行。
      - 加载或交换进程所需的I/O更少。
    - 该方案将主内存抽象为一个非常大、统一的存储数组，将用户逻辑内存与物理内存分离。
      - 程序员不再受物理内存可用量限制。
  - 虚拟内存还允许进程共享文件和库，实现共享内存，并为进程创建提供了有效机制。

Virtual Memory

- **虚拟内存**
  - 虚拟内存可以通过以下方式实现：
    - 请求调页
    - 请求调段
  - 基于分页/分段，进程可能被分成不需要在主存中连续定位的片段（页或段）。
    - 基于局部性原则，在执行进程中不需要将进程的所有部分加载到主内存中；所有地址都是虚拟的。

Virtual Memory

- **虚拟内存**
  - 大于可用物理内存的虚拟内存



Virtual Memory

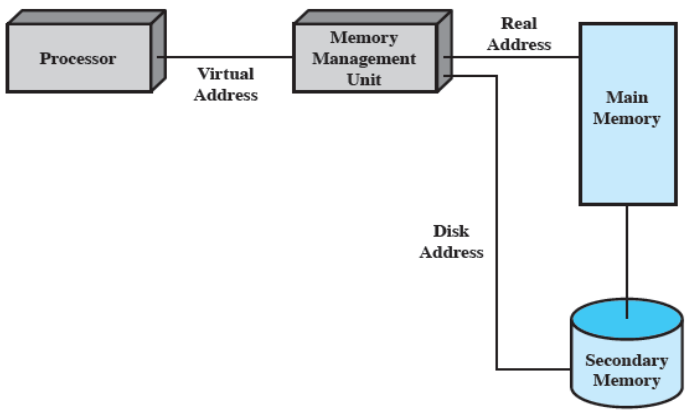
- **虚拟内存**
  - 部分加载的优点
    - 可以在主存中维护更多进程。
      - 仅加载每个进程的某些部分
      - 主内存中的进程越多，在任何给定时间更有可能存在一个进程处于就绪状态
    - 即使一个进程大于主内存大小，现在也可执行。
      - 逻辑地址甚至可以使用更多的位寻址，而不是寻址物理内存所需的位。
    - 实例
      - 寻址64KB的物理内存只需要16位($64\times1024=2^{16}$).
      - 若使用1KB的页面大小，则页面内的偏移需要10位。
      - 我们可以使用大于6位，比如说22位，来虚拟32位即4GB内存。

Virtual Memory

- **虚拟内存**
  - 虚拟内存需要的支持
    - 内存管理硬件必须支持分页和/或分段。
    - 操作系统必须能够管理页面和/或段在外部存储和主内存之间的移动，包括页/段的放置和替换。

Virtual Memory

- **虚拟内存**
  - 虚拟内存寻址

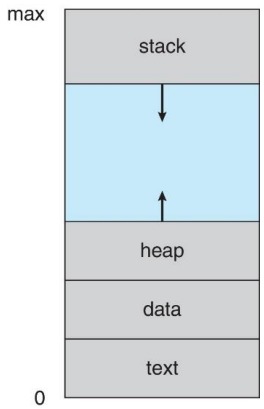


Virtual Memory

- **虚拟内存**
  - 虚拟逻辑地址空间的设计
    - 堆栈从最大逻辑地址开始“向下”增长；而堆从低位“向上”增长
      - 最大限度地利用地址空间
      - 两者之间未使用的地址空间为空
      - 在堆或堆栈增长到给定的新页之前，不需要物理内存
    - 使用有孔的稀疏地址空间，堆或栈增长时或加载动态链接库等时，可填充这些空闲孔。
    - 将系统库映射到虚拟地址空间以实现共享
    - 将页的读与写映射到虚拟地址空间来共享内存
    - 在fork()期间可以共享页面，从而加快进程创建。

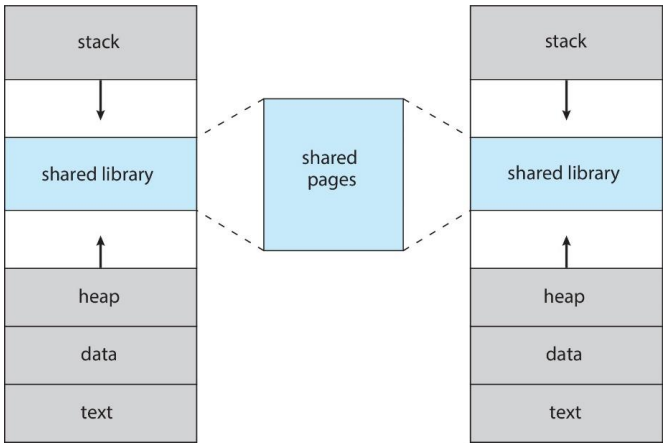
Virtual Memory

- **虚拟内存**
  - 虚拟地址空间



Virtual Memory

- **虚拟内存**
  - 使用虚拟内存的共享库



Virtual Memory

- **虚拟内存**
  - 进程执行
    - 操作系统只将程序的几个部分（包括它的起点）调入主内存。
      - 每个页/段表条目都有一个有效-无效位，仅当对应的页/段在主内存中时才设为有效。
      - 驻留集是进程在某个阶段位于主内存中的部分。
    - 当内存引用位于主存中不存在的部分时，会产生中断（内存错误）。
      - 操作系统将进程置于阻塞状态。
      - 操作系统发出一个磁盘I/O读取请求，将引用的块调入主内存中。
      - 在磁盘I/O发生时，另一个进程被调度运行。
      - 磁盘I/O完成时发出中断；这会导致操作系统将受影响的进程重新置于就绪状态。

