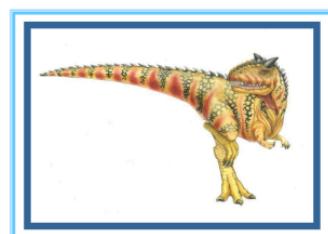


第 9 章：主内存



操作系统概念 - 第 10 版



第 9 章：内存管理策略

- Background Contiguous Memory Allocation 分段
-
-
- 页表的分页结构示例：Intel 32 位和 64 位体系结构以及 ARM 体系结构
-
-

操作系统概念 - 第 10 版

9.2





目标

- 解释逻辑地址和物理地址之间的区别，以及内存管理单元（MMU）在地址转换中的作用。
 - 动态存储分配问题 - 应用第一个、最佳和最差拟合策略来连续分配内存。
 - 解释内部和外部碎片之间的区别。在包含转换后备缓冲区（TLB）的分页系统中将逻辑地址转换为物理地址。
 -
 - 描述分层分页和散列分页描述 IA-32、x86-64 和 ARMv8 架构的地址转换
 -

操作系统概念 - 第 10 版

9.3



七

操作系统概念 - 第 10 版

94





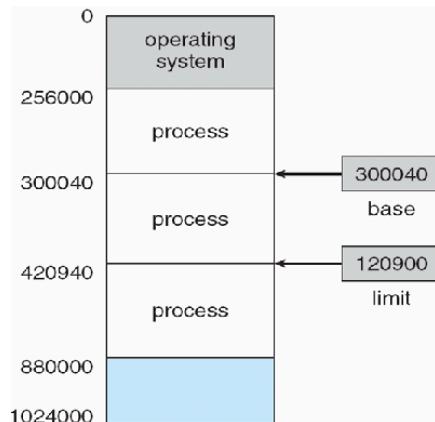
背景 (续)

- 主内存 (包括缓存) 和寄存器是 CPU 唯一可以直接访问的存储;换句话说, CPU 不能直接访问二级存储 (如磁盘), 而是通过 I/O 控制器
- 内存管理单元或 MMU 只能看到地址 + 读取请求流, 或地址 + 数据和写入请求
 - 可以在一个 CPU 时钟 (周期) 中访问寄存器 访问主内存 (通过内存总线) 可能需要许多 CPU 时钟, 当它还没有完成它正在执行的指令所需的数据时, 会导致内存停顿 缓存位于主内存和 CPU 寄存器之间, 驻留在 CPU 芯片上以实现快速访问
 -
- 需要内存保护以确保正确操作
 - 我们必须保护操作系统内存空间不被用户进程访问, 并保护用户进程彼此之间
 - 这种保护必须由硬件提供, 以实现性能 - 性能或速度



彼此分离的每进程内存空间

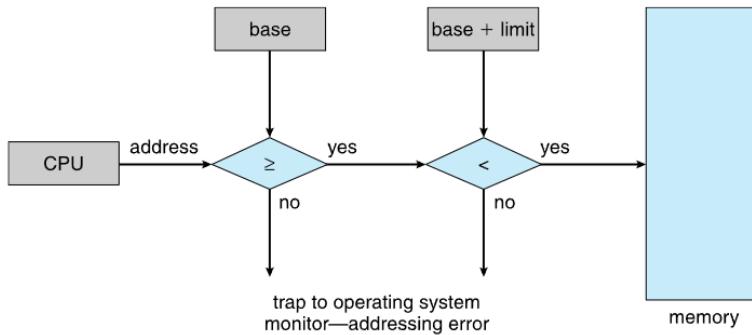
- 单独的每个进程内存空间保护进程彼此独立, 这是在内存中加载多个进程以进行并发执行的基础 - 多编程系统
- 一对基寄存器和限制寄存器定义进程地址空间的合法范围
 - 例如, 如果基寄存器持有 300040 而限制寄存器为 120900, 则程序可以合法访问从 300040 到 420939 (含) 的所有地址





硬件地址保护

- 通过让 CPU 将用户模式下生成的每个地址与这两个 registers 进行比较，可以保护内存空间
- base 和 limit 寄存器只能由操作系统加载，操作系统使用特殊的特权指令（仅在内核模式下执行）
- 但是，在内核模式下执行的操作系统可以不受限制地访问操作系统内存和用户内存



操作系统概念 - 第 10 版

9.7



地址绑定

- 通常，程序作为二进制可执行文件驻留在磁盘上，必须将其放入内存并放置在进程（其地址空间的一部分）中才能运行
- 大多数系统都允许用户进程驻留在物理内存的任何部分。尽管计算机可以从 00000 开始，但用户进程的第一个地址不需要是 00000 用户程序会经历几个步骤 - 其中一些步骤在执行之前可能是可选的。在这些步骤中，地址可能以不同的方式表示：
 - 源代码地址通常是符号的 - 变量 count 编译器通常将这些符号地址绑定到可重定位的地址，例如“从此模块的开头开始 14 字节”
 - 链接器或加载器反过来将可重定位地址绑定到绝对地址，即 74014
 - 每个绑定本质上都是从一个地址空间到另一个地址空间的映射



操作系统概念 - 第 10 版

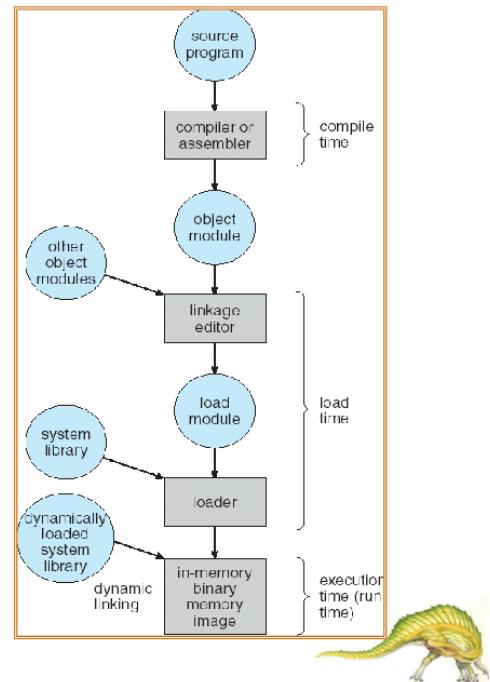
9.8



用户程序的多步骤处理

指令和数据到内存地址的地址绑定可以分三个不同的阶段进行

- 编译时：如果内存位置先验已知，则可以生成绝对代码；如果起始位置发生变化（例如，“gcc”），则必须重新编译代码。MS-DOS 使用此
- 链接或加载时：如果在编译时不知道内存位置（例如，Unix “ld”执行链接），则编译器必须生成可重定位代码。绑定会延迟到加载时间。如果起始地址发生变化，我们只需要重新加载用户代码以合并这个更改后的值
- 执行时间：如果进程在执行期间可以从一个内存段移动到另一个内存段，则绑定延迟到运行时。这需要硬件和操作系统支持地址映射（例如，基寄存器和限制寄存器），例如动态库。大多数通用操作系统都使用此方法。



