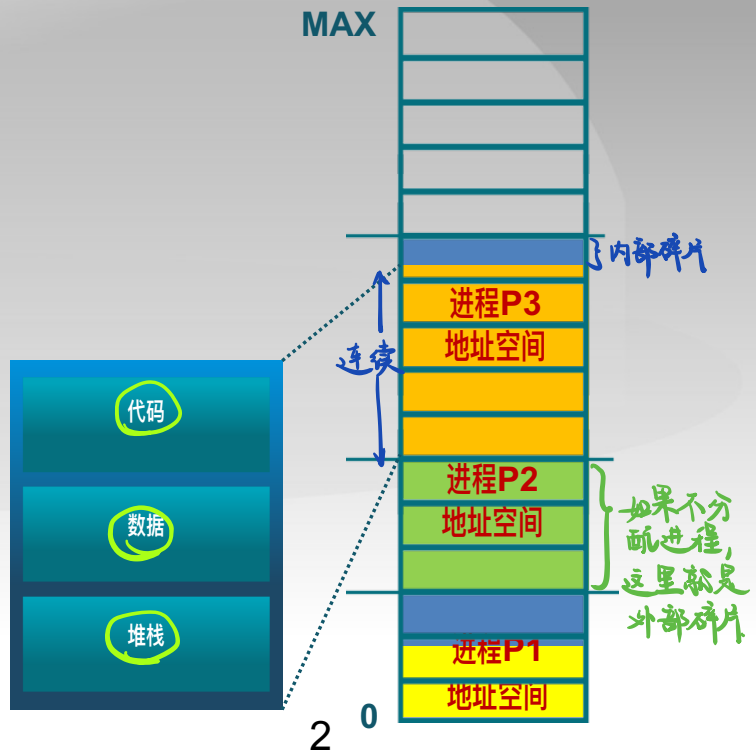
5.1 连续内存分配

- 连续内存分配
- 动态分区分配策略
- 碎片整理
- 伙伴系统

连续内存分配和内存碎片

- 连续内存分配
  - 给进程分配一块不小于指定大小的连续的物理内存区域
- 内存碎片
  - 空闲内存不能被利用
- 外部碎片
  - 分配单元之间的未被使用内存
- 内部碎片
  - 分配单元内部的未被使用内存
  - 取决于分配单元大小是否要取整



连续内存分配：动态分区分配

- 动态分区分配
  - 当程序被加载执行时，分配一个进程指定大小可变的分区(块、内存块)
  - 分区的地址是连续的

- 操作系统需要维护的数据结构
  - 所有进程的已分配分区
  - 空闲分区(Empty-blocks)⇒进程结束
- 动态分区分配策略
  - 最先匹配(First-fit)
  - 最佳匹配(Best-fit)
  - 最差匹配(Worst-fit)

连续内存分配

- 连续内存分配
- 动态分区分配策略
- 碎片整理
- 伙伴系统

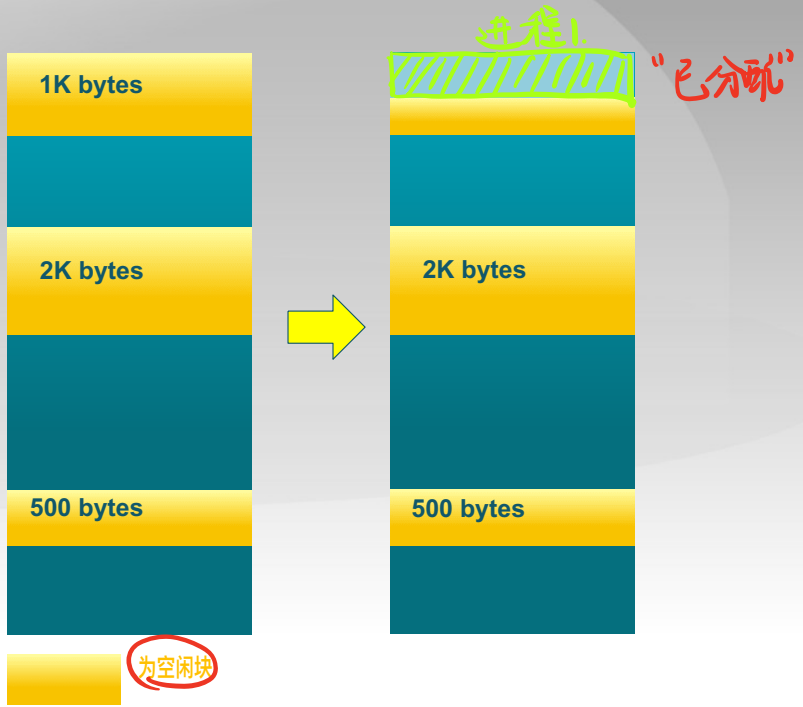
最先匹配(First Fit Allocation)策略

思路:

分配n
个字节，使用第一个可用的空间比n大的空闲块。

示例:

分配400字节，使用第一个1KB的空闲块。



最先匹配(First Fit Allocation)策略

- 原理 & 实现
  - 空闲分区列表按地址顺序排序
  - 分配过程时，搜索一个合适的分区(第一个)
  - 释放分区时，检查是否可与临近的空闲分区合并
- 优点
  - 简单
  - 在高地址空间有大块的空闲分区 不会往后找，可以找到大块的分配区.
- 缺点
  - 外部碎片
  - 分配大块时较慢

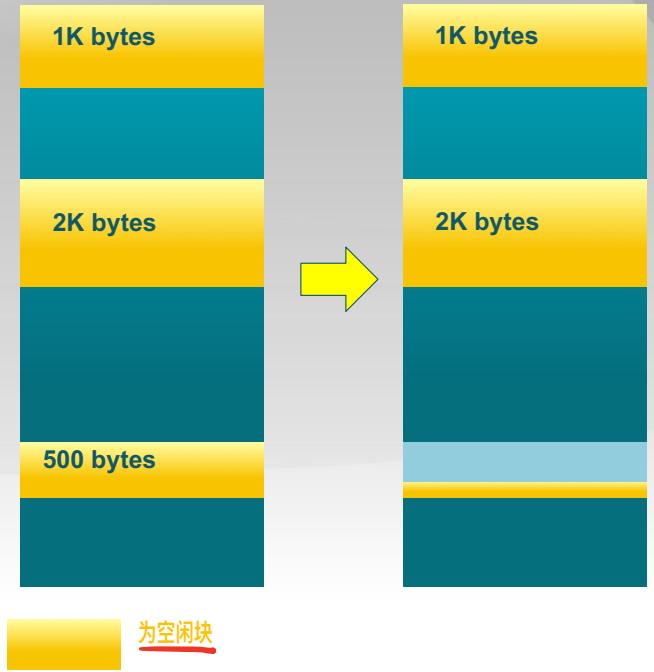
最佳匹配(Best Fit Allocation)策略

思路:

分配n字节分区时，查找并使用不小于n的最小空闲分区

示例:

分配400字节，使用第3个空闲块(最小)



最佳匹配(Best Fit Allocation)策略

- 原理 & 实现
  - 空闲分区列表按照大小排序
  - 分配时，查找一个合适的分区
  - 释放时，查找并且合并临近的空闲分区（如果找到）
- 优点
  - 大部分分配的尺寸较小时，效果很好
    - 可避免大的空闲分区被拆分
    - 可减小外部碎片的大小
    - 相对简单
- 缺点
  - 外部碎片
  - 释放分区较慢
  - 容易产生很多无用的小碎片

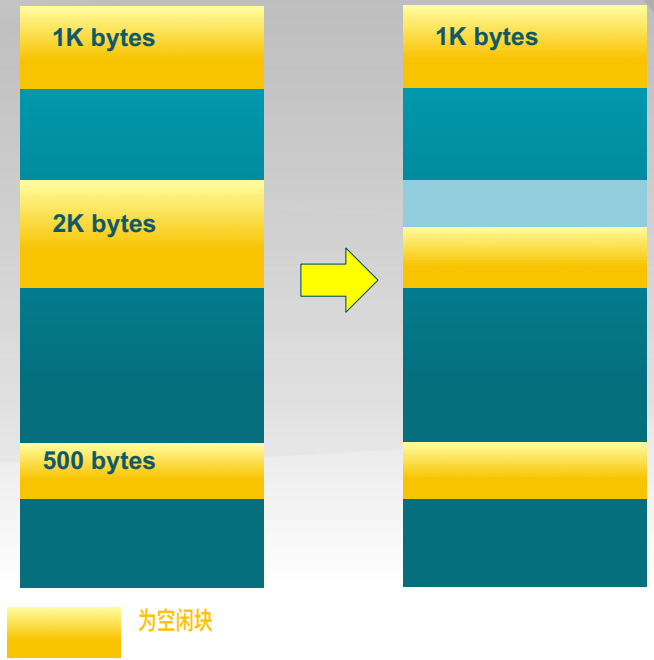
最差匹配(Worst Fit Allocation)策略

思路:

分配n字节，使用尺寸不小于n的最大空闲分区

示例:

分配400字节，使用第2个空闲块（最大）



最差匹配(Worst Fit Allocation)策略

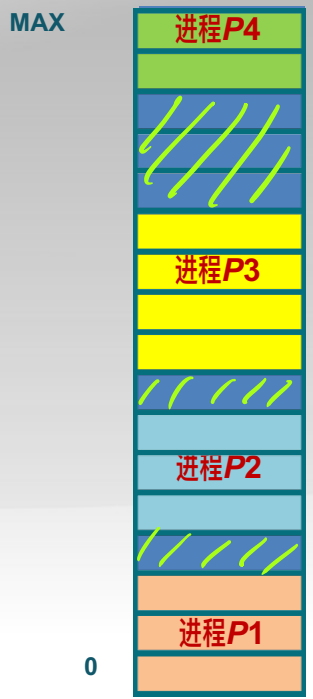
- 原理 & 实现
  - 空闲分区列表按由大到小排序 找最大的.
  - 分配时，选最大的分区
  - 释放时，检查是否可与临近的空闲分区合并，进行可能的合并，并调整空闲分区列表顺序
- 优点
  - 中等大小的分配较多时，效果最好
  - 避免出现太多的小碎片
- 缺点
  - 释放分区较慢
  - 外部碎片
  - 容易破坏大的空闲分区，因此后续难以分配大的分区

连续内存分配

- 连续内存分配
- 动态分区分配策略
- 碎片整理
- 伙伴系统

碎片整理：紧凑(compaction)

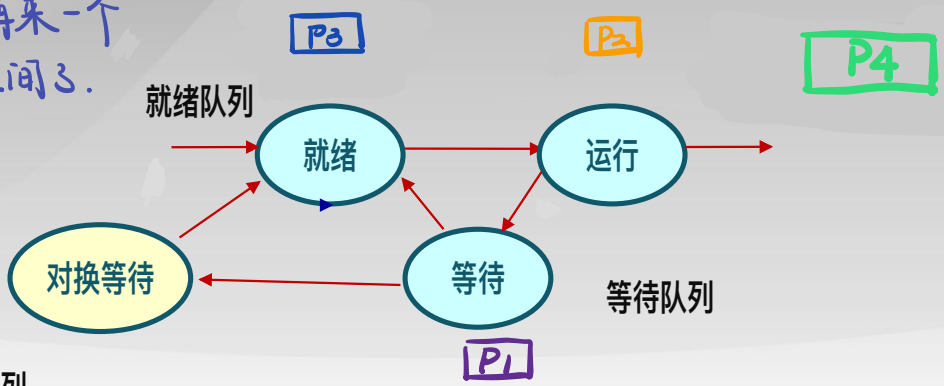
- 碎片整理
  - 通过调整进程占用的分区位置来减少或避免分区碎片
- 碎片紧凑
  - 通过移动分配给进程的内存分区，以合并外部碎片
    但若有绝对地址引用，则不可随意搬动！
  - 碎片紧凑的条件
    - 所有的应用程序可动态重定位
  - 需要解决的问题
    - 什么时候移动？等待时，运行时不可移动。
    - 开销

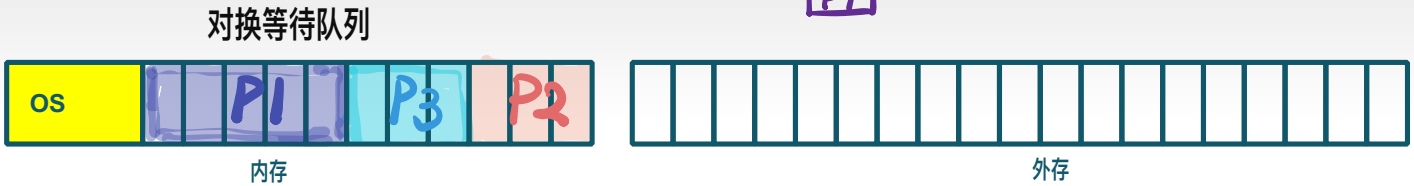


碎片整理：分区对换(Swapping in/out)

通过抢占并回收处于等待状态进程的分区，以增大可用内存空间

此时内存已满，再来一个P4就没有内存空间了.