Demand Paging

- **基本概念**
  - 仅在需要时才将页面调入内存。
    - 所需的I/O更少，没有不必要的I/O
    - 所需内存更少
    - 更快的响应
    - 更多的用户
  - 需要某页面 ⇒ 引用它
    - 无效访问（不在进程逻辑地址空间） ⇒ 中止
    - 不在内存中 ⇒ 调入内存
  - 类似于带交换的分页系统。
    - 调页程序是处理页面的交换程序。
    - 进程即将换入，在其再次换出前调页程序会猜测它将使用哪些页面。
  - *惰性交换程序* (*Lazy swapper*) – 除非需要页，否则永远不要将页交换到内存中。
    - 调页程序只将进程需要的页面调入内存。

Demand Paging

- **基本概念**
  - 需要新的MMU功能来实现请求调页。
    - 如果所需的页已经驻留在内存中
      - 与非请求调页没有区别
    - 如果所需页不在内存驻留
      - 需要检测页面并将其从存储器加载到内存中
        - 在不改变程序行为的情况下
        - 无需程序员更改代码

Demand Paging

- **基本概念**
  - 有效-无效位
    - 与每个页表条目关联的是一个有效-无效位（存在或不存在）。
      - *v*: 内存中的页面; *i*: 页面不在内存中
    - 初始时，所有条目上的有效-无效位均设置为*i*

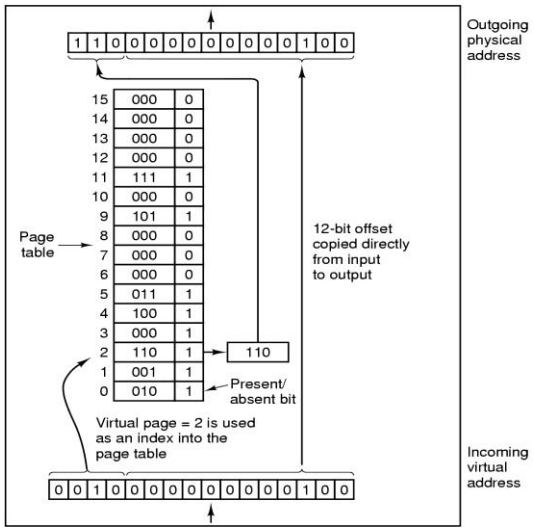
页帧#	v-i位
	v
	v
	v
	v
	i
… …	
	i
	i

页表

  - 在地址转换期间，如果页表条目中的有效-无效位是 *i* 则发生缺页中断(page fault).

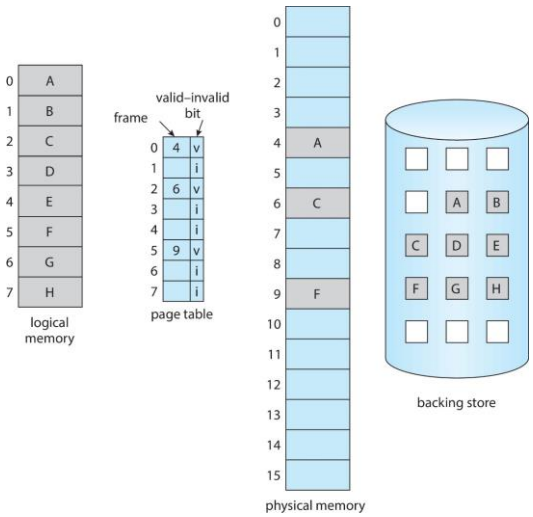
Demand Paging

- **基本概念**
  - 虚拟内存映射示例



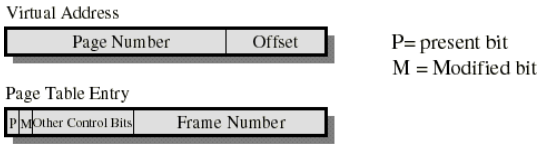
Demand Paging

- **基本概念**
  - 某些页不在主存中时的页表



Demand Paging

- **请求调页动态**
  - 通常，每个进程都有自己的页表。



P= present bit
M = Modified bit

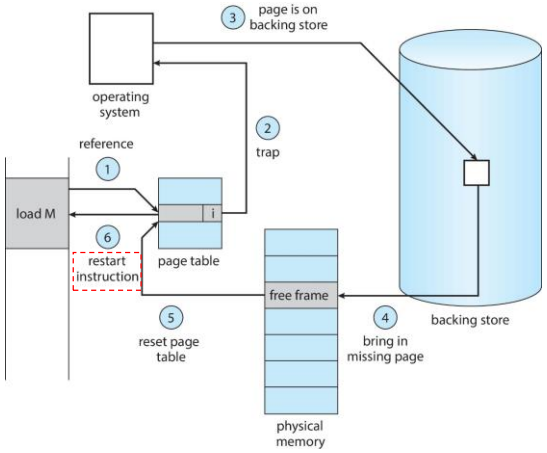
  - 每个页表条目都包含一个存在位(*present bit,* 有效无效位)，用于指示页是否在主存中。
    - 如果在主存中，则条目包含主存中相应页面的帧码。
    - 如果它不在主存中，则条目可能包含磁盘上该页的地址，或者可以使用页码索引另一个表（通常在PCB中），以获取磁盘上该页的地址。

