OPERATING SYSTEMS

Memory Segmentation & 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

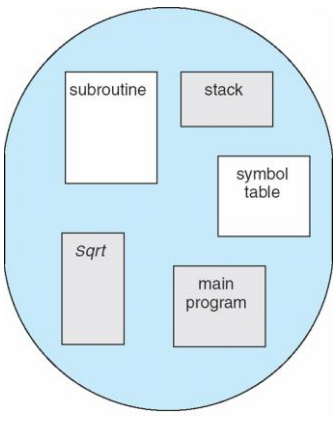
- 目录
  - 简单分段
  - 简单分页
  - 地址转换
  - 页表
  - 例子

Simple Segmentation

- **分段概念**
  - 分段是支持内存用户视图的内存管理方案。
  - 程序是段的集合。
    - 段是一个逻辑单元，例如：
      - 主程序
      - 程序
      - 函数
      - 方法
      - 对象
      - 局部变量，全局变量
      - 公共块
      - 堆栈
      - 符号表
      - 数组

Simple Segmentation

- **分段概念**
  - 程序的用户视图



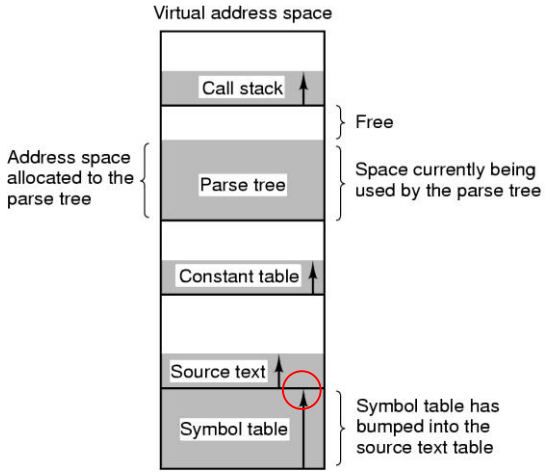
逻辑地址空间

Simple Segmentation

- **分段概念**
  - 程序的用户视图–示例。
    - 编译器有许多在编译过程中建立的表，可能包括：
      - 为打印列表保存的源文本（在批处理系统上）。
      - 符号表–变量的名称和属性。
      - 包含所用整数、浮点数和常量的表。
      - 解析树(语法树)，程序的语法分析。
      - 编译器中用于过程调用的堆栈。

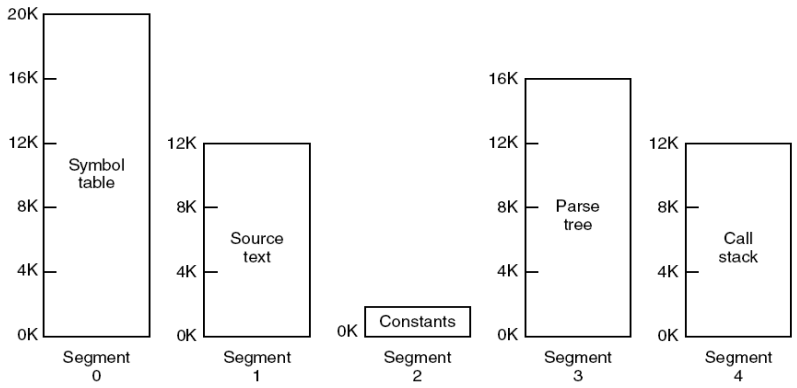
Simple Segmentation

- **分段概念**
  - 一维地址空间
    - 在表不断增长的一维地址空间中，一个表可能会撞到另一个表。



Simple Segmentation

- **分段概念**
  - 分段解决方案
    - 分段允许每个表独立于其他表进行增长或收缩。

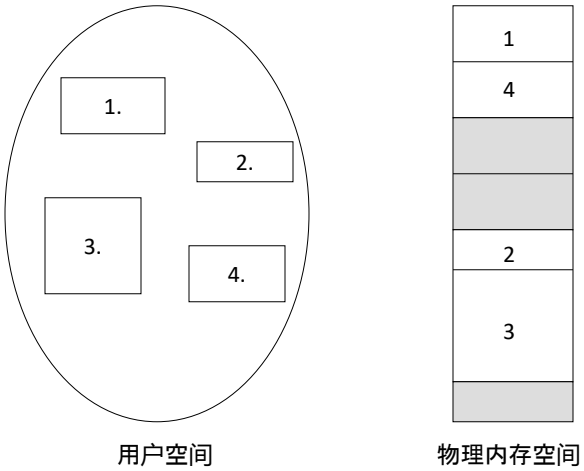


Simple Segmentation

- **简单分段的动态**
  - 每个程序被细分为大小不等的块，称为段(segment).
  - 当进程加载到主内存中时，它的不同段可以位于任何位置。
  - 每个段都有完整的指令/数据；没有内部碎片。
  - 存在外部碎片；当使用小段时，它会减少。
  - 与分页相反， 分段对程序员可见。
    - 为逻辑地组织程序提供便利（例如，数据在一段，代码在另一段中）。
    - 必须知道段大小的限制。
  - 操作系统为每个进程维护一个段表。每个条目包含：
    - 段的起始物理地址
    - 段的长度（用于保护）