地址转换和保护

- 在以前的 uni-programming（例如 MS-DOS）中，当时只有一个程序可以运行，因此它“几乎”占据了整个物理内存
 - 不需要地址转换，也没有任何保护
- 在 Windows 3.1 (1990-1992) 等多程序编程的早期阶段
 - 没有地址转换，绑定发生在链接或加载器时 – 当程序加载到内存中时调整地址。程序中的错误可能会使系统崩溃
 - 后来使用 base 和 limit 寄存器添加了保护，防止其他用户程序非法访问





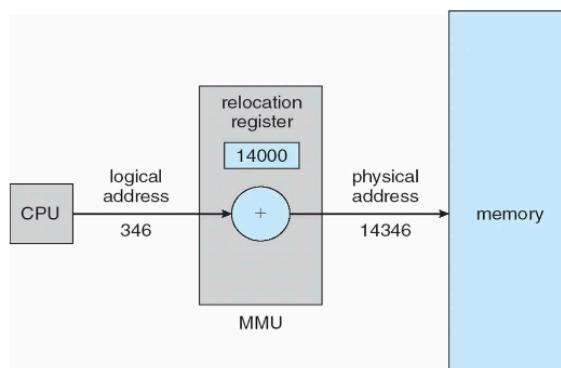
逻辑地址空间与物理地址空间

- 召回：地址空间
 - 进程可以“接触”的所有地址空间 每个进程都有自己唯一的地址空间，与内核地址空间不同
 -
- 因此，有两种内存视图
 - 逻辑地址 – 由 CPU 生成;也称为虚拟地址 物理地址 – 内存看到的地址
 -
 - 转换由内存管理单元或 MMU – 硬件设备完成
- 翻译使实施保护变得更加容易
 - 确保一个进程无法访问另一个进程的地址空间。
- 如果使用编译时和加载时地址绑定方案（在过去），则逻辑地址和物理地址是相同的;逻辑（虚拟）地址和物理地址仅在执行时地址绑定方案中有所不同
 - 逻辑地址空间是程序生成的所有逻辑地址的集合 物理地址空间是程序生成的所有物理地址的集合
 -



内存管理单元 (MMU)

- MMU 是一种硬件机制，在运行时将虚拟地址映射到物理地址有许多不同的映射方法，本章的其余部分将介绍这些方法
-
- 首先，考虑一个简单的方案，其中重定位寄存器中的值被添加到用户进程在发送到内存时生成的每个地址
 - 用户程序从不访问真实的物理地址
- 用户程序处理逻辑地址;它永远不会看到真正的物理地址
 - 当引用内存中的实际位置时，将发生执行时绑定





连续分配

- 连续分配是早期的内存分配方法之一 主内存通常分为两个分区 – 一个用于操作系统，一个用于用户进程
-
- 操作系统可以放置在低内存地址或高内存地址中，具体取决于许多因素，例如中断向量的位置许多操作系统（包括 Linux 和 Windows）将操作系统置于高内存
-
- 在多编程操作系统中，多个用户进程同时驻留在内存中。在连续内存分配中，每个进程都包含在单个内存部分中，该内存部分与包含下一个进程的部分相邻。

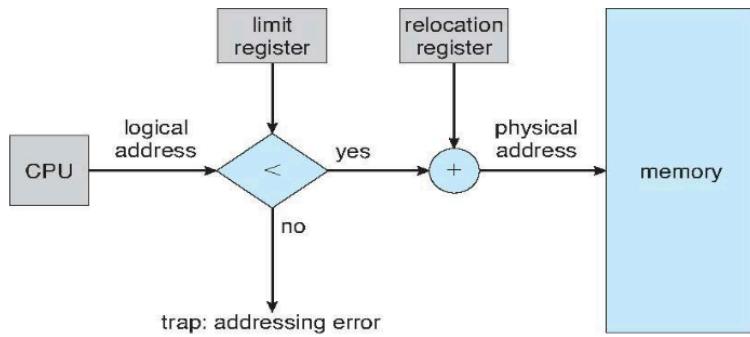
操作系统概念 - 第 10 版

9.13



对 Relocation 和 Limit 寄存器的硬件支持

- 重定位寄存器和限制寄存器用于保护用户进程彼此之间以及更改操作系统代码和数据
 - 重定位寄存器包含最小物理地址的值限制寄存器包含逻辑地址范围 – 每个逻辑地址必须小于限制寄存器
 -
 - MMU 动态映射逻辑地址



操作系统概念 - 第 10 版

9.14

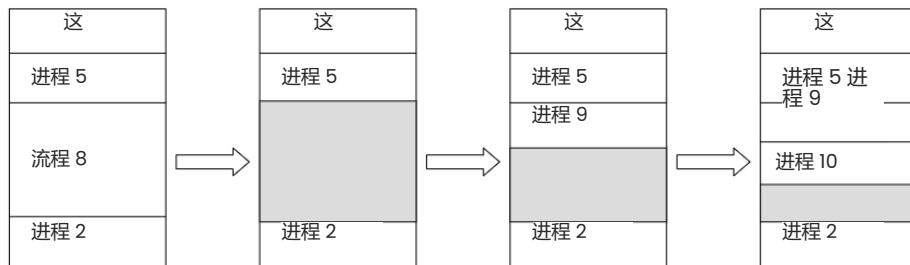




连续分配 (续)

□ 多分区分配

- 多程序编程的程度受分区数量的限制 可变分区大小 (根据给定进程的需要调整大小) Hole - 可用内存块: 各种大小的孔分散在整个内存中 当一个进程到达时, 它会从一个足够大的孔中分配内存以容纳它 进程退出释放其分区, 相邻的空闲分区组合在一起
-
-
-
- 操作系统维护有关 a) 已分配分区的信息; b) 空闲分区 (HOLE)



操作系统概念 - 第 10 版

9.15



动态存储分配问题

如何满足自由孔列表中 n 号的 (可变) 请求?

- First-fit: 分配足够大的第一个孔
- 最佳拟合: 分配足够大的最小孔
 - 必须搜索整个列表, 除非按大小排序 产生最小的剩余孔 - 打算不使用它
 -
- 最差拟合: 分配最大的孔
 - 还必须搜索整个列表 生成最大的剩余孔 - 旨在重复使用剩余的孔
 -

实验表明, 在减少时间和存储利用率方面, 首次拟合和最佳拟合都优于最差拟合。就存储利用率而言, 首次拟合和最佳拟合都明显优于另一个, 但首次拟合通常更快。



操作系统概念 - 第 10 版

9.16



破碎

- 外部碎片 – 可用于满足请求的总内存空间，但它不是连续的 – 分散的漏洞
 - 存储碎片化成大量小孔。
- Internal Fragmentation – 分配给进程的内存可能大于请求的内存；此大小差异是分区内部的内存，但未被使用



碎片 (续)

- 通过一种称为压缩的技术减少外部碎片
 - 随机排列内存内容，将所有空闲内存放在一个大块中
 - 仅当重新定位是动态的并且在执行时完成时，才能进行压缩。换句话说，如果重新定位是静态的，并且在装配或加载时完成，则无法进行压缩
 - 压缩成本高昂（耗时）
- 后备存储（二级存储）具有类似的碎片问题，稍后将讨论
- 另一种更可行的解决方案是允许进程的逻辑地址空间是非连续的，方法是将其划分为多个部分（可变大小的段或固定大小的页面），从而允许在有此类内存的地方为进程分配物理内存。这些技术包括分段和分页