Demand Paging

- **请求调页动态**
  - 修改位(Modified bit)表示页面自上次加载到主存后是否已被更改。
    - 如果未进行任何更改，则在需要换出页面时，不必将页面写回磁盘。
  - 如果在页面级别管理保护，则可能存在其他控制位。
    - 只读或读写位
    - 保护级别位：内核页或用户页（当处理器支持超过2个保护级别时，使用更多位）

Demand Paging

- **缺页中断**
  - 如果对某个不在内存中的页面进行引用，则对该页面的第一次引用将引发操作系统陷阱：这是一个缺页中断(page fault).



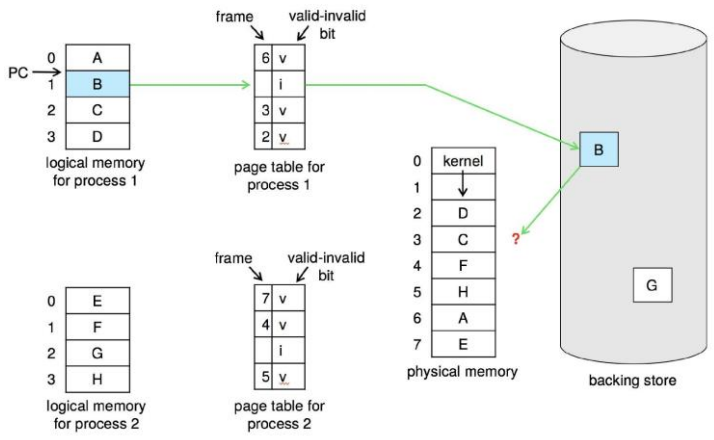
处理缺页中断的步骤

Demand Paging

- **缺页中断**
  - 处理缺页中断的步骤：
    (1) 如果对某个不在内存中的页面进行引用，则对该页面的第一次引用将陷入操作系统，导致缺页中断。
    (2) 缺页中断由相应的操作系统服务例程处理。OS查找另一个表来决定：
      - 无效引用 ⇒ 中止
      - 只是不在内存中
    (3) 在磁盘上（在文件或备份存储中）找到所需页面
    (4) 找到一个空闲的页帧；通过调度磁盘操作将页面换入空闲帧
    (5) 重置页表以指示现在内存中存在该页：
      - 设置有效-无效位为 v
    (6) 重新执行导致缺页中断的指令

Demand Paging

- **页面置换**
  - 如果没有空闲帧会发生什么？



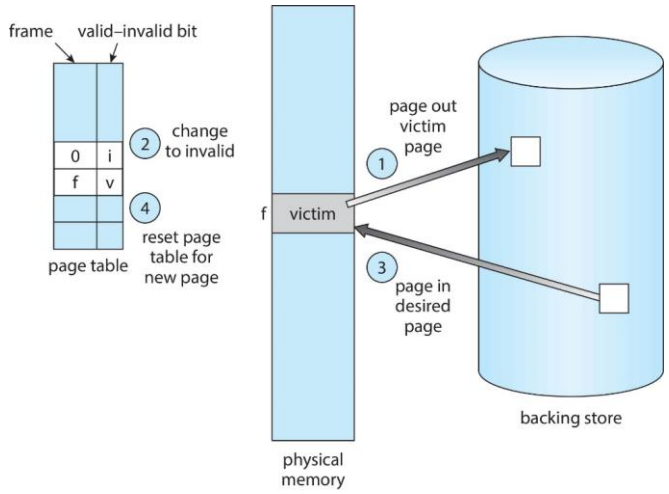
需要页面置换

Demand Paging

- **页面置换**
  - 如果没有空闲帧会发生什么？
    - 页面置换(Page replacement)
      - 在内存中找到一些未真正在用的页面，然后将其调出
        - 页面置换算法
    - 性能
      - 需要一个算法，使产生缺页中断的次数最小化
    - 同一页可能会多次调入内存

Demand Paging

- **页面置换**
  - 处理页面置换的步骤



Demand Paging

- **页面置换**
  - 处理页面置换的步骤
    - (1) 在磁盘上找到所需页面的位置。
    - (2) 查找空闲帧：
      - 如果有空闲帧，就使用它。
      - 如果没有空闲帧，则使用页面置换算法选择牺牲页(victim frame).
        - 牺牲页不应正被使用。
    - (3) 将所需页面读入（新的）空闲帧；更新页表和帧表。
    - (4) 继续缺页中断处理。