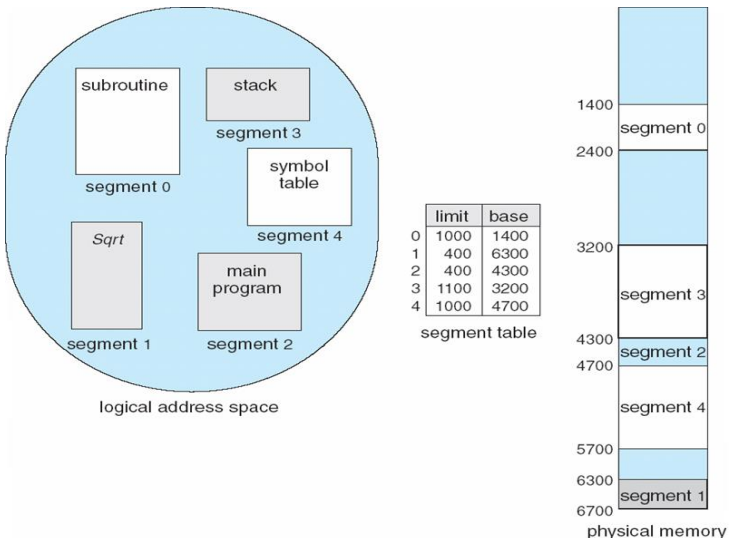
Simple Segmentation

- 简单分段的逻辑视图



Simple Segmentation

- 分段中的地址转换
  - 实例



Simple Segmentation

- **分段体系结构**
  - 逻辑地址由一个二元组组成：

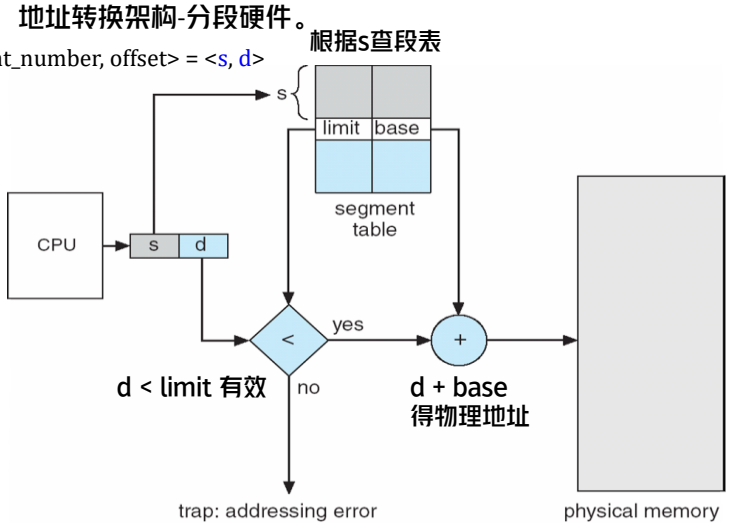
    <segment_number, offset>

    段号　　偏移
  - *段表* – 映射二维物理地址；每个表条目都有：
    - *base* – 包含段驻留在内存中的起始物理地址。
    - *limit* – 指定段的长度。
  - *段表基址寄存器*（*STBR*）指向段表在内存中的位置。
  - *段表长度寄存器*（*STLR*）指示程序使用的段数
    - 如果段号 s < STLR，则段号s是合法的。

Simple Segmentation

- **分段体系结构**
  - 地址转换架构-分段硬件。

<segment_number, offset> = <s, d>



Simple Segmentation

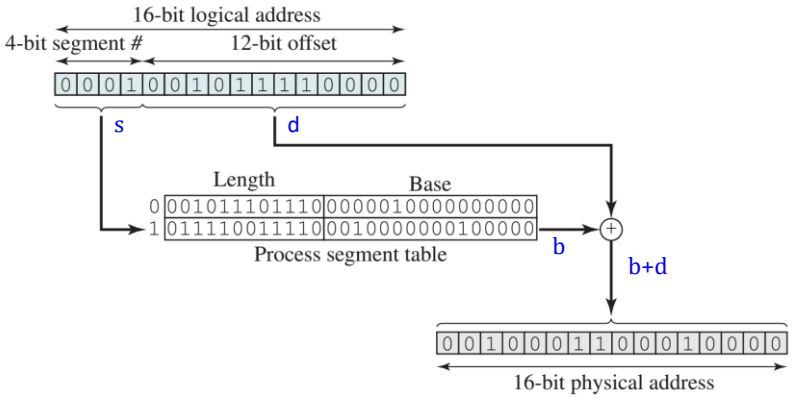
- **分段体系结构**
  - 分段中的地址转换
    - 当进程进入运行状态时，将加载一个专用寄存器，其中包含进程段表的起始地址。
    - 以逻辑地址呈现

      <segment_number, offset> = <s, d>,

      CPU对段表进行索引（使用s），以获取该段的起始物理地址base和长度限制/界限limit。
    - 物理地址通过将d与base相加获得。
      - 硬件还将偏移/位移d与该段的界限limit进行比较，以确定地址是否有效。

Simple Segmentation

- **分段体系结构**
  - 分段中的地址转换
    - 实例



Simple Segmentation

- **分段体系结构**
  - 保护
    - 在段表中的每个条目，包含：
      - 验证位=0 ➔ 非法段
      - 读/写/执行权限。
  - 为段增加保护位；代码共享发生在段级别。
  - 由于段的长度不同，内存分配是一个动态存储分配问题。

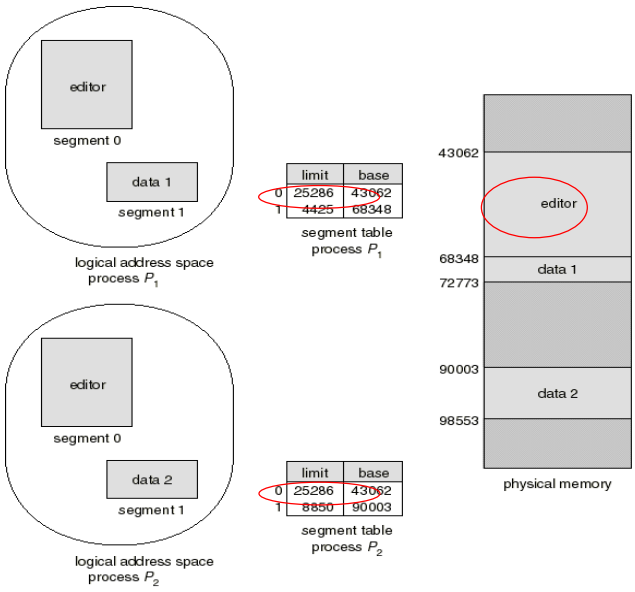
Simple Segmentation

- **共享段**
  - 分段系统中的共享
    - 分段的共享：两个不同进程的段表中的条目指向相同的物理地址
    - 示例
      - 文本编辑器的相同代码可以由许多用户共享。
      - 只有一个副本保存在主内存中。
    - 但是每个用户仍然需要有自己的私有数据段。