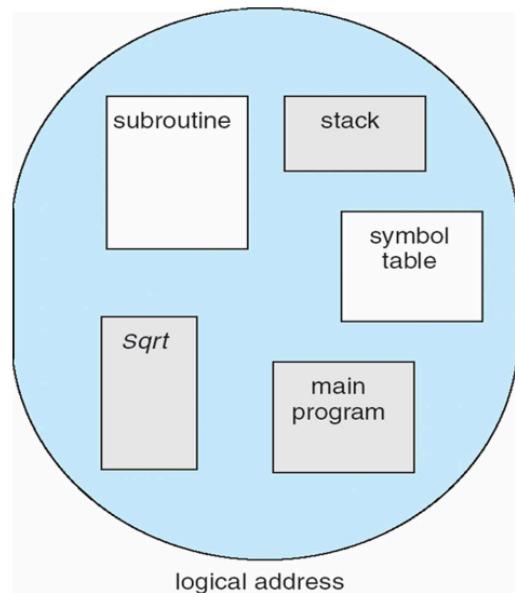


分段 - 用户视图

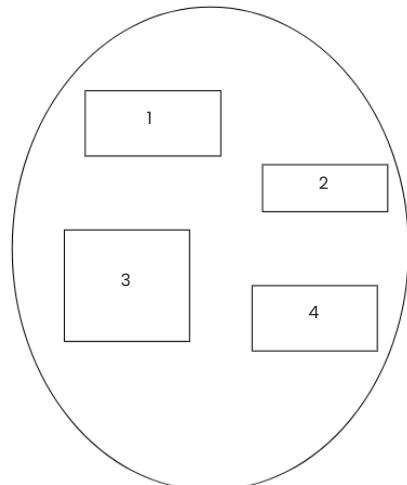
- 支持用户内存视图的内存管理方案
- 项目是区段的集合。段是一个逻辑单元，例如：

主程序 过程 函数 方法 对象 局部变量，全局变量 公共块

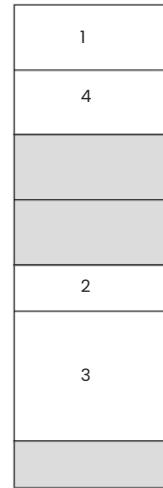
堆栈元素表数组



分段：逻辑 vs. 物理



用户空间



物理内存空间





分段架构

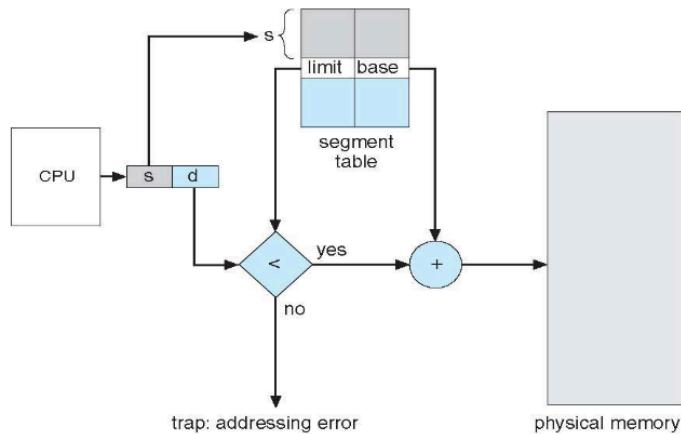
- Logical 地址现在由两个元组组成：
<段编号、偏移量>。
- Segment table – 将二维程序员定义的地址（虚拟地址）映射到一维物理地址;Segmentation 表中的每个条目都有：
 - base – 包含段驻留在内存中的起始物理地址 limit – 指定段的长度
 -
- STBR 和 STLR 都存储在进程的 PCB 中 – 召回进程 PCB 包含有关进程的所有信息
- Segment-table base register (STBR) 指向 segment table 在内存中的位置 Segment-table length register (STLR) 表示程序使用的 segment 数;
-

如果 $s < STLR$, 则段号 s 是合法的



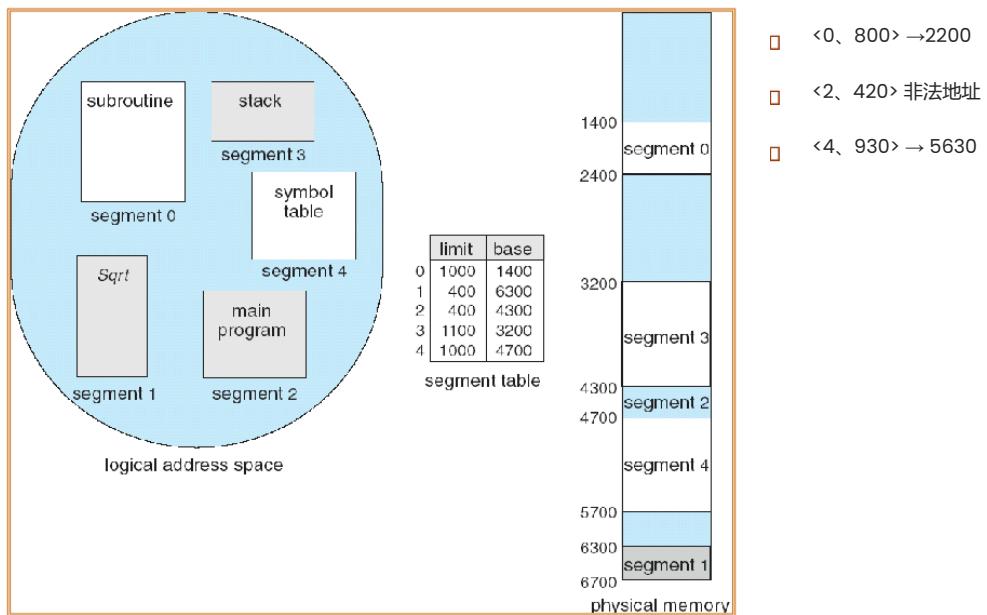
分段架构 (续)

- 保护。区段表中的每个条目都与以下项相关联：
 - 验证位 = 0 ⇔ 非法的段读/写/执行权限
 -
- 每个 segment 都关联了一个保护位;代码共享发生在区段级别
- 由于 Segment 的长度不同, 因此 Segment 的内存分配也是一个动态存储分配问题
- 外部碎片仍然存在, 但比连续分配中的要轻得多 – 因为每个 segment 都比进程的总内存空间小得多





