Lecture5 内存管理概述

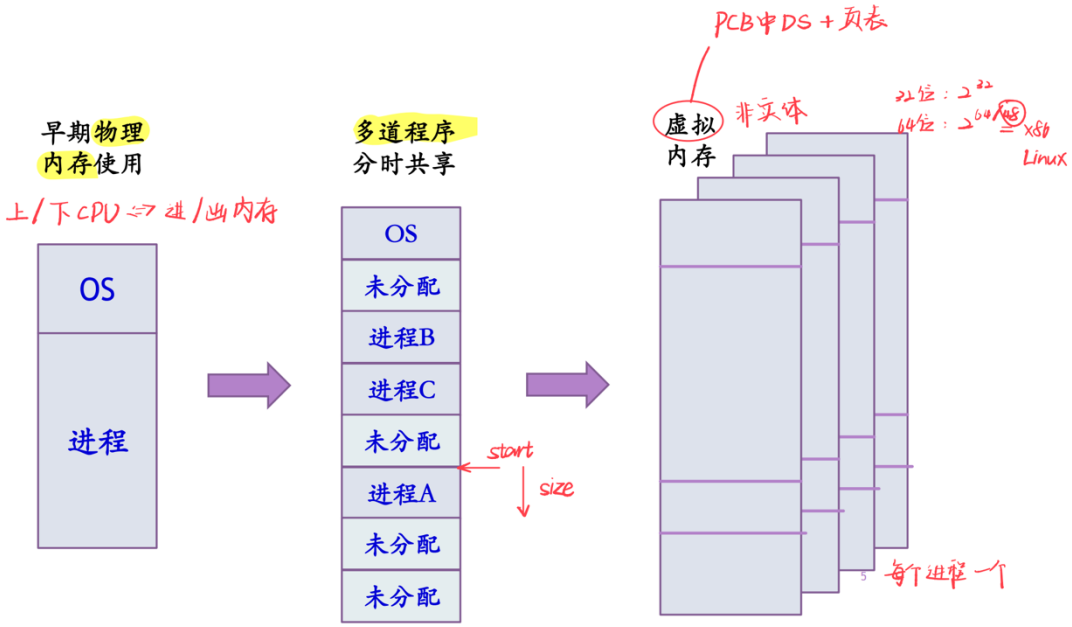
基本问题与概念

1. 问题
    1) 内存管理从简单到复杂的演化过程是什么？——多道程序设计
    2) 怎样理解“地址空间（address space）是对内存的抽象”？
    3) 内存管理追求的目标有哪些？
    4) 程序什么时候进入内存？Lazy allocation
    5) 进程空间与物理内存空间怎么对应？内存管理的核心
    6) 物理内存空闲空间怎么管理？有哪些数据结构和算法？ 空闲链表...
    7) 怎样“扩充”内存？内存小而程序大
    8) Linux 的伙伴系统的作用？
2. 重要概念
    1) 地址重定位（地址转换，地址变换，地址翻译，地址映射）
    2) 逻辑地址与物理地址
    3) 存储保护
    4) 存储共享
    5) 单一连续区
    6) 固定分区
    7) 可变分区
    8) 页式
    9) 段式
    10) 段页式
    11) 局部性原理
    12) 存储体系
    13) 覆盖技术
    14) 交换技术
    15) 地址空间

大纲

- 重要概念
  - 存储体系、存储保护、地址重定位
- 物理内存管理
  - 数据结构（位示图、空闲区表、空闲区链表）
  - 分配算法（首先适配、最佳适配、最差适配 ……）
- 各种内存管理方案
  - 单一连续区、固定分区、可变分区、页式、段式、段页式
- 内存“扩充”
  - 紧凑技术、覆盖技术、交换技术、虚存技术

内存管理的演化历史



内存管理的基础知识

1. 常识

   1) 程序装载(加载)到内存才可以运行 文件 >VM>Page Fault> 进 DMEM

      程序以文件形式（可执行文件格式）保存在磁盘上

   2) **多道程序设计模型**

      允许多个程序同时进入内存

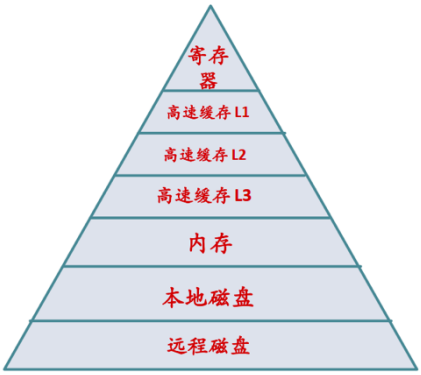
   3) 每个进程有自己**独立**的（同样大的）地址空间

      a) 一个进程执行时不能访问另一个进程的地址空间 保护

      b) 不能执行不适当的操作

4) 局部性原理

5) 存储体系



2. 多道程序设计与内存管理

1) 内存管理要做更多的工作

a) 为什么？

因为同时有多个用户程序在内存中

b) 需要支持**地址重定位**

程序中的地址不一定是最终的物理地址，因为在程序运行前无法计算出物理地址（物理地址程序不可见，由 os 执行地址转换，程序发出的地址应与物理内存地址无关），不能确定程序被加载到内存什么地方

c) 需要支持**地址保护**

进程之间不能互相访问（不能访问其他进程的内存）——**地址空间(address space)**是对内存的抽象

3. 地址重定位

1) 简介

a) 定义

- 将用户程序中的逻辑地址转换为运行时可由机器直接寻址的物理地址的过程
- 逻辑地址（相对地址，虚拟地址）
  - ✓ 用户程序经过编译、汇编后形成目标代码，目标代码通常采用相对地址的形式，其首地址为 0，其余指令中的地址都相对于首地址而编址