Simple Segmentation

- **共享段**
  - 实例

两个进程共享editor的代码段，同时拥有自己的数据段

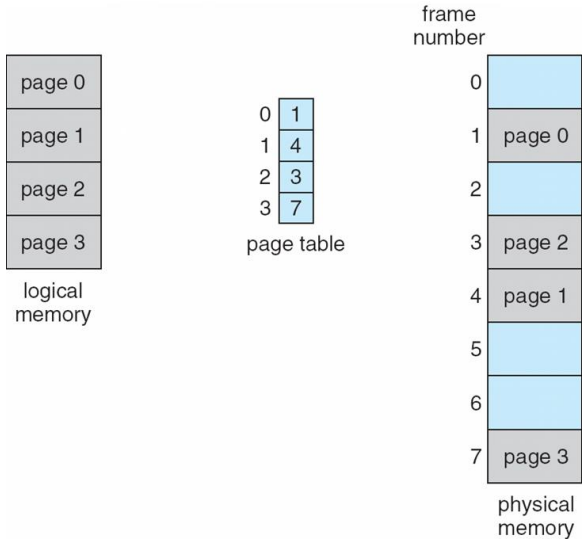


Simple Paging

- **简单的分页概念**
  - 想法：进程的物理地址空间可以是不连续的；只要物理内存大小可用，就为进程分配物理内存。
    - 避免外部碎片。
    - 避免了大小不一的内存块问题。
  - 将物理内存划分为称为帧(页帧，页框，物理页)的固定大小的块，大小为2的幂，通常介于512字节和16MB之间。
  - 将逻辑内存划分为与物理帧大小相同的块，称为页(页，页面，逻辑页)。
  - 因此，可以给进程页分配主存储器中的任何空闲帧；进程不需要占用物理内存的连续部分。要运行一个大小为n页的程序，我们需要找到物理内存的n个空闲帧并加载该程序。
    - 所以我们需要使用一个空闲帧列表来跟踪物理内存中的所有空闲帧。
    - 我们需要设置一个页表将逻辑页转换为物理帧。
  - 内部碎片仅可能存在于程序末尾的页。

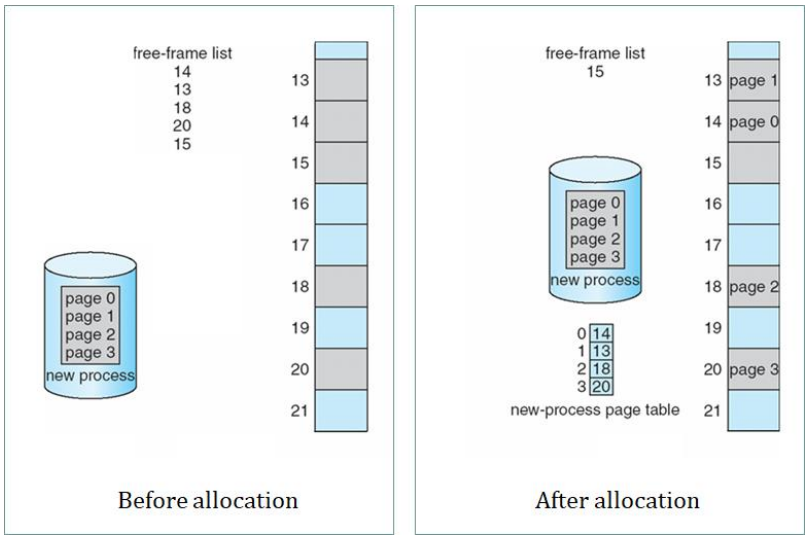
Simple Paging

- **简单的分页概念**
  - 分页示例



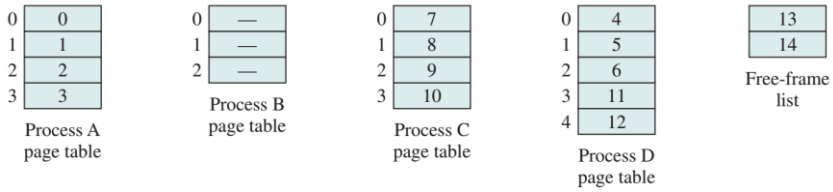
Simple Paging

- **简单的分页概念**
  - 空闲帧列表示例



Simple Paging

- **简单的分页概念**
  - 操作系统现在需要在主内存中为每个进程维护一个页表。
    - 页表的每个条目都包含对应页码的物理帧编号。
    - 页表按页码索引以获得帧编号。
  - 还需维护一个可用帧的空闲帧表/列表。
  - 实例



Simple Paging

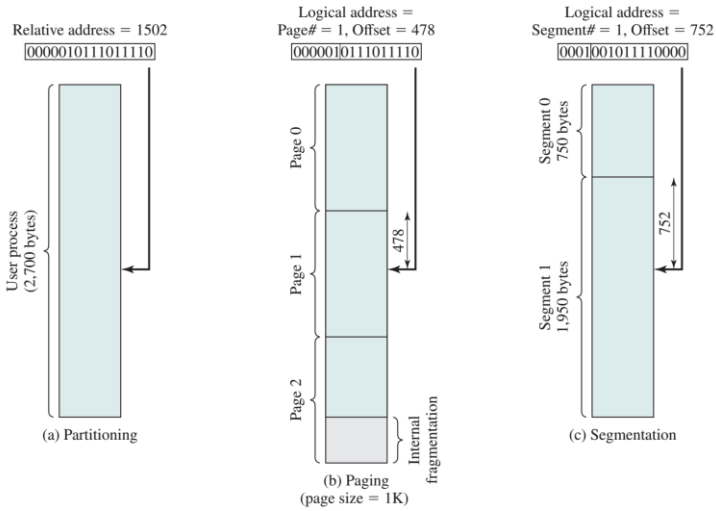
- **计算内部碎片**
  - 实例
    - 页/帧大小：2048字节
    - 假设某个进程的大小为：72766字节
      - 72766 = 2048 × 35 + 1,086.
      - 需要为进程分配36个帧。
      - 内部碎片：2048-1086=962（字节）
    - 最坏情况的内部碎片大小: 2048 - 1 (byte)
      - 对大小为2048p+1字节的进程。
    - 平均内部碎片：1/2帧大小。
  - 因此，帧大小越小越好吗？
    - 但每个页表条目都需要内存来跟踪。
    - 页面大小随时间增长：
      - Linux页面大小：4KB/2MB

```
$getconf PAGESIZE
$cat /proc/meminfo | grep Huge
```

      - Solaris页面大小：8KB/4MB。

Simple Paging

- **分页中的逻辑地址**
  - 当页面大小为2的幂时，逻辑地址变为相对地址。
    - 分区、分页和分段的比较



Simple Paging

- **分页中的逻辑地址**
  - 当页面大小为2的幂时，逻辑地址变为相对地址。
    - 实例
      - 若使用16位地址且页面大小=1K，则偏移量需要10位，页码需要6位。
      - 那么，相对于进程开始位置的16位逻辑地址以10个最低有效位作为偏移量，6个最高有效位作为页码。

Simple Paging

- **分页中的逻辑地址**
  - 在每个程序中，每个逻辑地址必须由页码和页内的偏移量/位移组成

    <page_number, offset> = <p, d>.
  - 专用寄存器始终保存当前运行进程的页表的起始物理地址。
  - 对逻辑地址

    <page_number, offset> = <p, d>,

    处理器访问页表以获取物理地址

    <frame_number, offset> = <f, d>.