分段示例

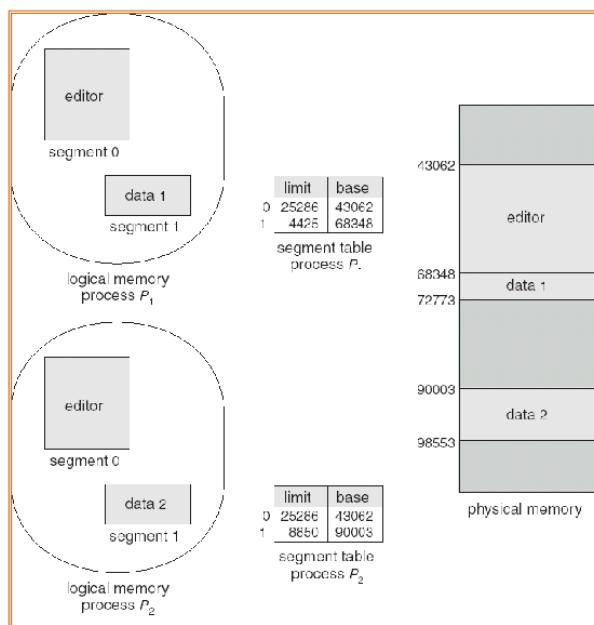


操作系统概念 - 第 10 版

9.23



区段共享



操作系统概念 - 第 10 版

9.24





摘要 - 分段

- 在分段方案中易于保护 – 在分段表中
 - 代码段将是只读的，数据和堆栈将是读写的（允许存储），共享段可以是只读或读写的
- 地址可以超出有效范围吗？
 - 是的，这就是允许 stack 和 heap 增长的方式。例如，堆栈出错，系统会自动增加堆栈的大小
- 分割的主要问题
 - 它必须将可变大小的块放入物理内存中 – 经典的动态存储分配问题
 - 它可能会多次移动进程以适应所有内容外部碎片
 -



寻呼

- 进程的物理地址空间可以是非连续的；每当物理内存空间可用时，都会为进程分配物理内存
避免外部碎片 避免大小不同的内存块的问题 将物理内存划分为固定大小的块，称为帧
- - 通常，大小是 2 的幂，介于 4KB（物理内存地址中的 12 位偏移量）和 1GB（30 位偏移量）之间——由硬件定义
- 将逻辑内存划分为相同大小的块，称为页面 – 帧的大小相同 操作系统需要跟踪主内存中所有可用的空闲（物理）帧 要运行大小为 N 页的程序，需要在内存中找到 N 个空闲帧（非连续）并将程序加载到其中
-
-
- - 页表用于将逻辑地址转换为物理地址 – 跟踪分配的帧
- 但是，这在最后一页/帧中受到内部碎片的影响





地址转换方案

- CPU 生成的地址分为：
 - 页码 (p) – 用作页表的索引，该页表包含物理内存中每页的基址
 - 页面偏移量 (d) – 与基址相结合，以定义发送到内存单元的物理内存地址



- 总 m bits 逻辑地址，逻辑地址空间 2^m 字节，页面大小或帧大小为 2^n ，此逻辑地址包含的页数为 2^{m-n}



分页方案示例

- Suppose logical address of a system is



- The page size - 12 bits are used to offset into a page, the page size is $2^{12} = 4$ KB
- 32 bits virtual address indicates virtual address space is $2^{32} = 4$ GB
- 20 bits page number, the number of page is 2^{20}



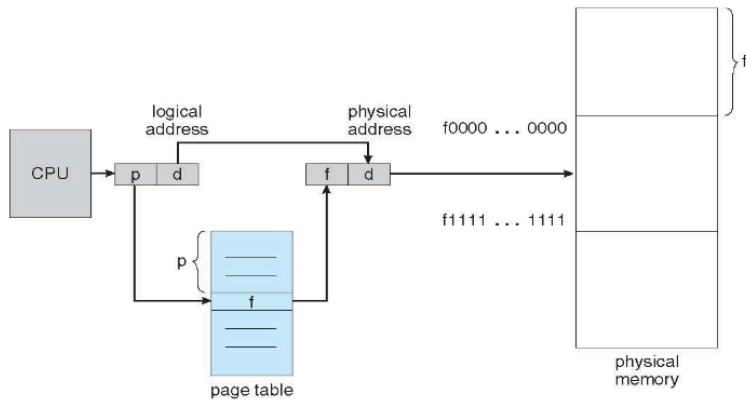


分页硬件

□ 下面概述了 MMU 将 CPU 生成的逻辑地址转换为物理地址的步骤：

1. 提取页码 p 并将其用作页表中的索引。
2. 从页表中提取相应的帧编号 f。
3. 将逻辑地址中的页码 p 替换为帧号 f 由于偏移量 d 不变，因此不会被替换，帧号和偏移量现在构成物理地址。

□

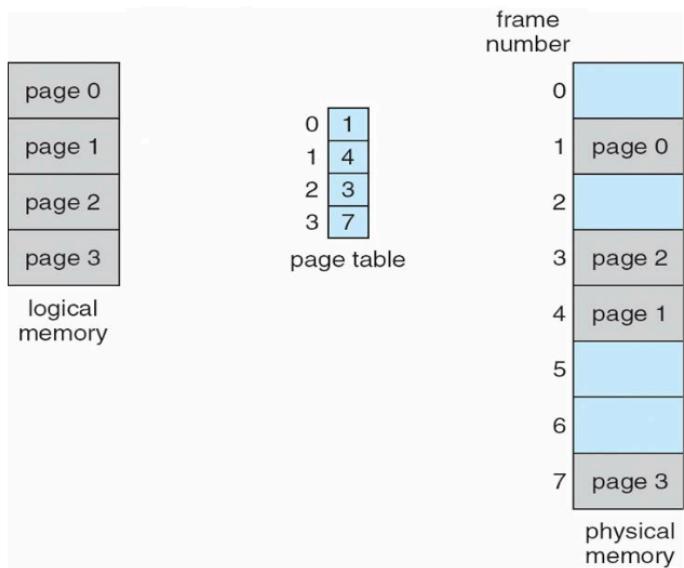


操作系统概念 - 第 10 版

9.29



逻辑内存和物理内存的分页模型



操作系统概念 - 第 10 版

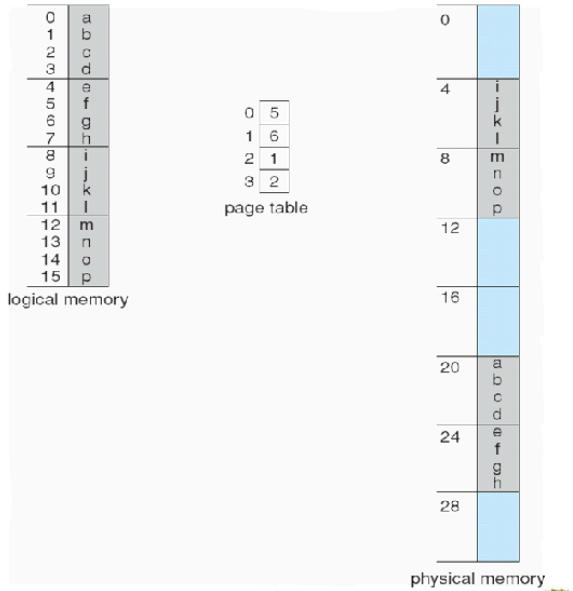
9.30



分页示例

- 帧号可以自动推导出每帧的起始地址
- $m=4$ 表示 16 字节的逻辑地址。逻辑地址 0 是第 0 页，偏移量为 0。索引到页表中，我们发现页面 0 位于第 5 帧中。因此，逻辑地址 0 映射到物理地址 $20 [= (5 \times 4) + 0]$ 。逻辑地址 3 (页面 0，偏移量 3) 映射到物理地址 $23 [= (5 \times 4) + 3]$ 。逻辑地址 4 是第 1 页，偏移量为 0。根据页表，页 1 映射到帧 6。因此，逻辑地址 4 映射到物理地址 $24 [= (6 \times 4) + 0]$ 。逻辑地址 13 映射到物理地址 9。
-
-
-