✓ 不能用逻辑地址在内存中读取信息

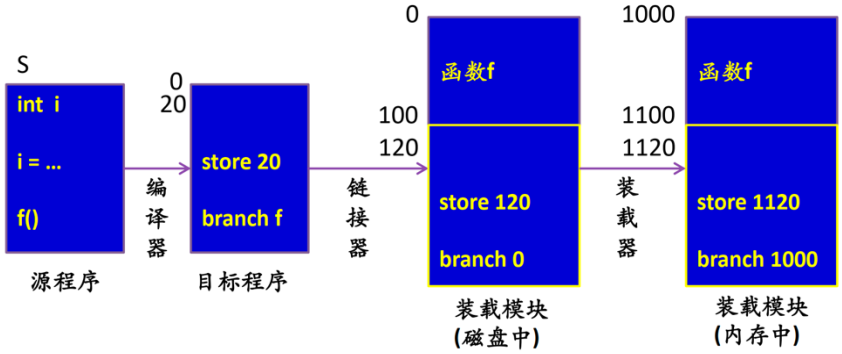
- 物理地址（绝对地址，实地址）
- 内存中存储单元的地址，可直接寻址

b) 目的

保证 CPU 执行指令时可正确访问内存单元

2) 静态地址重定位 不灵活，现在很少用

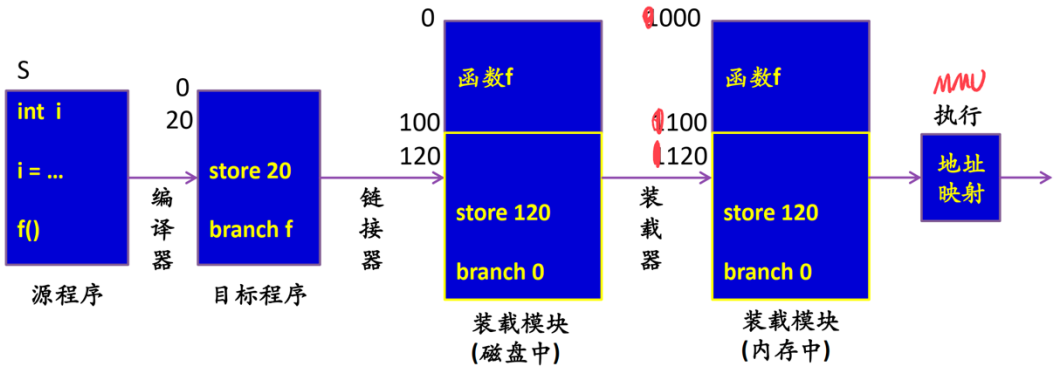
a) 当用户程序加载到内存时，一次性实现逻辑地址到物理地址的转换



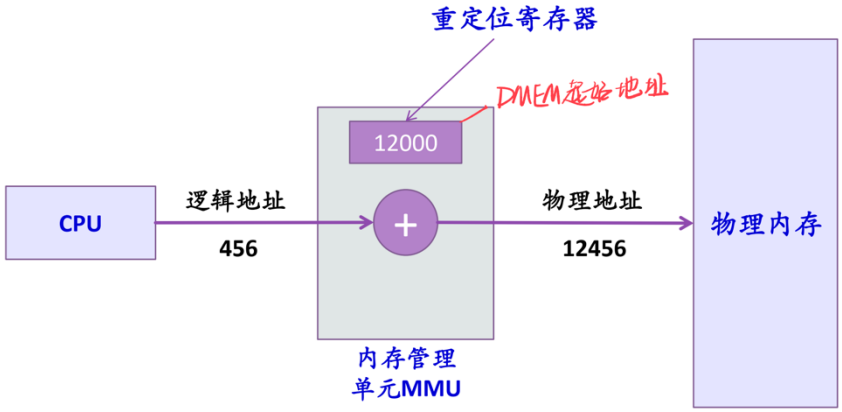
b) 一般可以由软件完成，无需硬件支持

3) 动态地址重定位

a) 在进程执行过程中进行地址变换，即逐条指令执行时完成地址映射



b) 需要硬件支持：MMU



重定位寄存器一般保存物理内存的起始地址

4. 地址保护

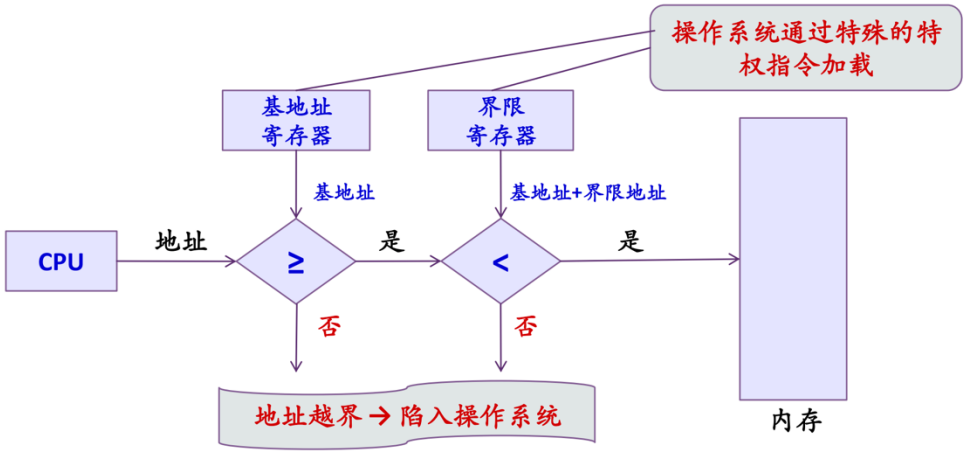
1) 目的：防止地址越界

a) 确保每个进程有独立的私有的地址空间

b) 确定进程可访问的合法地址的范围，以确保进程只访问其合法地址

（虚拟内存的出现自然解决了该问题，为进程提供了分隔）

2) （过去的）实现

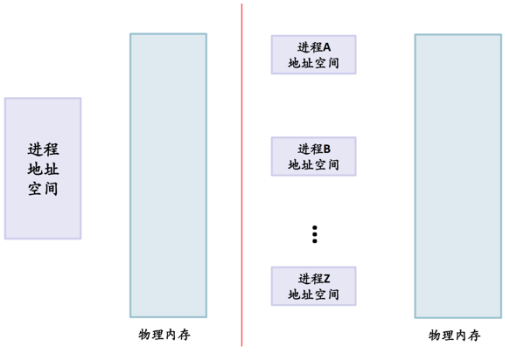


很明显的问题：进程在物理内存中不能被离散化存储

内存管理的目标

1. 内存管理要解决的问题

1) 进程地址空间与物理内存的映射关系 （背景：多道程序设计）



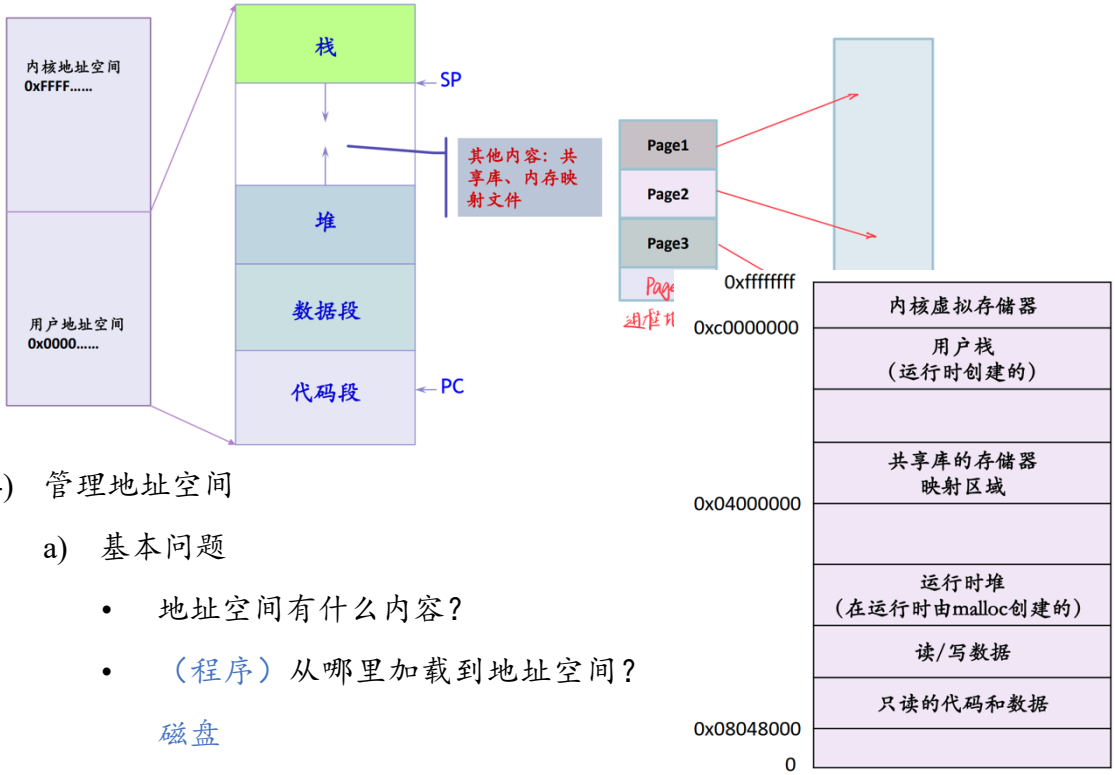
2) 对进程地址空间中已分配内存的管理

a) 连续性 — 离散性

b) 驻留性 — 交换性

c) 一次性 — 多次性

3) 回顾：进程地址空间有什么内容？