Simple Paging

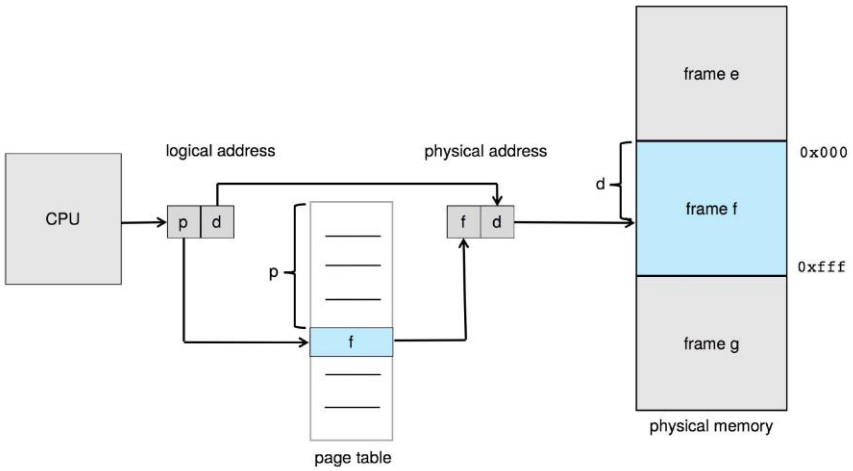
- **地址转换方案**
  - CPU生成的逻辑地址分为两部分：
    - 页码p – 用作页表的索引，页表包含物理内存中每个页的基址。
    - 页偏移/位移d – 与基址相加，定义发送到内存单元的物理内存地址。
  - 对于给定的逻辑地址空间2^m和页面大小2^n, n<m

页码	页偏移
p	d
*m-n*位	*n*位

  - 通过使用2次幂的页面大小，这些页面对程序员、编译器/汇编程序和链接器将不可见。
  - 运行时的地址转换很容易在硬件中实现：
    - 逻辑地址<p, d>转换为物理地址<f, d>，方法是用p索引页表，查得帧码f后，将相同的偏移量d追加到的帧f中。

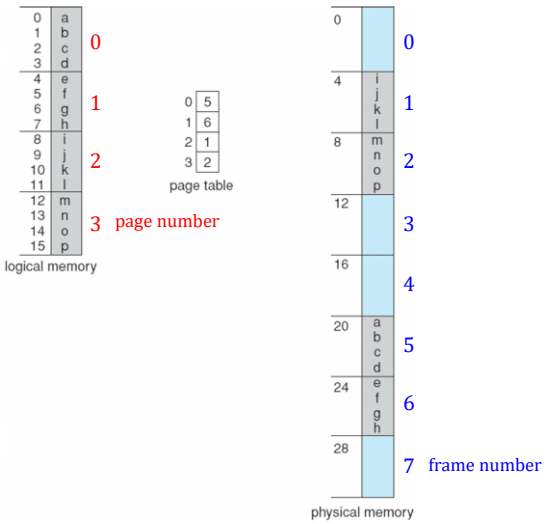
Simple Paging

- **地址转换体系结构**



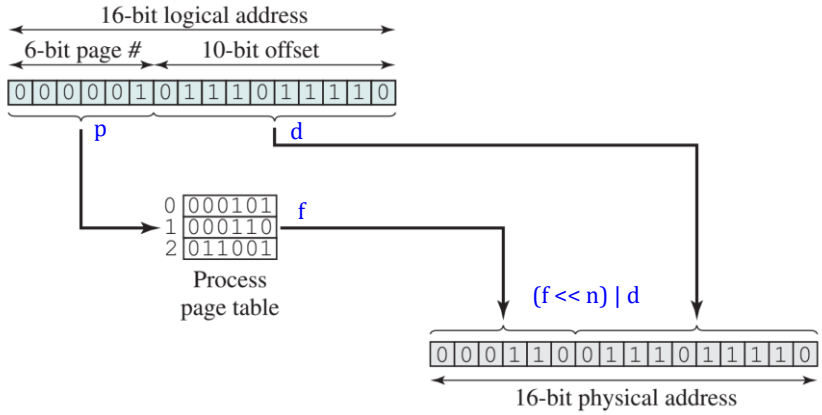
Simple Paging

- **地址转换体系结构**
  - 实例
    - *m=4，n=2，页面大小=2^2=4（字节），内存大小=32（字节）*



Simple Paging

- **地址转换体系结构**
  - 实例
    - *m=16，n=10，页面大小=2^{10}=1024（字节）*

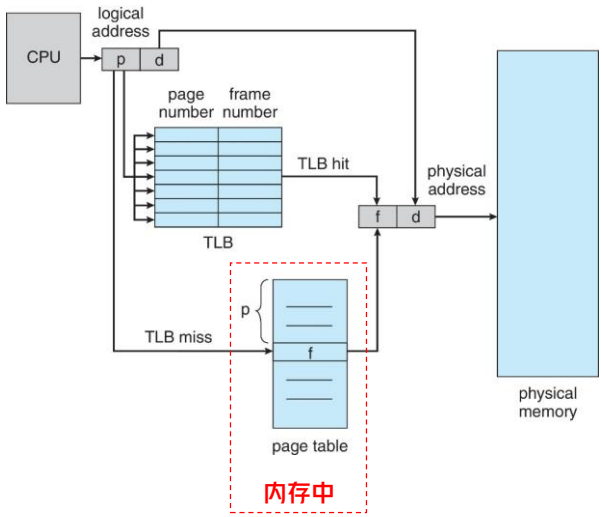


Page Tables

- **实现页表**
  - 如果我们将页表保存在主内存中
    - *页表基址寄存器*（*PTBR*）指向内存中页表的位置。
    - *页表长度寄存器*（*PTLR*）指示页表的大小。
    - 每次数据/指令访问都需要两次内存访问。
      - 一次用于页表，一次用于数据/指令
  - 如果我们在硬件中保存页表（在MMU中？）
    - 但是，页表可能很大—太贵了。
  - 将这两种机制结合起来可以解决两次内存访问问题。
    - 使用一种特殊的快速查找硬件缓存，称为关联内存（寄存器）或转换查找缓冲区（TLB, 快表）– 支持快速并行搜索。
    - 地址转换<p, d>
      - 若p在关联寄存器中，则获取帧编号。
      - 否则，从内存中的页表中获取帧编号。