$n=2$ 和 $m=4$ 32 字节内存和 4 字节页面



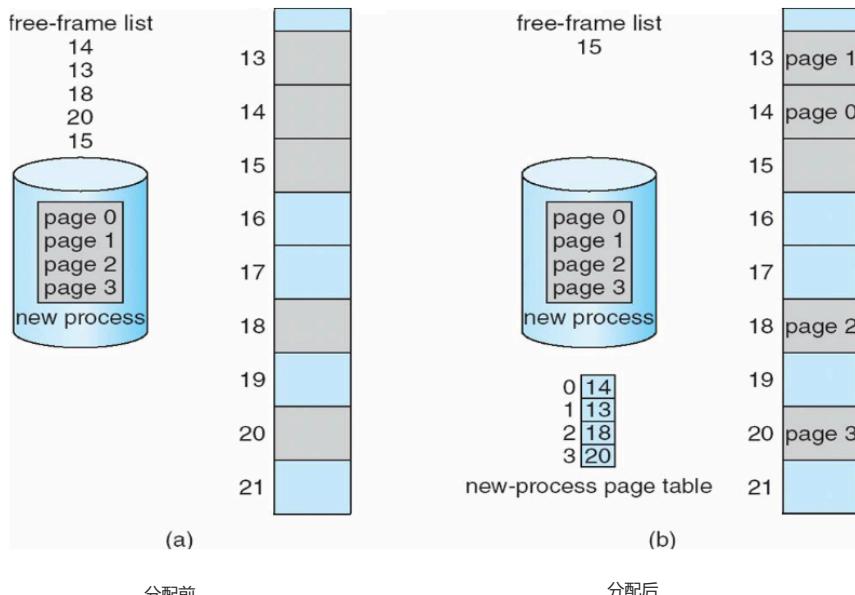
分页 (续)

- 计算内部分段。页面大小 = 2,048 字节 (2KB)。进程大小 = 72,766 字节。35 页 + 1,086 字节 内部分段。 $2,048 - 1,086 = 962$ 字节。最坏情况下的分段 = 1 帧 - 1 字节。平均分段 = 1 / 2 帧大小
- 那么小框架尺寸是可取的吗？-没有
 - 当页面大小增加时，每个页表条目所涉及的开销会减少。每个页表条目都需要内存来跟踪，较小的页面大小会导致页表条目的数量增加（每页一个），从而导致更大的页表
- 页面大小随着时间的推移而增长
 - SunMicro Solaris OS 支持两页大小 - 8 KB 和 4 MB
- 进程视图和物理内存现在大不相同
 - 程序员将内存视为一个单独的空间，仅包含这个程序。但实际上，用户程序分散在物理内存中，该内存还保存其他程序。





自由帧



页表的实现

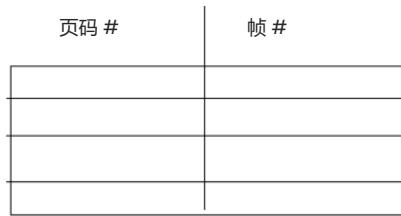
- 页表保存在主存储器中 – PCB 必须跟踪进程页表基寄存器 (PTBR) 指向页表 (起始地址) 的内存分配
- 页表长度寄存器 (PTLR) 表示页表的大小
- 在此方案中，每个数据/指令访问都需要两次内存访问
 - 一个用于页表 (翻译)，一个用于获取数据/指令 – 与分割方案相同
- 双内存访问问题可以通过使用称为关联内存或转换后备缓冲区 (TLB) 的特殊快速查找硬件缓存来解决
- TLB 通常较小（64 到 1,024 个条目）。某些 CPU 实现单独的指令和数据地址 TLB。它由系统中的所有进程（内核和用户进程）共享和使用
- 如果 TLB 未命中（如果页码不在 TLB 中），则值将加载到 TLB 中，以便下次更快地访问。这可以通过硬件 (MMU) 或软件 (运行内核代码) 来完成
 - 必须考虑替换策略 – LRU、循环甚至随机一些条目可以连接以实现永久快速访问，例如用于关键内核代码的 TLB 条目以实现快速访问
 -





Associative Memory

- 关联存储器 (TLB) – 并行搜索

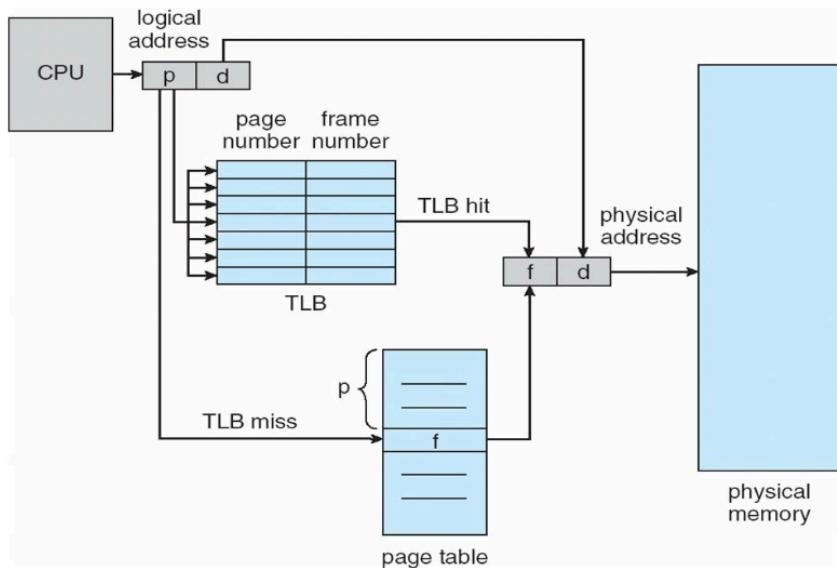


- 地址转换 (p, d)

- 并行检查所有条目 (硬件) – 所有进程共享和使用的 TLB (进程 ID 和页码)
 - 如果 p 在关联寄存器 (TLB) 中, 则取出帧 # – TLB 命中否则从内存中的页表中获取帧 #, 并将此条目带入 TLB
 -
- TLB 上的局部性
 - 指令通常保持在同一页面上 (顺序访问性质) 堆栈表现出局部性 (弹出和弹出)
 -
 - 数据较少的地方性, 仍然相当多



带 TLB 的分页硬件





有效访问时间

- Associative Lookup = ⊕ 时间单位
 - 通常< 10% 的内存访问时间
- 命中率 = ⊕ 命中率 – 在关联寄存器中找到页码的次数百分比;与 Associative Registers 数量相关的比率
- 有效访问时间 (EAT) – 内存访问时间标准化为 1
$$\text{食物} = \frac{(1 + \%) \times + (2 + \%) \times (1 - \%)}{2 + \% - \%}$$
- 考虑 \% = 80%, \% = 20ns 用于 TLB 搜索, 100ns 用于内存访问 $EAT = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$
- 考虑更真实的命中率 $\rightarrow \% = 99\%$, \% = 20ns 用于 TLB 搜索, 100ns 用于内存访问
 - 吃 = $0.99 \times 120 + 0.01 \times 220 = 121\text{ns}$