4) 管理地址空间

a) 基本问题

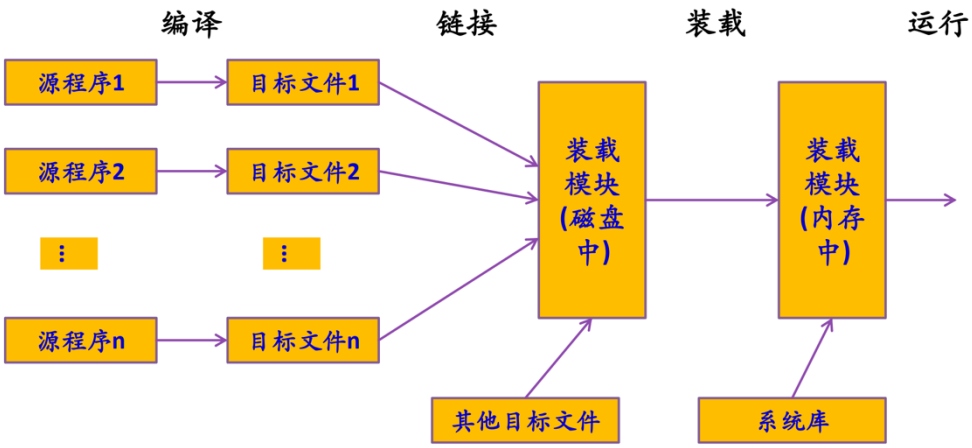
- 地址空间有什么内容？
- （程序）从哪里加载到地址空间？

  磁盘
- 具体的位置？ 由进程的页表决定

b) 物理内存空闲空间的组织、分配、placement（放置）、回收、分割、合并

c) 活跃页面 or 工作集 or 常驻集（如内核所在的部分页常驻物理内存）

5) 回顾：程序执行前的准备过程



问题：何时将指令、数据绑定到物理内存地址？ 运行时（理论上在链接/加载阶段也可以指定进程在物理内存的何处运行，但是这样很不灵活，难以管理）

2. 内存管理的目标

1) 基本功能：虚拟化内存——透明性（对用户进程而言）、效率（如 TLB）、保护

a) 给进程分配内存——地址空间

- 创建进程时（PCB）

b) 往内存加载内容——映射进程地址空间到物理内存

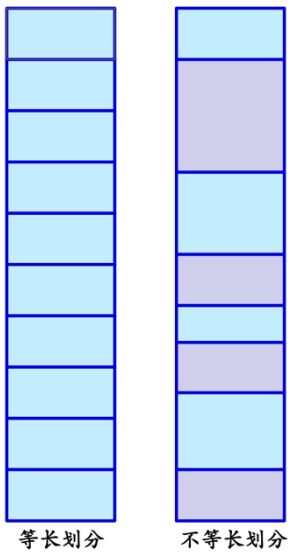
- e.g., 通过 Page Fault 来实现

c) 存储保护——地址越界、权限

d) 管理共享的内存

e) 最小化存储访问时间

物理内存管理方案



1. 空闲物理内存管理

1) 数据结构

a) 位图（一般对应等长划分）

每个分配单元对应于位图中的一位，0 表示空闲，1 表示占用（或者相反）

b) 空闲区表、已分配区表（一般对应不等长划分）

表中每一项记录了空闲区（或已分配区）的起始地址、长度、标志

c) 空闲块链表

隐式空闲链表、显式空闲链表、分离空闲链表

2) 衡量指标

a) 内存资源利用率——尽可能减少资源浪费

- 内碎片（一般出现在等长划分中）
- 外碎片（一般出现在不等长划分中）