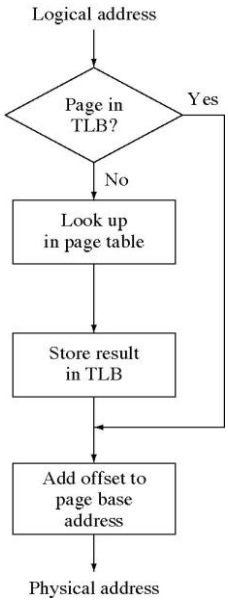
Page Tables

- **实现页表**
  - 具有TLB的分页硬件



Page Tables

- **实现页表**
  - 具有TLB的分页硬件
    - TLB流程图



Page Tables

- **实现页表**
  - 带TLB的分页硬件
    - TLB利用了局部性原理(Locality Principle).
    - TLB使用关联映射硬件同时查询所有TLB条目，以查找页码的匹配/命中。
    - TLB命中率应为90%+.
    - 每次新进程进入运行状态时，必须刷新（擦除）TLB.
      - 只有一个全局TLB服务于所有进程。
    - TLB信息可能在进程上下文中保存/加载。

Page Tables

- **实现页表**
  - 带TLB的分页硬件
    - 每次关联（内存）查找时间= ε 时间单位。
      - 时间单位：内存访问时间
      - 可以 < 10%（远快于）内存访问时间
    - 命中率=α
      - 在关联存储器中找到页码的次数百分比；
      - 与关联寄存器的数量相关。
    - 有效访问时间(EAT)

      $EAT = \alpha(\varepsilon + 1) + (1 - \alpha)(\varepsilon + 2) = 2 + \varepsilon - \alpha.$

      单位：内存访问时间。
    - EAT在 $1 + \varepsilon$ 到 $2 + \varepsilon$ 内存访问时间之间
      - 应该接近1.

Page Tables

- **实现页表**
  - 带TLB的分页硬件
    - 有效访问时间(EAT)

      $EAT = \alpha(\varepsilon + 1) + (1 - \alpha)(\varepsilon + 2) = 2 + \varepsilon - \alpha.$

      单位：内存访问时间。
    - 示例：假设内存访问周期时间为100ns，且ε = 20ns用于TLB搜索。
      - 考虑α = 80%. 那么

        EAT = 0.80 x (20 + 100) + 0.20 x (20 + 200) = 140(ns), or
        EAT = 2 x 100 + 20 – 1 x 100 x 0.8 = 140(ns)

        因此，内存访问时间减少了 (140–100)/100=40%
      - 考虑更现实的命中率α = 98%. 然后

        EAT = 0.98 x 120 + 0.02 x 220 = 122(ns), or
        EAT = 2 x 100 + 20 – 1 x 100 x 0.98 = 122(ns)

        因此，只有(122–100)/100=22%的内存访问时间延迟。

Page Tables

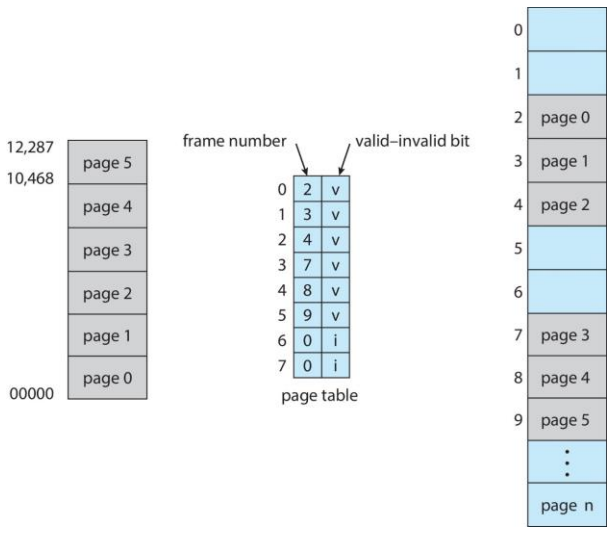
- **实现页表**
  - 带TLB的分页硬件
    - 高级TLB
      - 某些TLB在每个TLB条目中存储地址空间标识符(ASID) – 唯一标识每个进程，为该进程提供地址空间保护：
        - 否则，需要在每个上下文切换时刷新。
      - TLB通常较小（64到1024个条目）。
      - TLB未命中时，将缺失条目（页码与帧码）加载到TLB中，以便下次更快地访问：
        - TLB已满时，必须考虑条目替换策略。
        - 一些条目可以被固定（而不是从TLB中删除）以进行永久快速访问。

Page Tables

- **内存保护**
  - 内存保护通过将保护位与每个帧相关联来实现，以指示是否允许只读或读写访问。
    - 还可以添加更多位以指示仅执行页面，依此类推。
  - 页表中的每个条目都附加了有效位和无效位。
    - “valid”表示关联页位于进程的逻辑地址空间中，因此是合法页。
    - “invalid”表示页不在进程的逻辑地址空间中。
    - 或使用页表长度寄存器（PTLR）
  - 任何违规行为都会导致内核陷阱。