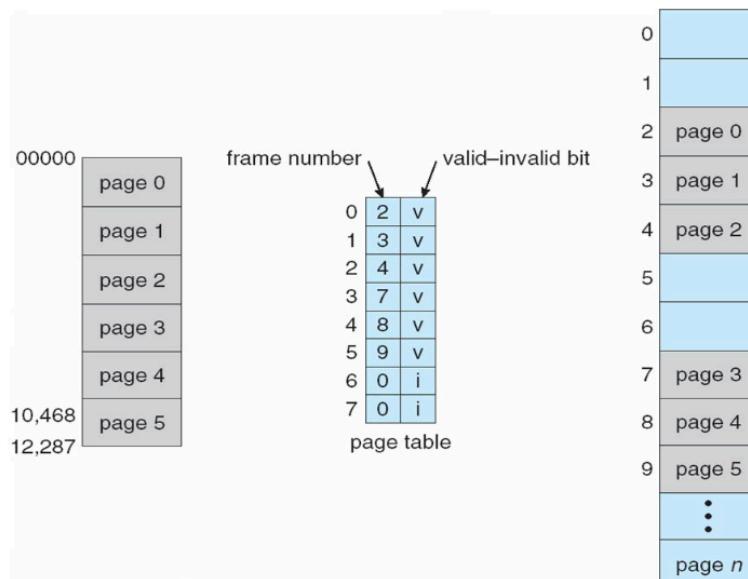
内存保护

- 分页中的内存保护由与每个帧关联的保护位完成。通常，这些位保存在页表中。
- 一个 bit 可以将一个页面定义为读写或只读，或者多个 bits 来表示只执行 Valid-invalid 位附加到页表中每个条目：
 - “valid” 表示关联的页面位于进程的逻辑地址空间中 “invalid” 表示该页面不在进程的逻辑地址空间中 操作系统为每个页面设置此位，以允许或禁止访问该页面
 -
 -
 -
 - 任何冲突都会导致内核陷阱
 - 示例：14 位地址空间（0 到 16383）。如果程序只使用地址 0 到 10468，页面大小为 2KB，则页面 0-5 有效，页面 6-7 无效。
 - 一个进程很少使用它的所有地址范围，通常只使用可用地址空间的一小部分。创建一个页表，其中包含地址范围内每个页面的条目，这将是浪费的，并且大多数表条目都是未使用的，但会占用内存空间 使用页表长度寄存器（PTLR）来指示页面的大小





页表中的有效 (v) 或无效 (i) 位



操作系统概念 - 第 10 版

9.39



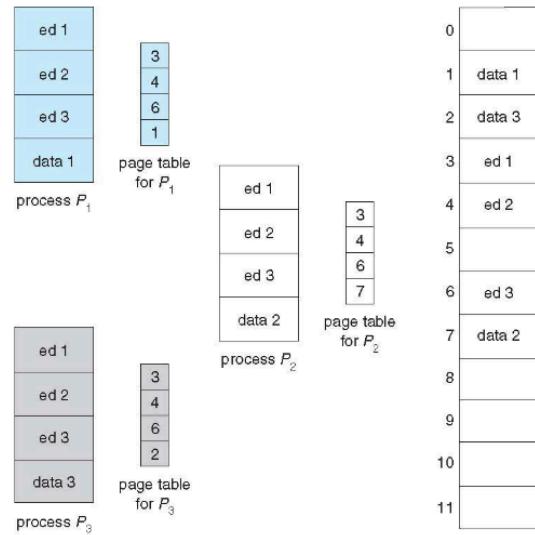
共享页面

共享代码

- 在进程（即文本编辑器、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
- 如果允许共享读写页面，则对 IPC 也很有用

私有代码和数据

- 每个进程都保留代码和数据的单独副本
- 私有代码和数据的页面可以出现在逻辑地址空间中的任何位置



操作系统概念 - 第 10 版

9.40





示例：intelx86 页表条目

- 页表条目或 PTE 中有什么？对于页面翻译，每个页表都由许多 PTE 权限位组成：有效、只读、读写、只写

- 示例：Intel x86 体系结构 PTE：

- 地址与上一张幻灯片的格式相同（10 位、10 位、12 位偏移量）称为“目录”的中间页表

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	CD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: 存在（与其他架构中的“有效”位相同） W: 可写

U: 用户可访问 PWT: 页面写入透明：外部缓存直写 PCD: 页面缓存已禁用（无法缓存页面） A: 已访问：页面最近被访问 D: 脏（仅限 PTE）：页面最近被修改 L: L=1MB 页面（仅限目录）。

虚拟地址的底部 22 位用作页面内的偏移量



页表的结构

- 使用直接方法，分页的内存结构可以极大地增长

- 考虑现代计算机上的 32 位逻辑地址空间页面大小为 4 KB (2¹²)

-

- 页表将有 100 万个条目 (2²² / 2¹²) 如果每个条目为 4 字节 → 仅页表就有 4 MB 的物理地址空间/内存

-

- 用于消耗大量内存的内存量 □ 页表也不能在主内存中连续分配，这将是

分配到多个页面（帧） – 对页表进行分页，这意味着需要另一个页表来跟踪内存中的页表

- 页面大小 4 MB (2²²) 会导致页表包含 1,000 个条目，问题更少 64 位逻辑地址呢？

-

- 分层分页哈希页表

-





分层页表

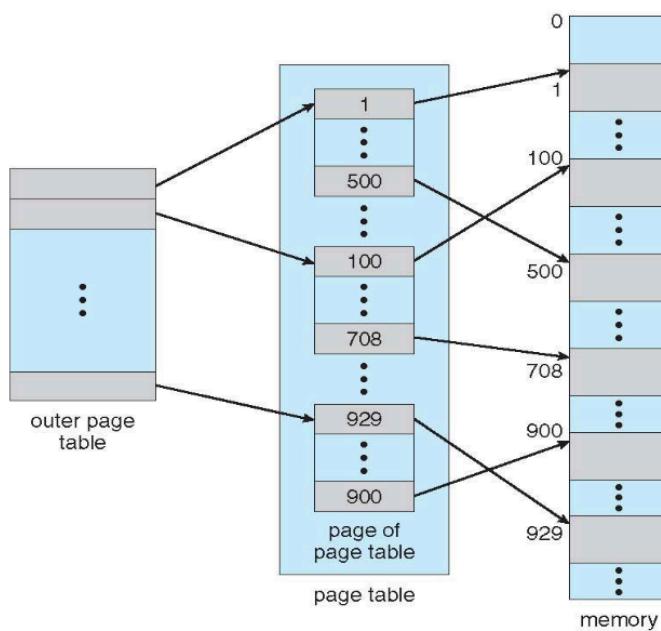
- 将逻辑地址空间分解为多个页表
- 一种简单的技术是两级页表或分层页表
- 对页表进行分页