b) 性能——尽可能降低分配延迟，节约 CPU 资源

2. 内存分配算法

1) 首次适配 first fit

在空闲区表中找到第一个满足进程要求的空闲区

2) 下次适配 next fit

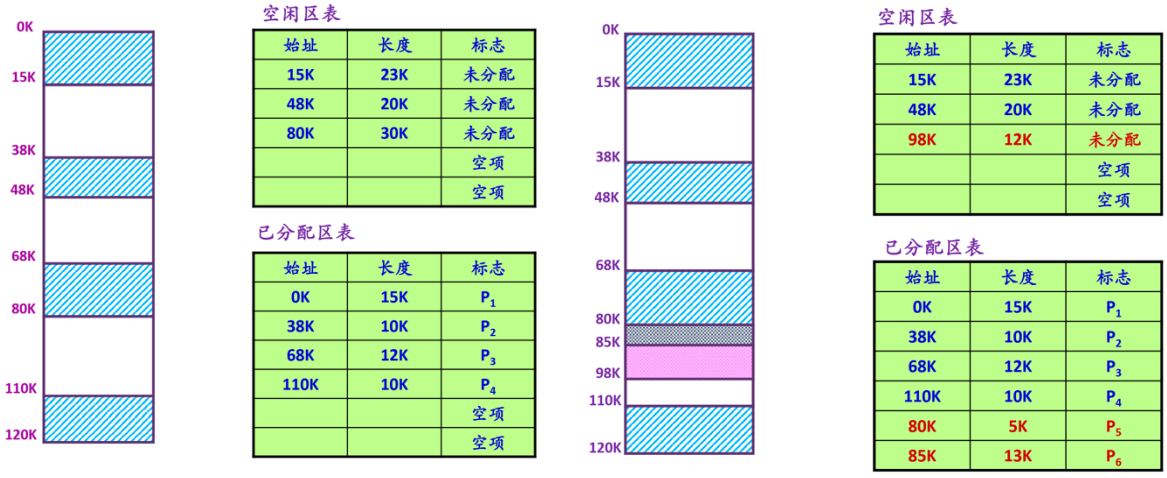
从上次找到的空闲区处接着查找

3) 最佳适配 best fit

查找整个空闲区表，找到能够满足进程要求的最小空闲区

4) 最差适配 worst fit

总是分配满足进程要求的最大空闲区，将该空闲区分为两部分，一部分供进程使用，另一部分形成新的空闲区



3. 内存回收算法（Malloclab）

1) 当某一块归还后，前后空闲空间合并，修改内存空闲区表

2) 四种情况：上相邻、下相邻、上下都相邻、上下都不相邻

4. 物理内存管理实例

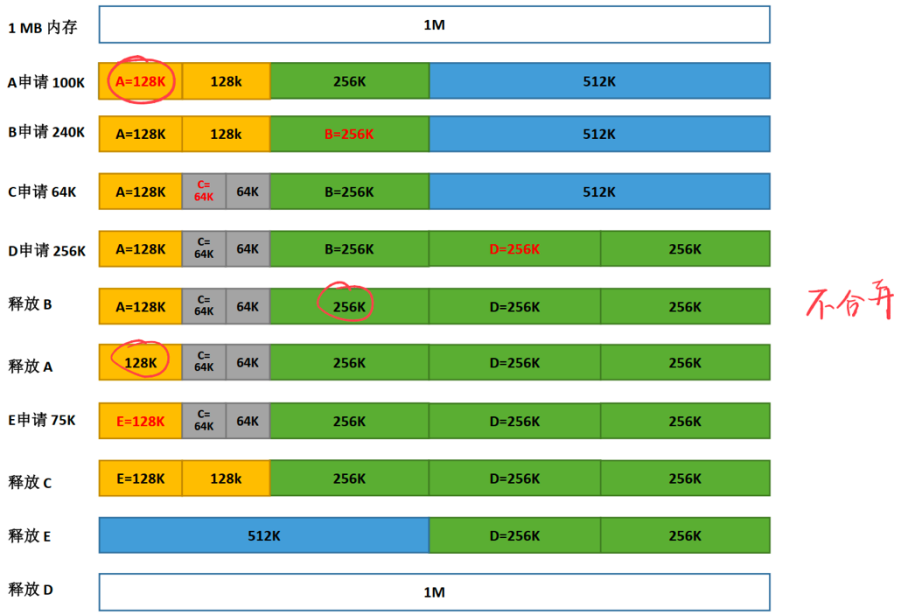
1) 伙伴系统（buddy system）

a) 简介

- 一种经典的内存分配方案
- Linux 底层内存管理采用

b) 主要思想

- 将内存按 2 的幂进行划分，组成若干空闲块链表
- 查找该链表找到能满足进程需求的最佳匹配块



c) 算法：一种特殊的“分离适配”算法

- 首先将整个可用空间看作一块：2^U
- 假设进程申请的空间大小为 s，如果满足：$2^{U-1} < s <= 2^U$，则分配整个块；否则，将块划分为两个大小相等的伙伴，大小为 2^{U-1}
- 一直划分下去直到产生大于或等于 s 的最小块

2) SLAB 分配器

a) 简介

- 上世纪 90 年代，Jeff Bonwick 最先在 Solaris 2.4 操作系统内核中设计并实现 SLAB 分配器
- Linux 内核中的内存分配器，用于在运行时动态地管理内核内存分配和释放，追求可靠性和高性能