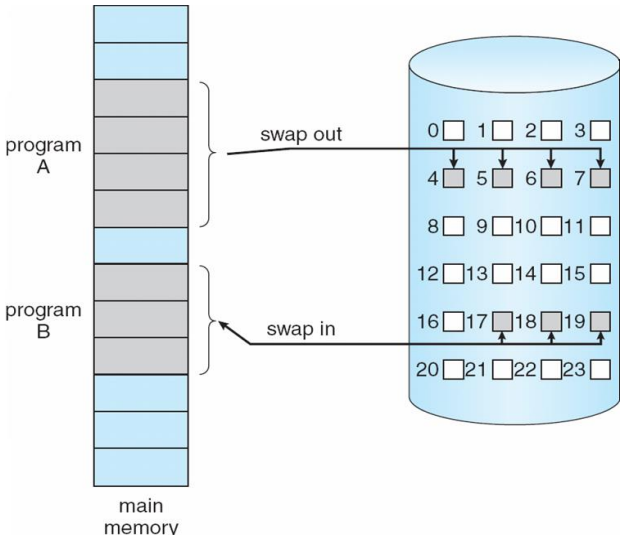
Page Tables

- **内存保护**
  - 页表中的有效（v）位或无效（i）位。



Page Tables

- 将分页内存传输到连续磁盘空间

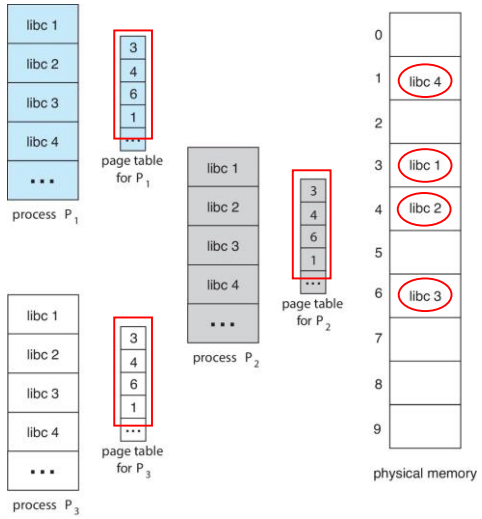


Page Tables

- 共享页面
  - 共享代码
    - 进程（如文本编辑器、编译器、窗口系统）之间共享的只读（可重入）代码的一个副本。
      - 类似于共享同一进程空间的多个线程。
    - 共享代码必须出现在相同物理地址上，且在所有进程的逻辑地址空间中。
  - 私有代码和数据
    - 每个进程保留代码和数据的单独副本。
    - 专用代码和数据的页面可以出现在逻辑地址空间的任何位置。

Page Tables

- **共享页面**
  - 共享页面示例

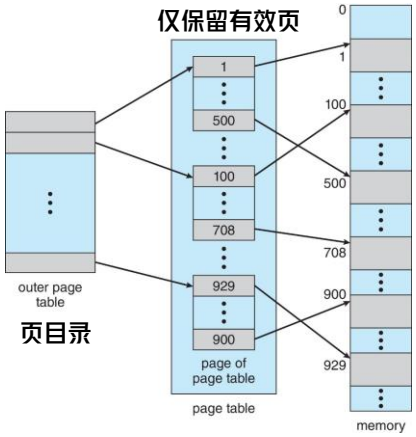


Page Tables

- **页表的结构**
  - 使用直接（数组）的方法，分页的内存结构可能会变得非常庞大
    - 考虑在现代计算机中一个32位逻辑地址空间，页面大小为4KB(2^{12}字节)
    - 页表将有一百万个条目($2^{32-12}=2^{20}$).
    - 如果每个条目4字节，那么仅页表就需要4MB的物理地址空间。
      - 使用的内存量非常昂贵。
      - 很难在主内存中连续分配该内存。
  - 页表有三种结构
    - 分层页表
    - 散列（哈希）页表
    - 倒置页表

Page Tables

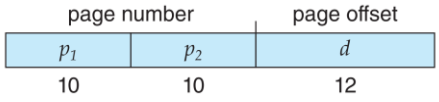
- **页表的结构**
  - 分层页表
    - 将逻辑地址空间拆分为多个页表（类似树）
      - 一种简单的技术是两级页表：把页表再分页



一种两级页表方案

Page Tables

- **页表的结构**
  - 分层页表
    - 考虑32位逻辑地址空间。
      - 逻辑地址（在页大小为2^{12}=4K的32位机器上）分为：
        - 由20位组成的页码
        - 由12位组成的页偏移
      - 由于页表是分页的，因此页码进一步分为：
        - 10位外部页码（相当于目录，用于找内部页）
        - 10位内部页偏移（定位页条目）
      - 因此，逻辑地址如下所示：

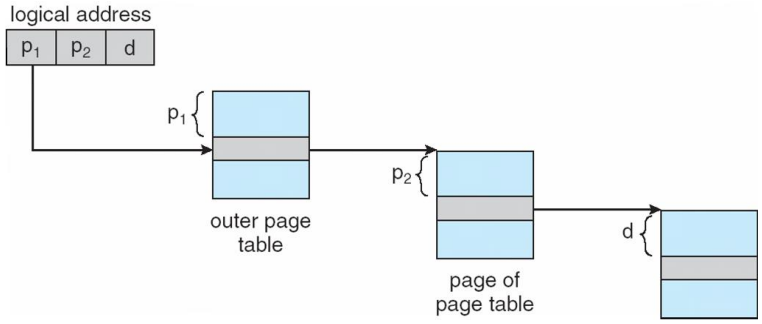


其中p_1是外部页表的索引，p_2是内部页的偏移。

      - 称为前向映射页表

Page Tables

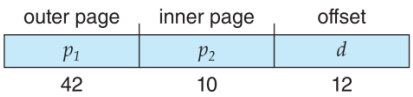
- **页表的结构**
  - 分层页表



两级分页方案

Page Tables

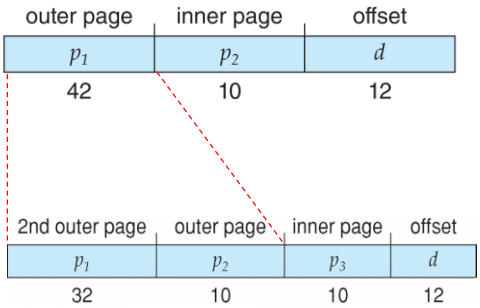
- **页表的结构**
  - 分层页表
    - 考虑一个64位逻辑地址空间。
      - 即使是两级分页方案也不够。
      - 若页面大小为4KB(=2^{12})，则页表有2^{52}个条目。
        - 对于两级方案，内部页表可以有2^{10}个长度为4字节的条目。地址看起来像