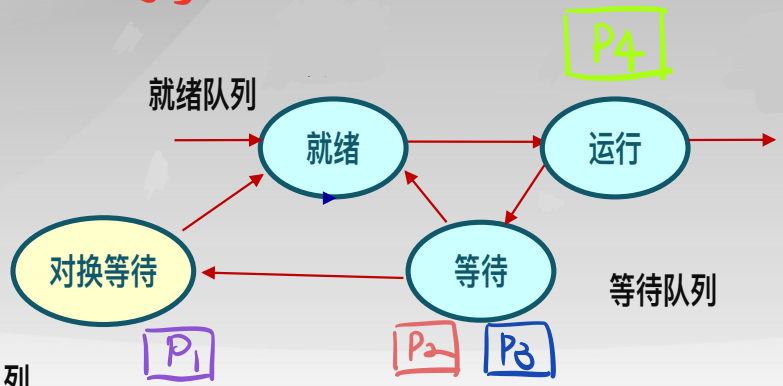




碎片整理：分区对换(Swapping in/out)

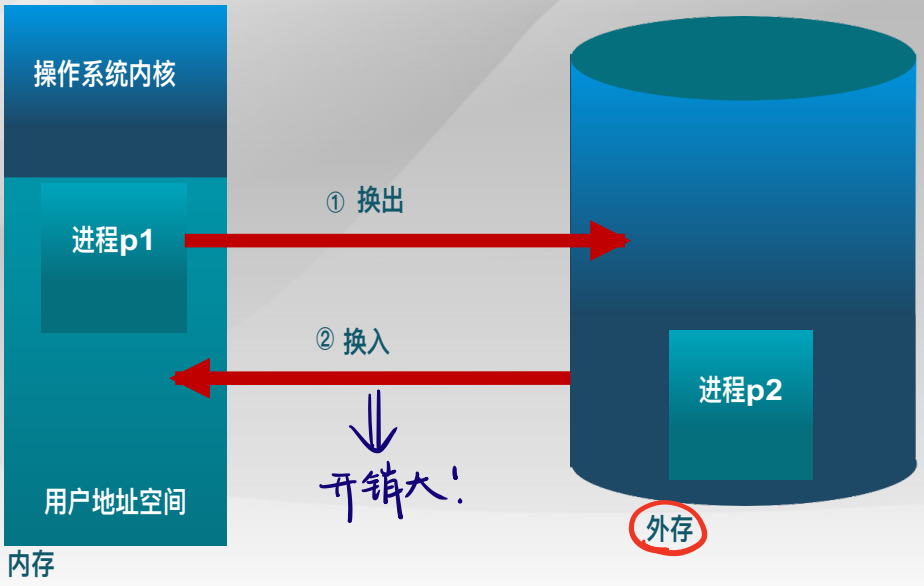
通过抢占并回收处于等待状态进程的分区，以增大可用内存空间

把等待进程在入外存.



对换等待队列

碎片整理：分区对换(Swapping in/out)



在最开始的操作系统中，就是内存中只有一个正在运行的进程，正在等待的进程都放在外存中.

- 需要解决的问题
  - 交换哪个（些）程序?

连续内存分配

- 连续内存分配
- 动态分区分配策略
- 碎片整理
- 伙伴系统

伙伴系统(Buddy System)

- 整个可分配的分区大小2^U 如果该块比我需要的2倍还大，就一直切一半；
- 需要的分区大小为$2^{U-1} < s \leq 2^U$时，把整个块分配给该进程；