Demand Paging

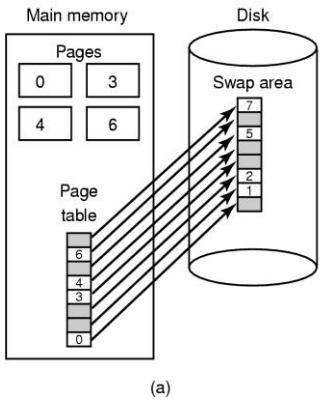
- **页面置换**
  - 关于页面置换的评论
    - 通过修改缺页中断服务例程以包括页面置换，防止内存过度分配
    - 使用修改位（脏位）减少页面传输的开销
      - 只有修改过的页面才会写回磁盘。
    - 页面置换完成了逻辑内存和物理内存之间的分离。
      - 可以在较小的物理内存上提供较大的虚拟内存。

Demand Paging

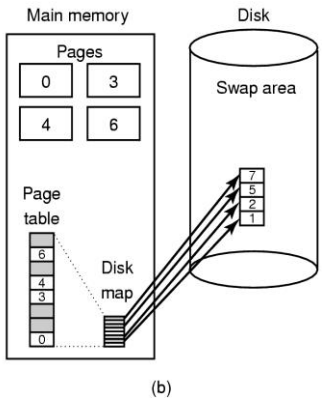
- **请求调页的几个方面**
  - 极端情况–在内存中没有页面的情况下启动进程。
    - 这是纯请求调页。
    - 操作系统将指令指针设置为进程的第一条指令，非内存驻留将导致缺页中断。
      - 且，对每个其他进程页的第一次访问时
  - 实际上，一条给定的指令可能访问多个页面并导致多个缺页中断?
    - 考虑获取并译码一个ADD指令，它对内存中2个数字相加并把结果保存回内存中，访问了三个地址。
      - 因局部引用(locality of reference)，实际上最多只需一次调页，性能影响不大
  - 请求调页所需的硬件支持。
    - 具有有效-无效位的页表
    - 外存（辅助存储，带交换空间的交换设备）
    - 指令重启机制

Demand Paging

- **备份/交换存储**



(a) 分页到静态交换区



(b) 动态备份页面

Demand Paging

- **转换表缓冲区(TLB)**
  - 由于页表位于主存中，每个虚拟内存引用都会导致至少两次物理内存访问：
    - 一次用于获取页表条目
    - 一次用于获取数据
  - 为了解决这个问题，为页表条目设置了一个特殊的高速缓存，称为转换表(查找)缓冲区(TLB, 快表)：
    - TLB包含最近使用的页表条目。
    - 它的工作原理类似于主内存缓存。
  - 实例

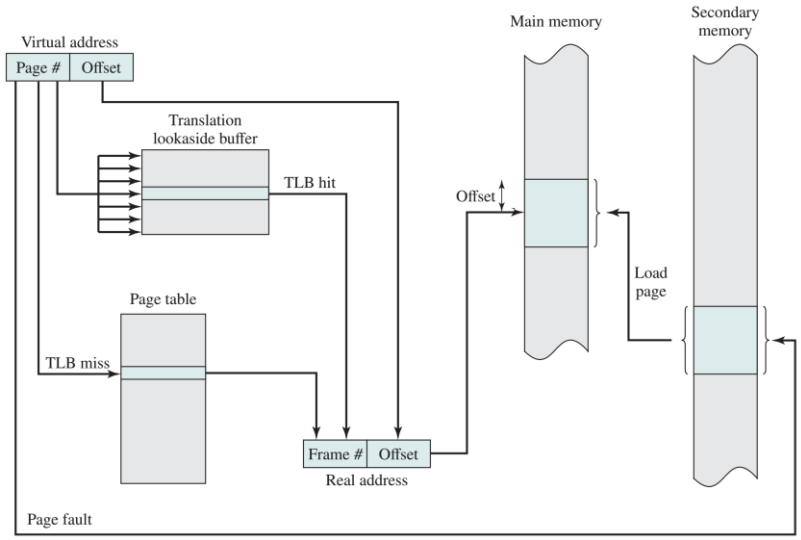
Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Demand Paging

- **TLB动态**
  - 给定一个逻辑地址，处理器检查TLB
    - 如果存在页表条目（命中），则检索帧码并形成真实物理地址
    - 如果找不到页表条目（未命中），则使用页码索引进程页表：
      - 如果设置了有效位，则访问相应的帧
      - 如果没有，则发出缺页中断以将引用的页面读入主内存
  - TLB将更新以包含新的页面条目