        - 外部页表有2^{42}个条目（每条目4字节，共2^{44}个字节）
      - 一种解决方案是将外部页表进一步细分，添加第二外部页表。

Page Tables

- **页表的结构**
  - 分层页表
    - 三级分页方案示例：第二外部页表的大小仍然有2^{34}字节(16GB)
      - 可能需要4次内存访问才能访问一个物理内存地址。

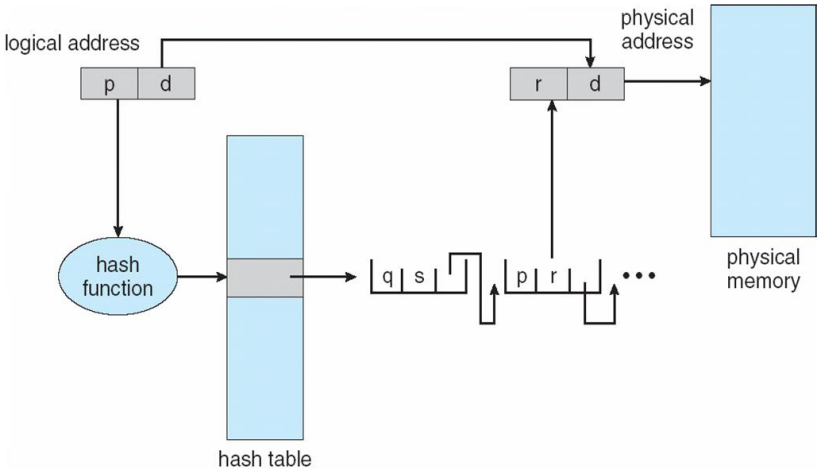


Page Tables

- **页表的结构**
  - 散列/哈希页表(hashed page table)
    - 常用于地址空间>32位
    - 虚拟页码散列到页表中。
      - 此页表包含散列到同一位置的元素链表。
    - 链表的每个元素都包含：
      - 虚拟页码
      - 映射的页帧的值
      - 指向下一个元素的指针。
    - 通过比较链表中的虚拟页码搜索匹配项
      - 如果找到匹配，则提取相应的物理帧。
    - 64位地址的变体是聚簇页表(clustered page table)
      - 与哈希相似，但每个条目都引用多个页面（如16页），而不是1页。
      - 特别适用于稀疏地址空间（内存引用不连续且分散）。

Page Tables

- **页表的结构**
  - 散列页表

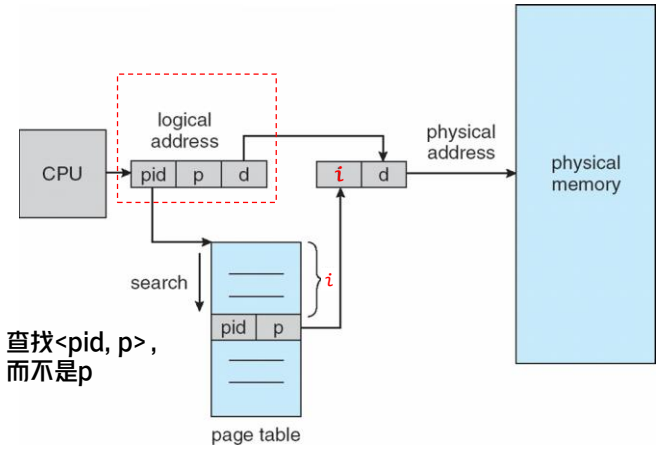


Page Tables

- **页表的结构**
  - 倒置/反转页表
    - 与其每个进程都有一个页表并跟踪所有可能的逻辑页，不如跟踪所有物理页。
    - 倒置页表的条目由存储在该真实（物理）内存位置的页的虚拟地址以及有关拥有该页的进程的信息组成。
      - 以物理页顺序
      - 内存的每一实页有一个条目
        - 例如，<process-id, page-number>
      - 用于64位Ultra SPARC、PowerPC…
    - 此方案减少了存储每个页表所需的内存，但增加了发生页引用时搜索表所需的时间。
      - 进程id、虚拟页码乱序
    - 使用哈希表将搜索限制为一页或最多几页的表项。
      - TLB可以加速访问（非命中才搜索哈希表）
    - 但是如何实现共享内存呢？
      - 虚拟地址到共享物理地址的一种映射
      - 物理帧只与一个逻辑页映射
      - 未映射的逻辑页导致页错误

Page Tables

- **页表的结构**
  - 倒置页表



倒置页表结构

Page Tables

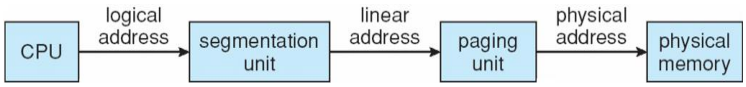
- **简单分段/分页比较**
  - 分段对程序员是可见的，而分页是透明的。
  - 自然支持保护/共享。
  - 分段可视为提供给程序员的必需品，它从逻辑上将程序组织为段以实现不同类型的保护（例如：仅对代码执行，但对数据读写）
  - 段的大小可变；页的大小是固定的。
  - 分段需要比分页更复杂的地址转换硬件。
  - 分段受到外部碎片的影响；分页只会产生小的内部碎片。
  - 也许可以结合分段和分页？

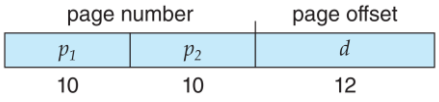
Operating System Examples

- **英特尔32位体系结构**
  - 主导产业芯片
  - 奔腾(Pentium)CPU是32位的，称为IA-32体系结构
    - 当前的英特尔CPU是64位的，称为IA-64体系结构
  - 芯片中的许多变体涵盖了这里的主要思想
  - 支持分段和段页式内存管理
    - 每个段可以是4GB
    - 每个进程最多16K个段
    - 分成两个分区
      - 第1个分区最多8K个私有段（保存在本地描述符表LDT中）
      - 第2个分区最多8K个段在所有进程之间共享（保存在全局描述符表GDT中）
  - CPU生成逻辑地址
    - 提供给分段单元以产生线性地址
    - 把线性地址给分页单元：
      - 它在主存中生成物理地址
      - 页大小可以是4KB或4MB
    - 分段单元和分页单元相当于MMU