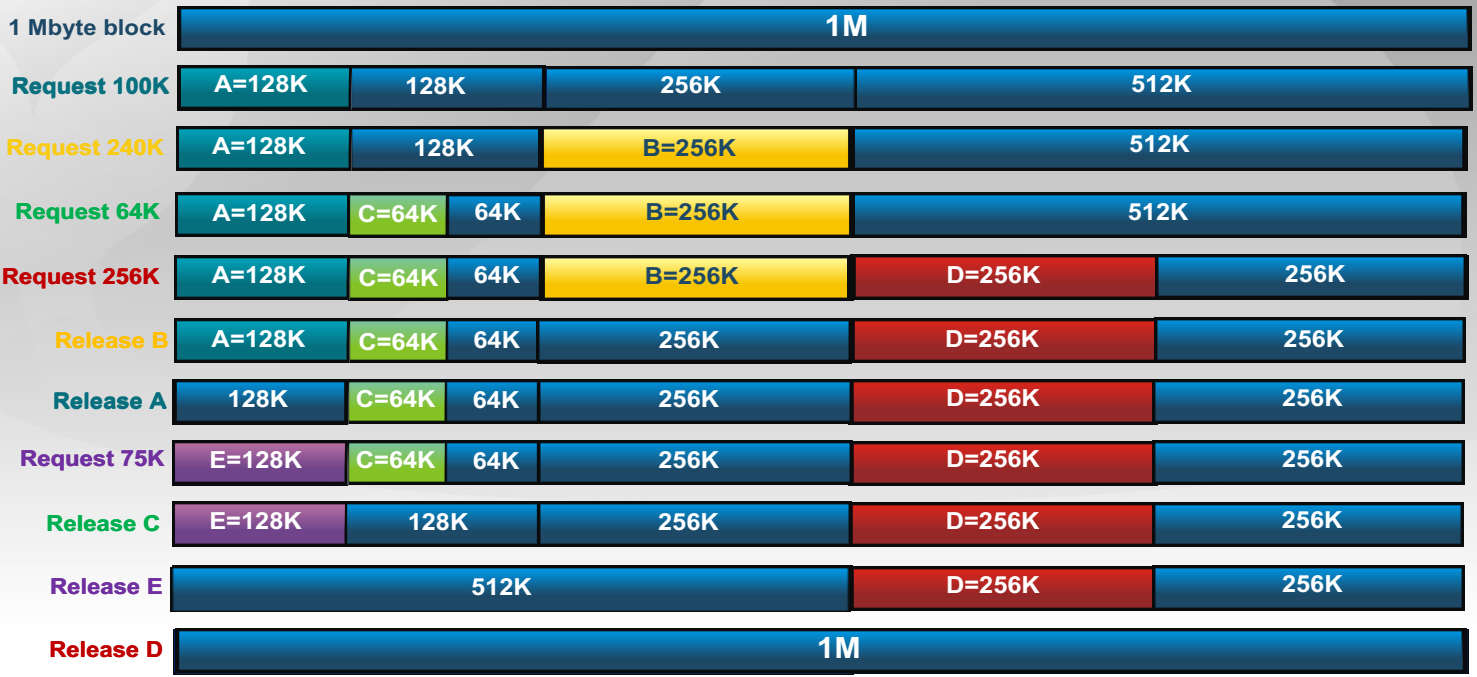
  如果该块比我需要的块大，但比我需要的2倍小，则把这块分给我

  - 如$s \leq 2^{i-1}$，将大小为2^i的当前空闲分区划分成两个大小为2^{i-1}的空闲分区
  - 重复划分过程，直到$2^{i-1} < s \leq 2^i$，并把一个空闲分区分配给该进程