Demand Paging

- **TLB动态**
  - TLB的使用



Demand Paging

- **请求调页的步骤（坏的情况）**
  - (1) 陷入操作系统（陷阱Trap）
  - (2) 保存用户寄存器和进程状态
  - (3) 确定中断是一个缺页中断
  - (4) 检查页面引用是否合法，并确定页面在磁盘上的位置
  - (5) 向空闲帧发出从磁盘读取的命令：
    - (a) 在此设备的队列中等待，直到读取请求得到服务
    - (b) 等待设备寻道和/或延迟时间
    - (c) 开始将页面传输到空闲帧
  - (6) 等待时，将CPU分配给其他用户进程（CPU调度，可选）
  - (7) 从磁盘I/O子系统接收中断（I/O完成）
  - (8) 为其他进程保存寄存器和进程状态（如果执行了第6步）
  - (9) 确定中断来自上述磁盘
  - (10) 更正页表和其他表以显示页现在在内存中
  - (11) 等待CPU再次分配到本进程
  - (12) 恢复用户寄存器、进程状态和新页表，再重新执行被中断的指令

Demand Paging

- **请求调页的性能**
  - 三个主要耗时活动：
    - 为中断服务
      - 仔细的编码可减少到几百条指令
    - 读入页面
      - 很长时间...
    - 重新启动进程
      - 再一次，只是一小部分时间
  - 设p为缺页中断的概率($0 \le p \le 1$).
    - 如果p=0，则无缺页中断。
    - 如果p=1，则每个引用都会导致缺页中断。
    - 缺页中断服务时间 = 缺页中断开销
      + 换出页面
      + 换入页面
      + 重新启动开销
  - 有效访问时间(EAT)

    EAT = (1–p) x 内存访问时间 + p x 缺页中断服务时间
    - 为简单起见，我们忽略了TLB查找时间 ε.

Demand Paging

- **请求调页的性能**
  - 实例
    - 内存访问时间 = 200纳秒
    - 平均缺页中断服务时间 = 8毫秒

      ```
      EAT = (1-p) x 200ns + p x 8ms
          = (1-p) x 200 + p x 8000000(ns)
          = 200 + 7999800p（纳秒）。
      ```
    - 如果1000次访问中有一次导致缺页中断，那么 p = 0.001

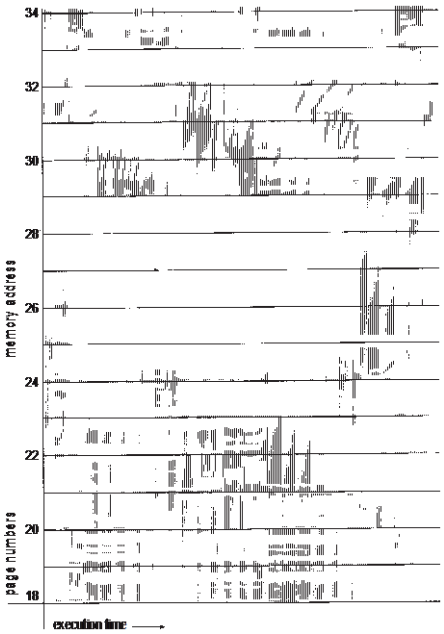
      ```
      EAT = 8199.8ns = 8.2μs
      ```
      - 这是一个减速因子（倍数）

        ```
        8.2μs/200ns = 41
        ```
    - 如果我们希望性能降低 < 10%, 则

      ```
      EAT/200 < (1+0.1)
      (200+7999800p) / 200 < (1+0.1)
      ```
      缺页中断概率 $p < 2.5 \times 10^{-6}$

Paging Considerations

- **内存引用模式中的局部性**
  - 程序内存访问模式有：
    - 时间局部性
    - 空间局部性
  - 沿着给定时间片访问的一组页面称为“工作集”。
  - 工作集定义了进程正常运行所需的最小页数。



Paging Considerations

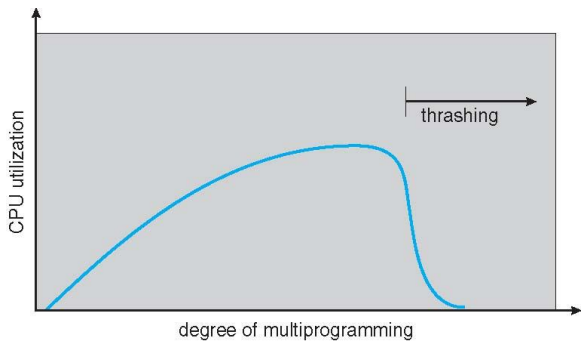
- **局部性与虚拟内存**
  - 内存引用的局部性原则
    - 进程中的内存引用倾向于聚簇(cluster).
  - 因此，在短时间内只需要进程的一小部分。
    - 对未来需要哪些部分进行智能的猜测是可能的。
    - 这表明虚拟内存可以有效地工作（即抖动不会太频繁发生）

Paging Considerations

- **抖动的可能性**
  - 为了容纳尽可能多的进程，主内存中只维护每个进程的一小部分。
  - 如果主内存中一个进程没有“足够的”的页，它必须
    - 产生缺页中断以获取页面
    - 如果主内存过度分配(over-allocated)，则需置换一些现有帧
    - 但可能很快就需要把换出的页重新换入
  - 抖动(Thrashing, 振荡)
    - 处理器将大部分时间用于交换数据块，而不是执行用户指令。
  - 这导致：
    - CPU利用率低
    - 操作系统需要考虑增加多道程序的程度
    - 将更多进程添加到系统中