Operating System Examples

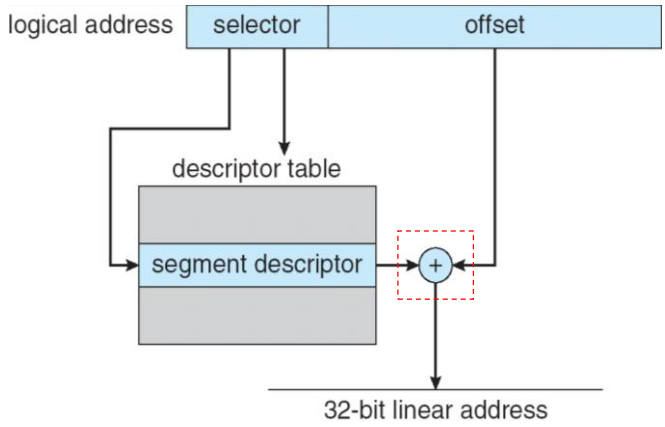
- **英特尔32位体系结构**
  - 逻辑到物理地址转换





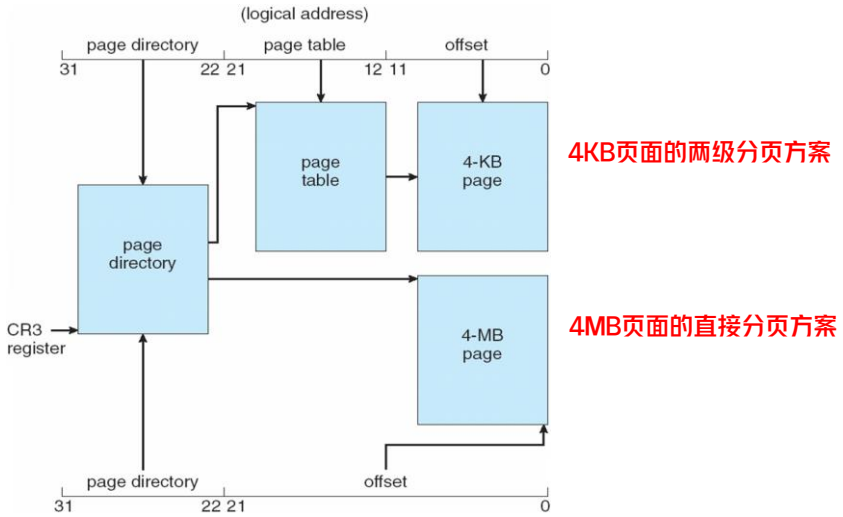
Operating System Examples

- **英特尔32位体系结构**
  - 英特尔奔腾的分段



Operating System Examples

- **英特尔32位体系结构**
  - 英特尔奔腾分页体系结构

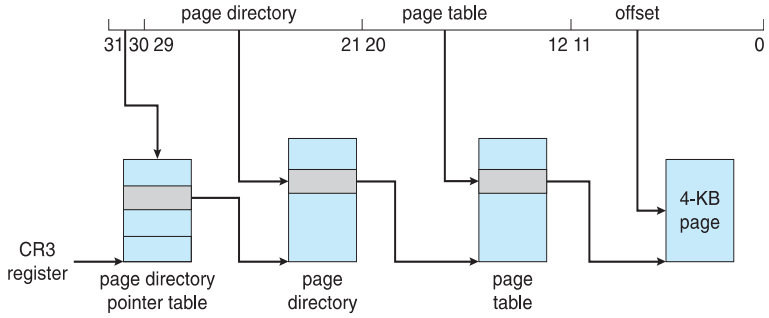


4KB页面的两级分页方案

4MB页面的直接分页方案

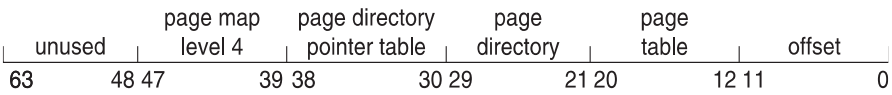
Operating System Examples

- **英特尔32位体系结构**
  - 英特尔IA-32页地址扩展
    - 32位地址限制促使Intel创建页面地址扩展(PAE)，允许32位应用程序访问超过4GB的内存空间。
      - 分页采用了三级方案
      - 前两位指向页目录指针表
      - 页目录和页表条目的大小从32位增加到64位
      - 效果是将地址空间增加到36位(64GB)的物理内存



Operating System Examples

- **英特尔32位体系结构**
  - 英特尔X86-64
    - 当前一代英特尔x86体系结构
    - 64位为ginormous (>16 EB, exabyte)
    - 实际上，仅实现48位寻址
      - 页面大小为4KB/2MB/1GB
      - 四级分页层次结构
    - 还可以使用PAE，使虚拟地址为48位，物理地址为52位

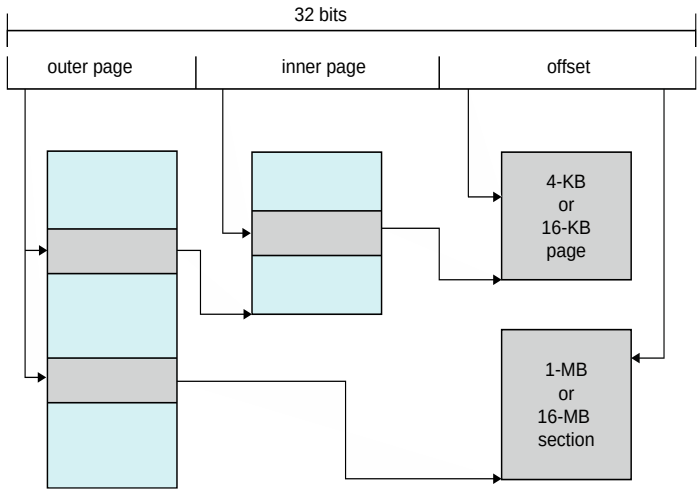


Operating System Examples

- **ARM架构**
  - 主导移动平台芯片（例如苹果iOS和谷歌安卓设备）
  - 现代、节能、32位CPU
  - 支持4KB/16KB页面
  - 支持1MB/16MB页面（称为段/节section）
  - 一级分页用于分段，两级分页用于较小的页面
  - 支持两级TLB
    - 外部有两个微TLB（一个用于数据，一个用于指令）
    - 内部有一个主TLB
    - 首先检查外部，未命中时再检查内部，仍未未命中则由CPU执行页表查找

Operating System Examples

- **ARM架构**



Operating System Examples

- **Linux**
  - Linux中的三级分页