- 广泛应用于许多 Linux 内核模块中，例如文件系统、网络协议栈等，能够满足大多数内核模块的需求

b) 作用

满足操作系统（频繁的）分配小对象的需求，把大块内存进一步细分成小块内存进行管理（只分配固定大小的内存块，块大小通常是 2n 个字节（$3 \leqslant n < 12$））

在运行中动态管理部分物理内存，满足小对象的内存分配

c) 实现方法

- 将可用内存按照大小划分为不同的区域（按照大小分级管理），

每个区域有一个空闲对象列表

- 需要分配内存时，SLAB 在合适的空闲对象列表中寻找一个大小与请求相符的空闲区域进行分配
- 释放内存时，将该内存块添加回相应的空闲对象列表中

d) 问题

- 维护了太多的队列、实现日趋复杂、存储开销增大
- 在高并发环境下容易出现锁竞争，多核系统中可能会出现性能问题（单核环境下可以通过加锁/解锁解决，但是多核环境下就不行了）

e) 改进

- SLUB 分配器
  - ✓ 2007 年由 Pekka Enberg 开发，被合并到 Linux 内核 2.6.22 版本中（？）
  - ✓ SLUB 机制可为体系结构而做优化，比如针对多核、多节点（NUMA）的本地性设计
  - ✓ 采用了简单的数据结构，减少了锁竞争，提高了多核系统下的并发性能
- SLOB 分配器
  - ✓ 最早出现在 Linux 内核 2.2 版本
  - ✓ 简单的 Linux 内核内存分配器
  - ✓ 设计目标：尽可能的简单和紧凑，以减少代码大小和内存开销
  - ✓ 适用于小型嵌入式系统和资源受限的环境

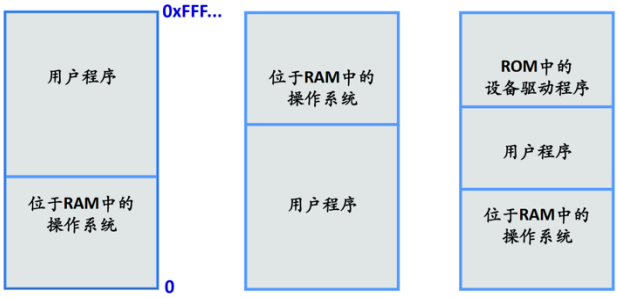
物理内存管理方案

1. 内存管理基本方案(汇总)