Paging Considerations

- **抖动的可能性**



Paging Considerations

- **局部性与抖动**
  - 为什么请求调页有效？
    - 局部性模型：
      - 进程由多个局部组成
      - 进程从一个局部移到另一个局部
      - 局部可能重叠
  - 为什么会发生抖动？
    - 局部的总大小 > 内存总大小

Paging Considerations

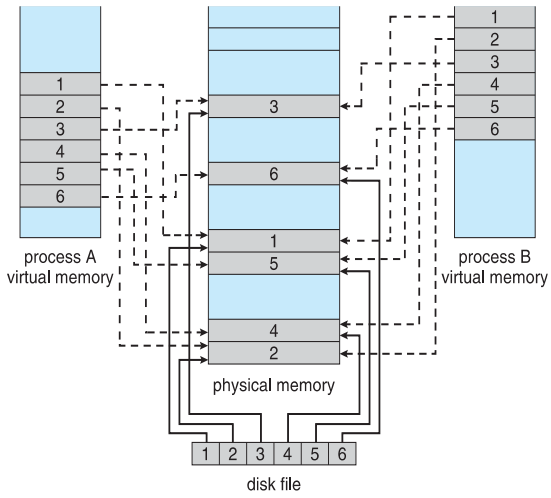
- **内存映射文件**
  - 内存映射文件I/O通过将磁盘块映射到内存中的页面，允许将文件I/O视为常规内存访问。
  - 最初使用请求调页读取文件。
    - 从文件系统的文件读入页面大小的数据块到物理内存页面
    - 对文件的后续读取（或写入）被视为普通内存访问
  - 这简化和加速了文件访问，通过在内存中驱动文件I/O而不是通过read()和write()系统调用。
    - 多个进程映射同一个文件时，可实现内存中页面的共享
  - 但是，写入的数据什么时候进入磁盘？
    - 定期和/或在文件close()时
    - 例如，当调页程序扫描脏页时

Paging Considerations

- **内存映射文件**
  - 一些操作系统将内存映射文件用于标准I/O
    - 进程可以通过mmap()系统调用显式请求内存映射一个文件。
      - 现在，文件被映射到进程地址空间。
    - 对于标准I/O (open(), read(), write(), close()), 也使用mmap方式
      - 不过，是将文件映射到内核地址空间。
      - 用户进程仍然执行read()和write()
        - 在内核空间和用户空间之间复制数据
      - 使用高效的内存管理子系统
        - 避免需要单独的子系统
    - 写时复制(COW, Copy-on-Write)可用于读/写非共享页面。
    - 内存映射文件可以用于共享内存（内存驱动或系统调用方式）

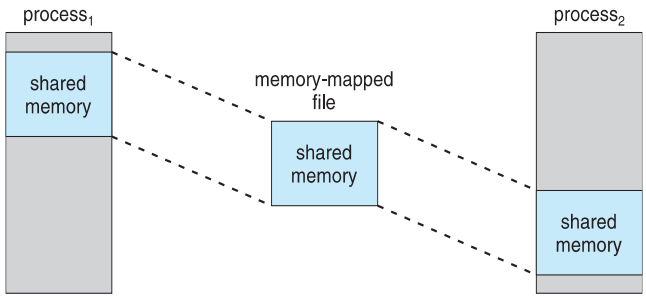
Paging Considerations

- **内存映射文件**



Paging Considerations

- **内存映射文件**
  - 通过内存映射I/O共享内存。



Paging Considerations

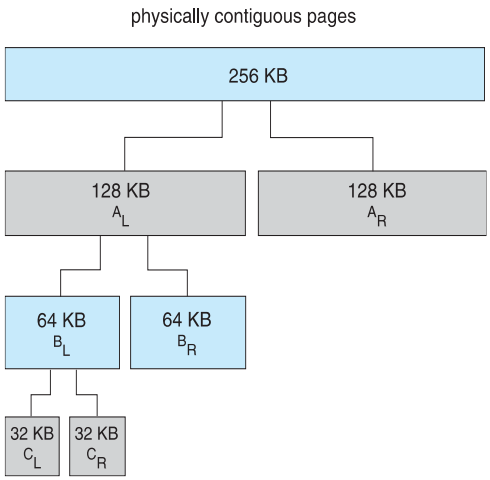
- **内存映射文件**
  - Windows API中的共享内存
    - 首先为要映射的文件创建文件映射
      - 然后在进程的虚拟地址空间中建立映射文件的视图
    - 考虑生产者/消费者
      - 通过生产者使用内存映射功能创建共享内存对象
      - 通过CreateFile()打开文件，返回句柄
      - 通过CreateFileMapping()创建映射创建命名共享内存对象
      - 通过MapViewOfFile()创建视图
    - 查阅课本中的示例代码