伙伴系统的实现

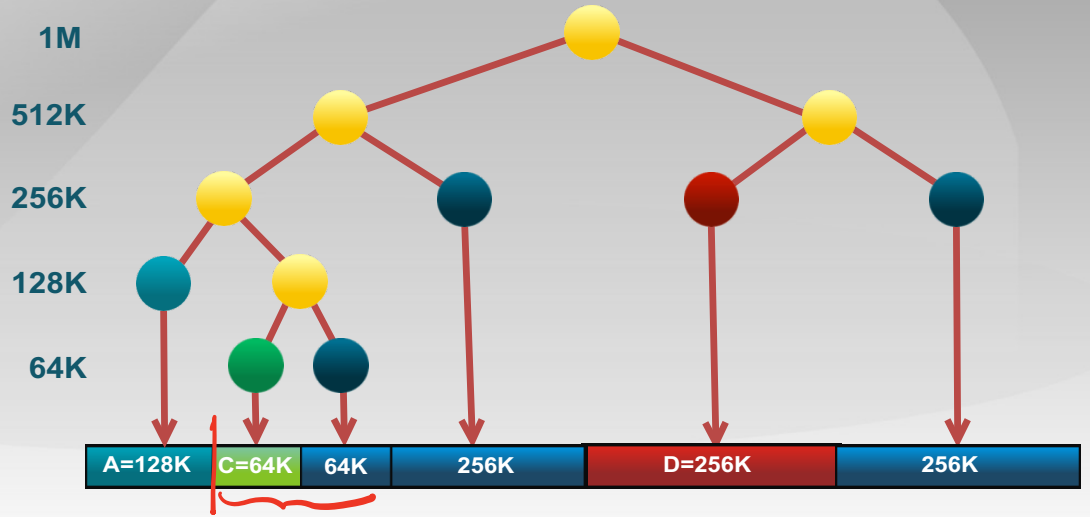
- 数据结构
  - 空闲块按大小和起始地址组织成二维数组
  - 初始状态：只有一个大小为2^U的空闲块
- 分配过程
  - 由小到大在空闲块数组中找最小的可用空闲块
  - 如空闲块过大，对可用空闲块进行二等分，直到得到合适的可用空闲块

伙伴系统中的内存分配


1 Mbyte block
1M
Request 100K
A=128K
128K
256K
512K
Request 240K
A=128K
128K
B=256K
512K
Request 64K
A=128K
C=64K
64K
B=256K
512K
Request 256K
A=128K
C=64K
64K
B=256K
D=256K
256K
Release B
A=128K
C=64K
64K
256K
D=256K
256K
Release A
128K
C=64K
64K
256K
D=256K
256K
Request 75K
E=128K
C=64K
64K
256K
D=256K
256K
Release C
E=128K
128K
256K
D=256K
256K
Release E
512K
D=256K
256K
Release D
1M

伙伴系统的实现

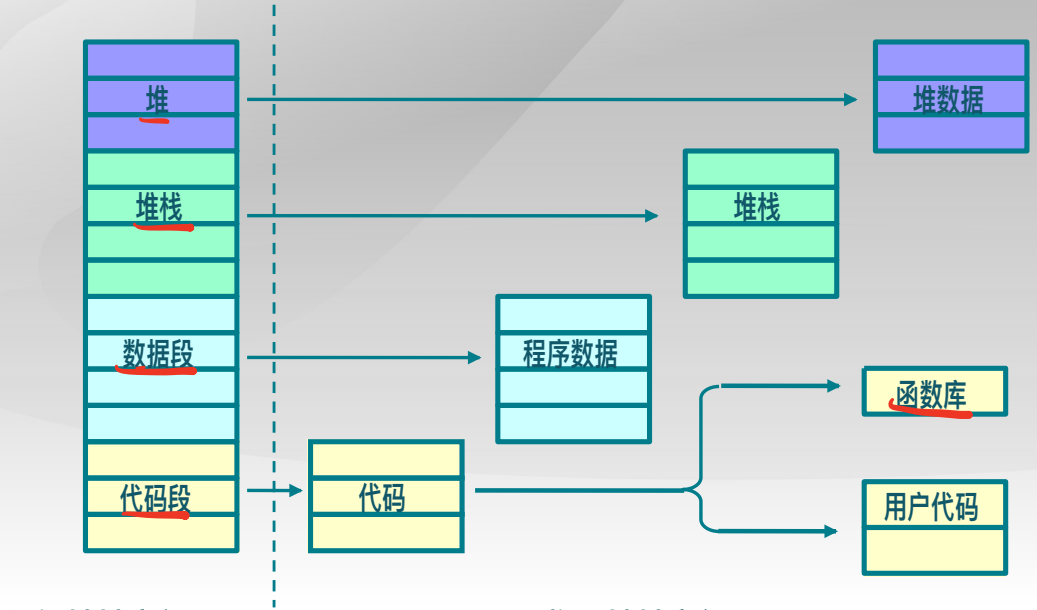
- 释放过程
  - 把释放的块放入空闲块数组
  - 合并满足合并条件的空闲块
- 合并条件
  - 大小相同2^i
  - 地址相邻
  - 低地址空闲块起始地址为2^{i+1}的位数