操作系统概念 - 第 10 版

9.43



两级页表方案



操作系统概念 - 第 10 版

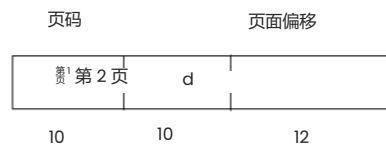
9.44





两级分页示例

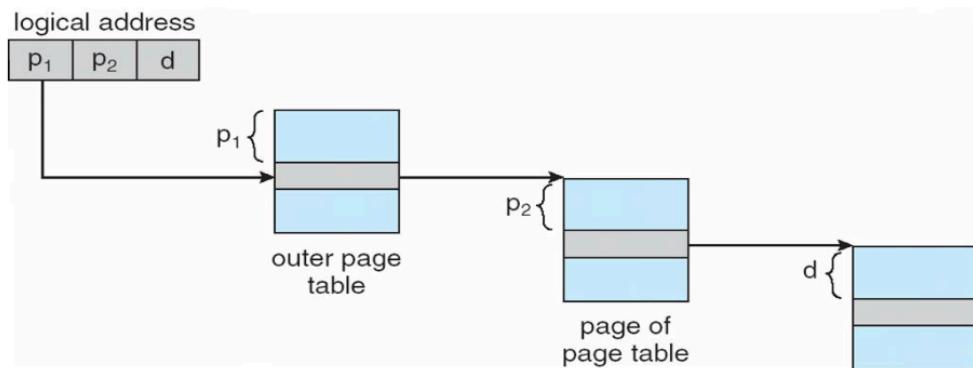
- 逻辑地址（在具有 4K 页面大小的 32 位计算机上）分为：
 - 由 20 位组成的页码，由 12 位组成的页偏移
 - 移量
- 由于页表是分页的，因此页码进一步分为：
 - 一个 10 位页码 P2，如果每个 PTE 占用 4 个字节，则必须为 10 位，确保每个内页表可以存储在一个帧（页）中，
 - 10 位页面偏移量 P1。如果 PTE 也占用 4 个字节，则 P1 中的位数不能超过 10，否则需要进一步划分 – 超过两个级别
- 因此，逻辑地址如下所示：



- 其中 p1 是外页表的索引（在 Intel 架构中，这称为“目录”），p2 是内页表页面内的位移
- 由于地址转换从外部页表向内工作，因此此方案也称为正向映射页表



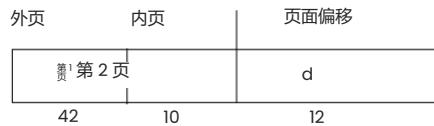
地址转换方案





64 位逻辑地址空间

- 即使两级分页方案也不够 如果页面大小为 4 KB (2¹²)
 - 然后页表有 252 个条目如果是两级方案, 内页表可以是 210 个 4 字节条目 地址看起来像
 -
 -

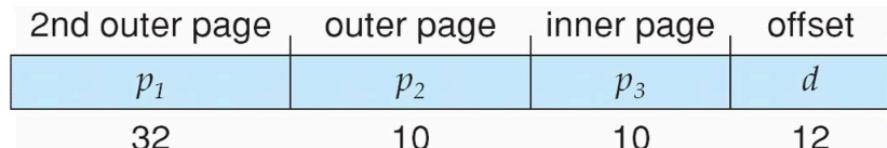
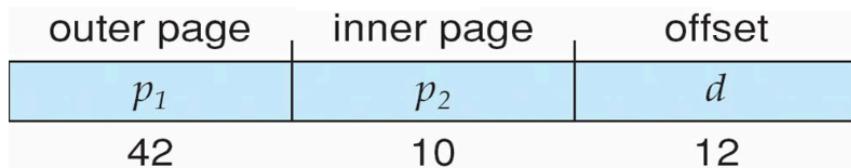


- 外部页表有 242 个条目或 244 字节 一种解决方案是添加第二个外部页表 但是在以下示例中, 第二个外部页表的大小仍然是 234 字节 (16 GB) 并且可能需要 4 个内存访问才能到达一个物理内存位置
 -
 -

¤ 64 位 UltraSPARC 需要 7 级分页 (令人望而却步的内存访问次数) 来转换每个逻辑地址



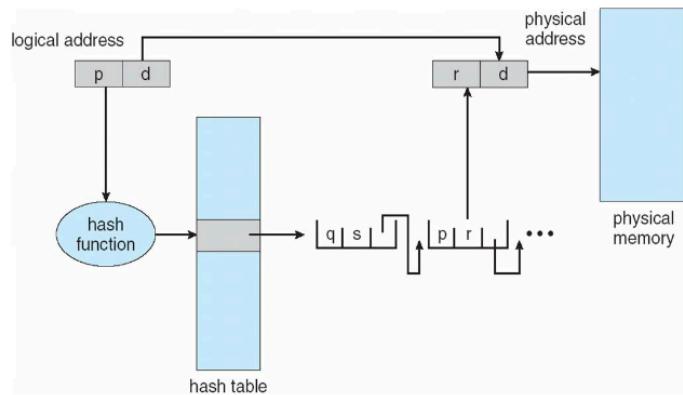
三级分页方案





哈希页表

- 常见于地址空间 > 32 位页码被哈希处理到页表中
- 此页表包含一组元素，这些元素散列到同一位置每个元素都包含 (1) 页码 (2) 映射的页帧的值 (3) 指向下一个元素的指针
- 在此链中比较页码以查找匹配项
 - 如果找到匹配项，则提取相应的物理帧



操作系统概念 - 第 10 版

9.49



示例：Intel 32 位和 64 位体系结构

- 优势行业芯片
 - 16 位 Intel 8086 (1970 年代后期) 和 8088 用于原始 IBM PC
- Pentium CPU 是 32 位的，称为 IA-32 架构
 - 它支持分段和分页 - CPU 生成逻辑地址，这些地址将提供给分段单元。分段单元为每个逻辑地址生成一个线性地址。然后将线性地址提供给分页单元，分页单元反过来在主内存中生成物理地址。
- 当前的 Intel CPU 是 64 位的，称为 IA-64 架构
 - 目前最流行的 PC 操作系统在 Intel 芯片上运行，包括 Windows、MacOS 和 Linux (Linux 也运行在其他几种架构上)
 - 英特尔的主导地位尚未蔓延到移动系统，它们主要使用 ARM 架构
- 芯片的许多变化，这里只介绍主要思想



操作系统概念 - 第 10 版

9.50



示例：Intel IA-32 架构

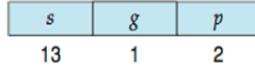
□ 支持分段和带分页的分段

- 每个段可以最大 4GB (32 位) 进一步划分为页面每个进程最多可以有 16K 个段 (14 位)，分为两个分区
 - 第一个分区最多 8 K 个段，专用于进程 (保存在本地描述符表 (LDT) 中，第二个分区，最多 8K 个段，在所有进程之间共享 (保存在全局描述符中表 (GDT))

表 (GDT))