Paging Considerations

- **伙伴系统分配器**
  - 伙伴系统从由物理连续页组成的固定大小段分配内存。
  - 使用2的幂分配器分配内存
    - 满足以2的幂为单位的请求
    - 请求四舍五入到2的更高次幂
    - 当需要分配的空间小于等于当前可用块的一半时，当前块将分成两个2的低一次幂的伙伴。
      - 重复直到找到合适大小的可用块
  - 例如，假设256KB块可用，内核请求21KB
    - 拆分为每个128KB的A_L和A_R
      - 其中一块进一步分为64KB的B_L和B_R
        - 其中一块再次分为32KB的C_L和C_R—选一个来满足请求
  - 优点
    - 快速将空闲块合并成更大的块
  - 缺点
    - 碎片

Paging Considerations

- **伙伴系统分配器**

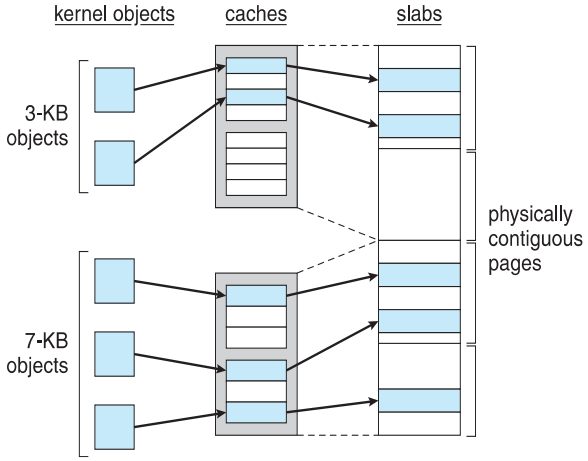


Paging Considerations

- **Slab分配器**
  - Slab(板)分配是分配内核内存的另一种策略。
    - Slab由一个或多个物理上连续的页组成。
    - 缓存由一个或多个Slab组成。
    - 每个独特的内核数据结构都有一个单独的缓存
      - 例如，用于表示进程描述符的数据结构的单独缓存、用于文件对象的单独缓存、用于信号量的单独缓存等等。
  - 每个缓存都填充有对象，这些对象是缓存所代表的内核数据结构的实例化。
    - 例如，表示信号量的缓存存储信号量对象的实例，表示进程描述符的缓存存储进程描述符对象的实例，等等。

Paging Considerations

- **Slab分配器**
  - Slab、缓存和内核对象之间的关系
    - 两个3KB大小和三个7KB大小的内核对象，保存在单独的缓存中



Paging Considerations

- **Slab分配器**
  - 当内核为一个对象从slab分配器请求内存时，slab分配器首先尝试找到一个由空闲对象组成的slab来满足请求。
    - 如果不存在空闲对象，则从空slab指定空闲对象。
    - 如果没有可用的空slab，则从连续的物理页分配新slab并将其分配给缓存；对象的内存是从这个新slab分配的。
  - slab分配器提供两个主要优点：
    - 不会因碎片而浪费内存。
      - 每个独特的内核数据结构都有一个关联的缓存，每个缓存由一个或多个slab组成，这些slab被划分为与所表示的对象大小相同的块。
    - 内存请求可以很快得到满足。
      - 在频繁分配和释放对象时，slab分配方案对于管理内存特别有效。对象是预先创建的，因此可以从缓存中快速分配；在缓存中释放时，只需标记为free即可。

Paging Considerations

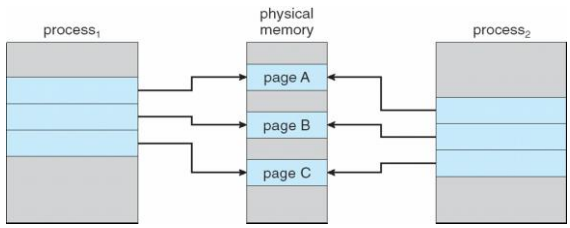
- **Slab分配器**
  - Linux中的Slab分配器
    - 示例：Linux中的进程描述符的类型为struct task_struct
      - 内存大小大约1.7KB
      - 新建任务 → 从缓存分配新 struct task_struct
      - 将使用现有的空闲 struct task_struct
    - slab可以处于三种可能状态之一
      - Full – 所有对象都已使用
      - Empty – 所有对象都空闲
      - Partial – 空闲对象和已使用对象的混合
    - 根据要求，slab分配器
      (1) 使用Partial slab中的空闲struct task_struct
      (2) 如果没有，则从空闲slab中取出一个
      (3) 如果没有空闲slab，则创建新的空slab.