单一连续区	每次只运行一个用户程序，用户程序独占内存，它总是被加载到同一个内存地址上
固定分区	把可分配的内存空间分割成若干个连续区域，每一区域称为分区。每个分区的大小可以相同也可以不同，分区大小固定不变，每个分区装一个且只能装一个进程
可变分区	根据进程的需求，把可分配的内存空间分割出一个分区，分配给该进程
页式	把用户程序地址空间划分成大小相等的部分，称为页。内存空间按页的大小划分为大小相等的区域，称为内存块（物理页面，页框，页帧）。以页为单位进行分配，逻辑上相邻的页，物理上不一定相邻
段式	用户程序地址空间按进程自身的逻辑关系划分为若干段，内存空间被动态的划分为若干个长度不相同的区域（可变分区）。以段为单位分配内存，每一段在内存中占据连续空间，各段之间可以不连续存放
段页式	用户程序地址空间：段式；内存空间：页式；分配单位：页

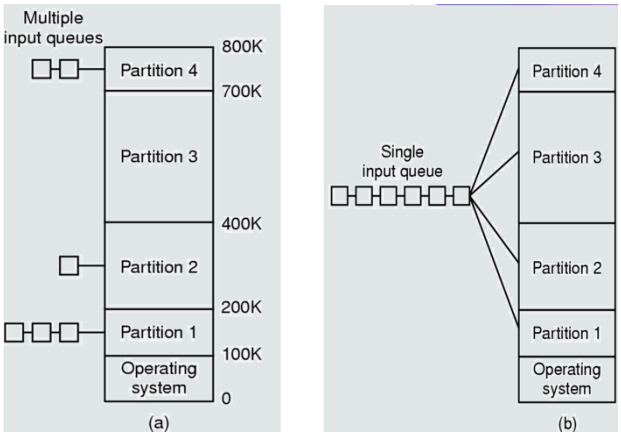
本质是进程地址空间与物理内存的映射关系，加载单位是进程。前三种无需硬件支持，后三种需要硬件支持

2. 单一用户（连续）区



   1) 特点
      a) 一段时间内只有一个进程在内存
      b) 简单，内存利用率低

3. 固定分区（背景：多道程序）

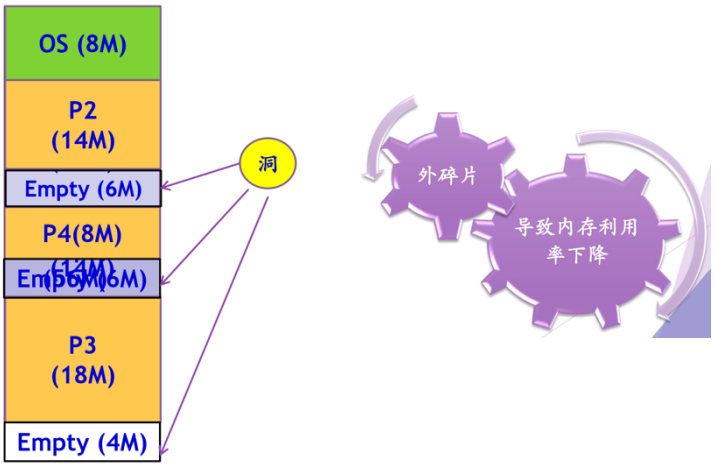


   1) 把内存空间分割成若干区域，称为分区。每个分区的大小可以相同也可以不同
   2) 内存被分割为分区后大小固定不变
   3) 每个分区装一个且只能装一个进程

4. 可变分区

1) 简介

根据进程的需要，把内存空闲空间分割出一个分区，分配给该进程，剩余部分成为新的空闲区



手机的物理内存管理：固定分区到可变分区

2) 问题及解决方案

a) 碎片：很小的、不易利用的空闲区；导致内存利用率下降

b) 解决方案：紧缩技术（memory compaction，压缩技术，紧致技术，搬家技术）

- 在内存中移动程序，将所有小的空闲区合并为较大的空闲区
- 紧缩时要考虑的问题：系统开销、移动时机...

  进程在 I/O 时不能移动（DMA 传送）

5. 页式内存管理

1) 设计思想

a) 将进程地址空间划分为大小相等的区域——页（page）

b) 划分是由系统自动完成的，对用户是透明的（用户只需提供填入地址空间的内容即可）

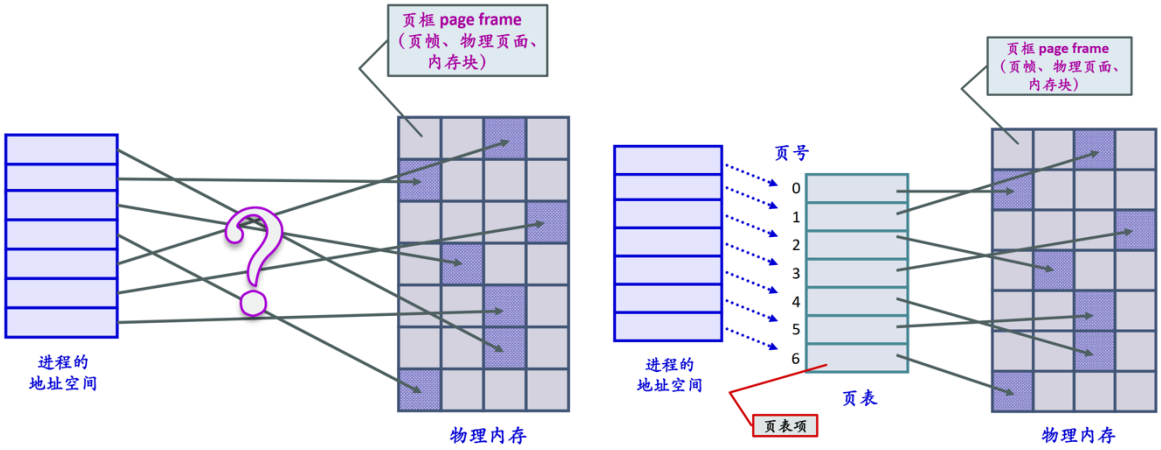
c) 内存空间同样按页大小划分为大小相等的区域，称为内存块（page frame，物理页面，页框，页帧）