□ CPU 生成逻辑地址

- 给分段单元的选择器 - 产生线性地址 - s 表示段编号，g 表示段是 LDT 还是 GDT，p 处理保护



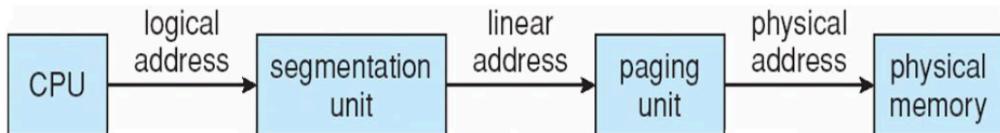
- 然后，将线性地址分配给分页单元
 - 在主存储器中生成物理地址 □ 分页单元形式等同于 MMU

□ 页面大小可以是 4 KB 或 4 MB



Intel IA-32 分段

- LDT 和 GDT 中的每个条目都由一个 8 字节的段描述符组成，其中包含有关特定段的详细信息，包括基本位置和限制段寄存器指向 LDT 或 GDT 中的相应条目。有关段的 base 和 limit 信息用于生成线性地址。
-

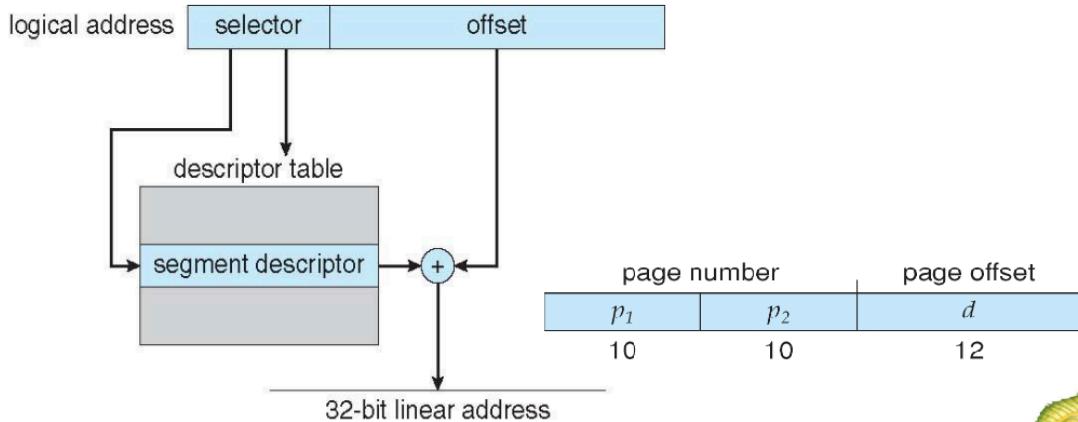




Intel IA-32 分段

IA-32 上的线性地址长度为 32 位，其格式如下

- 该限制用于检查地址的有效性。如果地址无效，则生成内存故障，trap 到操作系统。如果有效，则将偏移量的值与基数的值相加，从而得到 32 位线性地址（即每个段可以是 4GB）

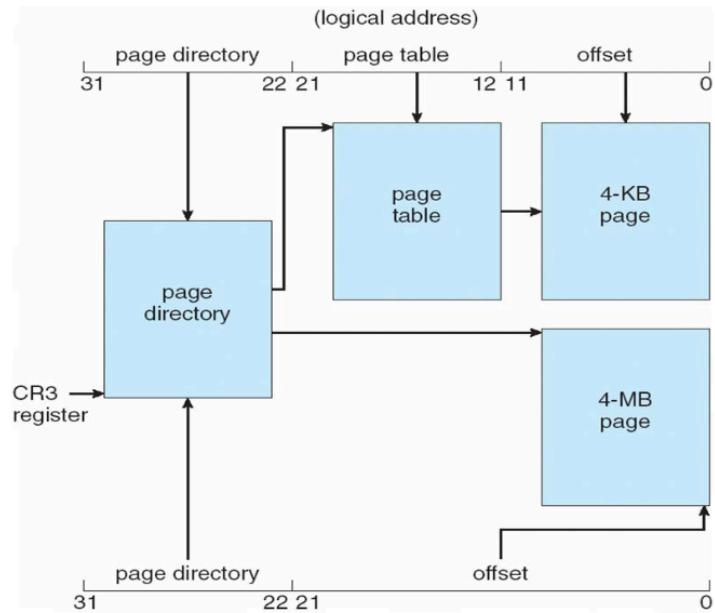


操作系统概念 - 第 10 版

9.53



32 分页体系结构



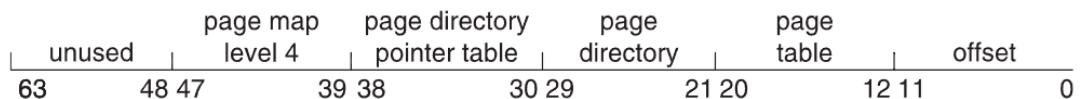
操作系统概念 - 第 10 版

9.54





- 当前一代 Intel x86-64 架构 64 位是巨大的 (> 16 EB) 在实践中只实现 48 位寻址
-
-
- - 页面大小为 4 KB、2 MB、1 GB 分页层
 - 次结构的四个级别
- 也可以使用 PAE (页面地址扩展)，因此虚拟地址为 48 位，物理地址为 52 位 (4096 TB)



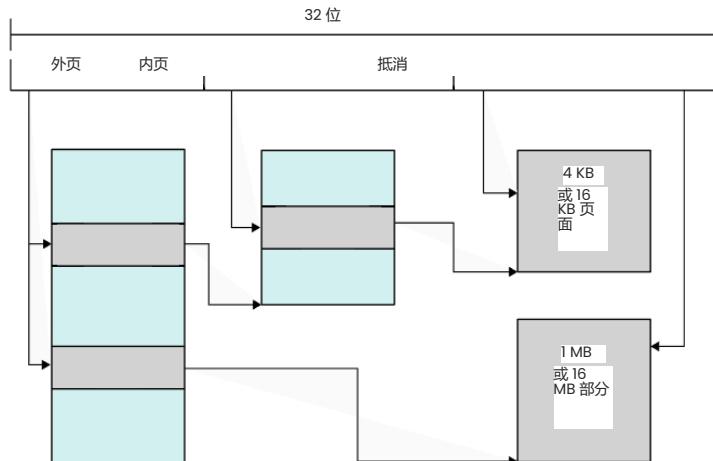
操作系统概念 - 第 10 版

9.55



示例：ARM 架构

- 占主导地位的移动平台芯片 (例如 Apple iOS 和 Google Android 设备)
- 现代、节能的 32 位 CPU
- 4 KB 和 16 KB 页面 1 MB 和 16 MB 页 (称为部分)
-
- 部分的一级分页，较小页面的两级分页 两级 TLB
- - 外层有两个微型 TLB (一个数据，一个指令)
 - 内部是单个主 TLB 首先检查内部，检查错过的外部，以及错过页面时遍历
-



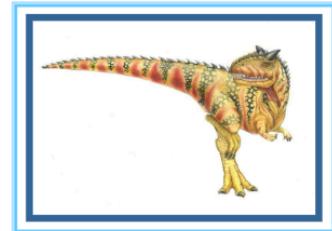
由 CPU 执行

操作系统概念 - 第 10 版

9.56



第 9 章结束



操作系统概念 - 第 10 版