Paging Considerations

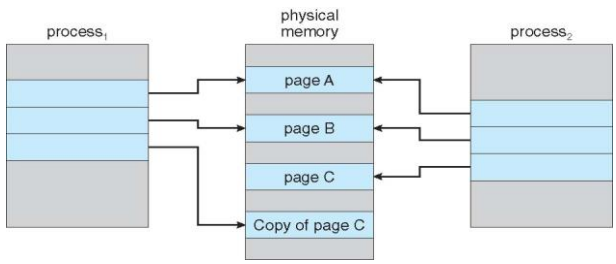
- **写时复制**
  - 写时复制(COW, Copy-on-Write)允许父进程和子进程初始时共享内存中的相同页面。
    - 只在任一进程写入共享页面时，才创建该页面的副本
  - 因为只复制修改过的页面，COW实现了更高效的进程创建。
  - 通常，空闲页面从按需零填充(zero-fill-on-demand)页面池中分配。
    - 页面池应该总是有空闲的帧来加速请求调页的执行
    - 不希望必须释放帧以处理缺页中断

Paging Considerations

- **写时复制**
  - 进程1修改C页前



  - 进程1修改C页后



Paging Considerations

- **其他问题**
  - 预调页
    - 预调页有助于减少进程启动或恢复时出现的大量缺页中断。
      - 在引用某个进程所需的全部或部分页面之前，对其进行预调页。
      - 如果预调页未使用，则会浪费I/O和内存。
    - 假设s页已预调入，并且使用了这些页中的一小部分α：
      - 节省的处理 $s \times \alpha$个缺页中断的成本大于或小于预调入$s \times (1-\alpha)$不必要页面的成本？
      - α接近零 $\Rightarrow$ 预调页亏本

Paging Considerations

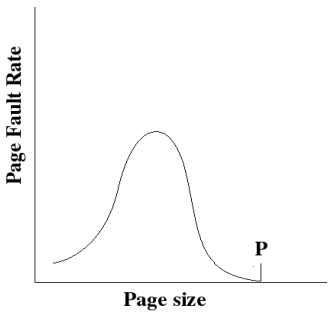
- **其他问题**
  - 页面大小
    - 有时操作系统设计者可以选择页面大小
      - 尤其是在定制的CPU上运行时。
    - 选择不同的页面大小必须考虑：
      - 碎片
      - 页表大小
      - I/O开销
      - 缺页中断数
      - 局部性
      - TLB的大小和有效性
    - 始终为2的幂，通常在2^{12}(4KB)到2^{22}(4MB)之间
    - 平均而言，随着时间在增长。

Paging Considerations

- **其他问题**
  - 页面大小
    - 页面大小由硬件定义；很难精确使用完整个页面。
    - 较大的页面大小是好的，因为对于较小的页面大小，每个进程需要更多的页面；每个进程的页面越多，意味着页表越大。
      - 虚拟内存中页表占了较大部分
    - 较大的页面大小是很好的，因为磁盘设计用于高效地传输大块数据。
    - 较大的页面大小意味着主内存中的页面更少；这会增加TLB命中率。
    - 较小的页面大小有助于最小化内部碎片。

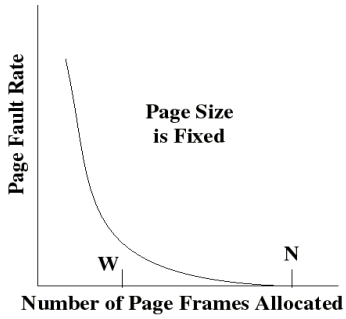
Paging Considerations

- **其他问题**
  - 页面大小
    - 对于非常小的页面大小，每个页面都与实际使用内存的代码相匹配。缺页中断率低（内存有效使用）。
    - 增加页面大小会导致每个页面包含更多未使用的代码。缺页中断率上升。
    - 如果我们接近P点，页面大小等于整个进程的大小，缺页中断率就会降低。



Paging Considerations

- **其他问题**
  - 页面大小
    - 缺页中断率也由每个进程分配的帧数决定。
    - 分配W帧时，缺页中断降至合理值。
    - 当帧数达到(N)某个进程完全在内存中时，将下降到0.



Paging Considerations

- **其他问题**
  - 页面大小
    - 最常用的页面大小为1KB到4KB。页面大小的增加与磁盘块(block)大小的增加趋势有关。
    - 页面大小并非微不足道，一些处理器支持多种页面大小：
      - 奔腾支持两种大小：4KB或4MB
      - MIPS R4000支持7种大小：4KB到16MB

Computer	Page Size
Atlas	512 48-bit words
Honeywell-Multics	1024 36-bit words
IBM 370/XA and 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
UltraSPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes or 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

Paging Considerations

- **其他问题**
  - TLB范围
    - TLB范围(Reach)是可从TLB访问的内存量。
    - TLB范围 = TLB大小 x 页面大小
      - 理想情况下，每个进程的工作集都存储在TLB范围中。
    - 增加TLB的大小
      - 可能很贵
    - 增加页面大小
      - 参考前面“其他问题-页面大小”
    - 提供多种页面大小：
      - 这使得需要较大页面大小的应用程序有机会在不增加碎片的情况下使用大页面。

Paging Considerations

- **其他问题**
  - 程序结构
    - 实例
      - 在C++中：

```
int A[][] = new int[1024][1024];
```

假设每行存储在一个页面中。

- 方案1：

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i][j] = 0;
```

  - 可能产生1024 x 1024个缺页中断。
- 方案2：

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i][j] = 0;
```

  - 可能产生1024个缺页错误。

Paging Considerations

- **其他问题**
  - I/O联锁
    - I/O联锁–页面有时必须锁定在内存中。
      - 用于从设备复制文件的页面必须锁定，以免被其他运行中的进程因缺页中断处理时把它换出。