2) 内存分配规则

a) 以页为单位进行分配，并按进程需要的页数来分配

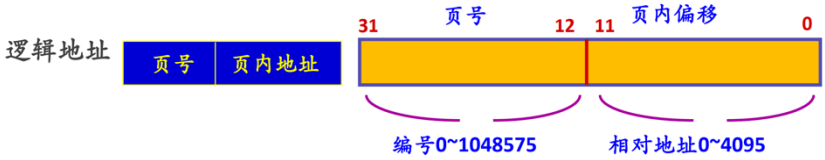
b) 逻辑上相邻的页，物理上不一定相邻

c) 典型的页面尺寸： 4K 或 4M



3) 地址转换（硬件和数据结构）

a) CPU 取到逻辑地址，自动划分（MMU）为页号和页内地址；用页号查页表，得到页框号，再与页内地址（页内偏移）拼接为物理地址



b) 页表

- 页表项（PTE）：记录了逻辑页号与页框号的对应关系
- 每个进程一个页表，存放在物理内存中

  在多级页表中只有一级页表需要常驻内存；页表的起始地址在进程的 PCB 中，当进程被调度上 CPU 运行时填入特定的页表基址寄存器（如 CR3）
- 页表起始地址保存在何处？

c) 需要考虑的潜在问题

- 内碎片
- 空闲内存管理

6. 段式内存管理

1) 设计思想

a) 用户程序地址空间：按程序自身的逻辑关系划分为若干个程序段，每个程序段都有一个段名

b) 内存空间被动态的划分为若干个长度不相同的区域，称为物理段，每

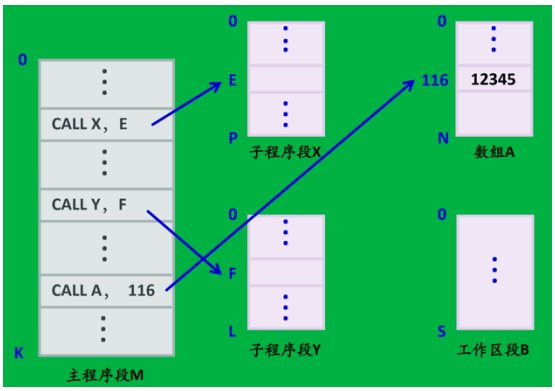
个物理段由起始地址和长度确定

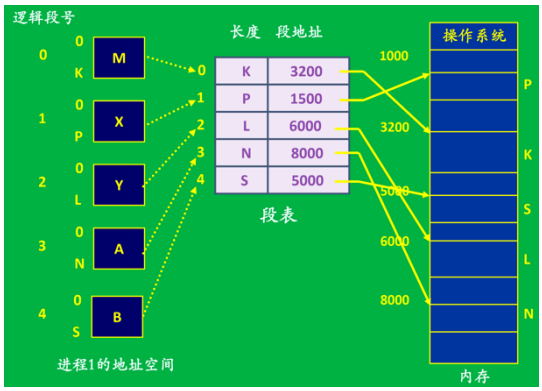
2) 内存分配规则

逻辑地址 | 段号 | 段内地址

a) 以段为单位进行分配

b) 每一个段在内存中占据连续空间，但各段之间可以不相邻





3) 地址转换（硬件和数据结构）

a) CPU 取到逻辑地址，用段号查段表，得到段的起始地址，再与段内偏移地址计算出物理地址

b) 段表

- 记录了段号、段首地址和段长度之间的关系
- 每个进程一个段表，存放在内存
- 段表起始地址保存在何处？

c) 需要考虑的潜在问题

- 物理内存管理（同可变分区）

7. 段页式内存管理

1) 产生背景

结合页式段式优点，克服二者的缺点

2) 设计思想

用户程序划分：按段式划分（对用户来讲，按段的逻辑关系进行划分；对系统讲，按页划分每一段）

3) 内存划分：按页式存储管理方案

内存分配：以页为单位进行分配

4) 地址转换（硬件与数据结构）

逻辑地址：

段号	段内地址	
	页号	页内地址

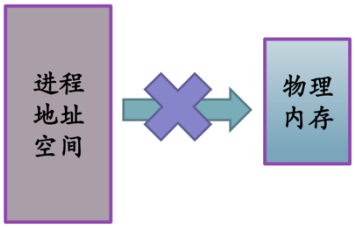
a) 段表：记录了每一段的页表

始址和页表长度

b) 页表：记录了逻辑页号与内存块号的对应关系（每一段有一个，一个程序可能有多个页表）

c) 需要考虑的潜在问题

- 空闲区管理：同页式管理
- 分配、回收：同页式管理



内存的扩充

1. 简介

1) 问题：内存不足时如何管理？

2) 目标：解决在较小的存储空间中运行较大程序时遇到的矛盾

3) 主要办法

a) 内存紧凑（例如：可变分区） 减少碎片

b) 覆盖技术

c) 交换技术（虚存技术的前身）

d) 虚存技术

2. 覆盖技术(Overlaying)

1) 简介

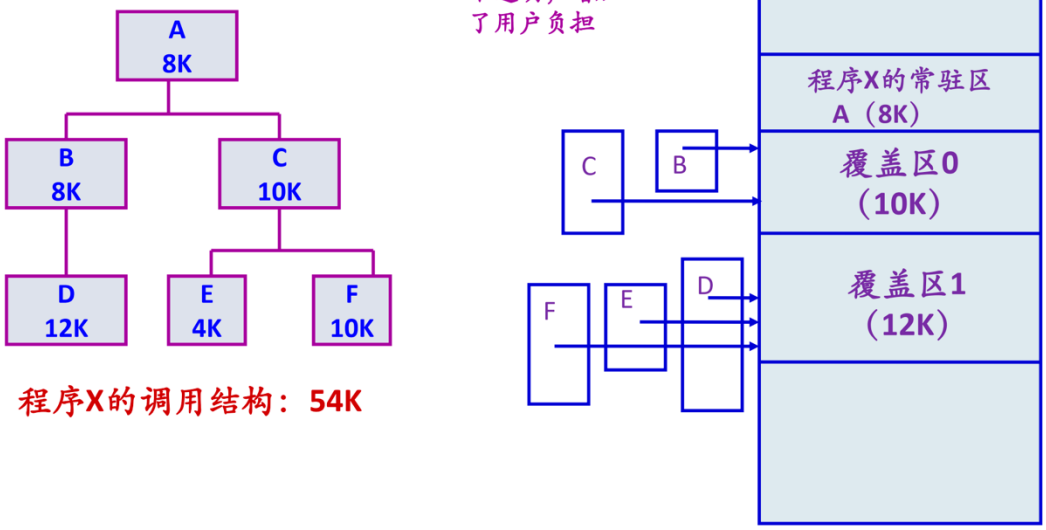
a) 问题：程序大小超过物理内存总和

b) 设计思想

- 程序执行过程中，程序的不同部分在内存中相互替代
  - ✓ 按照其自身的逻辑结构将那些不会同时执行的程序段共享同一块内存区域
  - ✓ 要求程序各模块之间有明确的调用结构
- 程序员声明覆盖结构，操作系统完成自动覆盖

c) 主要用在早期的操作系统，现在很少用

2) 示例



3) 问题

a) 增加编程难度

- 需程序员划分功能模块，并确定模块间的覆盖关系

b) 增加执行时间

- 从外存装入覆盖模块
- 时间换空间

3. 交换技术(Swapping)

1) 简介

a) 最早用于小型分时系统："roll in roll out"

上/下 CPU ⇔ 进/出物理内存

b) 设计思想

内存空间紧张时，系统将内存中某些进程暂时移到外存，把外存中某些进程换进内存，占据前者所占用的区域（进程在内存与外存之间的动态调度）

2) 实现时的问题

a) 进程的什么部分需要交换到磁盘？在磁盘的什么位置保存被换出的进程？

- 进程运行时创建或修改的内容：栈和堆（进程在磁盘上没有的部分，.rodata, .text 之类的不用交换）

- 交换区：一般系统会指定一块特殊的磁盘区域作为交换空间（swap space），包含连续的磁道，操作系统可以使用底层的磁盘读写操作对其高效访问

b) 交换时机？

- 只要不用就换出（很少再用）；内存空间不够或有不够的危险时换出 → 与调度器结合使用

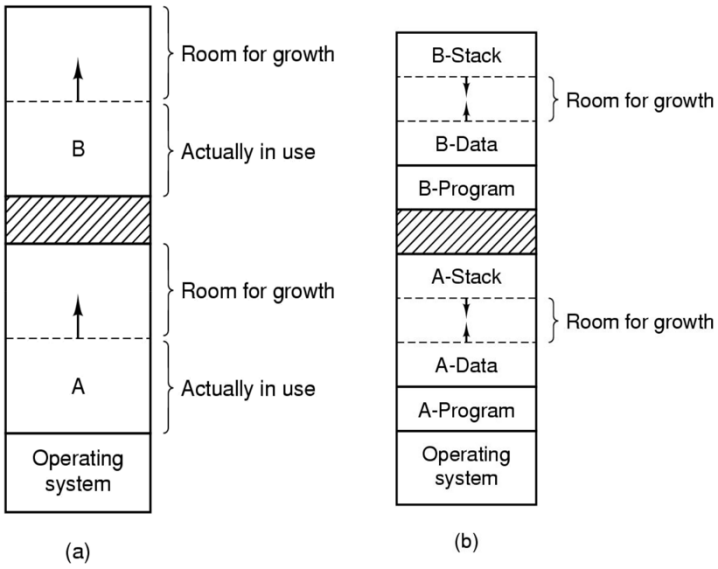
c) 如何选择被换出的进程？

- 考虑进程的各种属性；不应换出处于等待 I/O 状态的进程（和内存紧缩技术类似）

d) 换出后再换入的进程是否回到原处？

- 换出后又换入的进程不一定回到原处（采用动态重定位）

e) 如何处理进程物理内存空间（数据段、栈段、堆段）的增长？



预留空间，相向增长

重点小结

1. 基本概念
   1) 存储体系
   2) 交换与覆盖技术
   3) 逻辑地址、物理地址、地址重定位
   4) 地址保护
2. 物理内存管理

1) 位示图、空闲区表/已分配区表、空闲块链表、伙伴系统
2) 分配与回收算法
3) 碎片问题（内碎片、外碎片）

3. 各种内存管理方案
1) 单一连续区、固定分区、可变分区、分页、分段、段页式
2) 相关数据结构
3) 地址转换过程