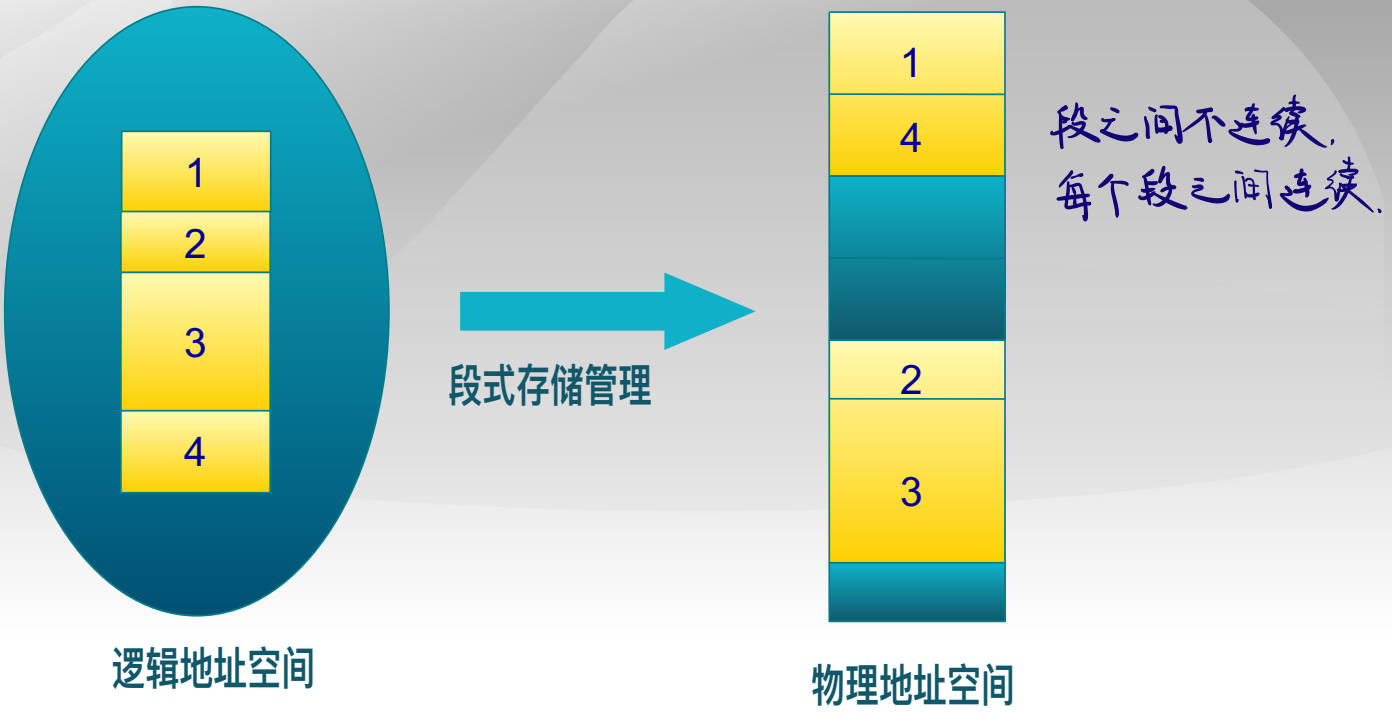
http://en.wikipedia.org/wiki/Buddy_memory_allocation

非连续内存分配
⇒段式内存分配

段式地址空间的不连续二维结构

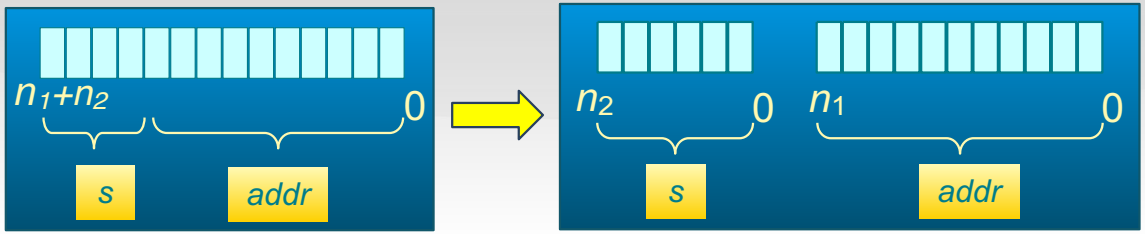


段地址空间的逻辑视图



段访问机制

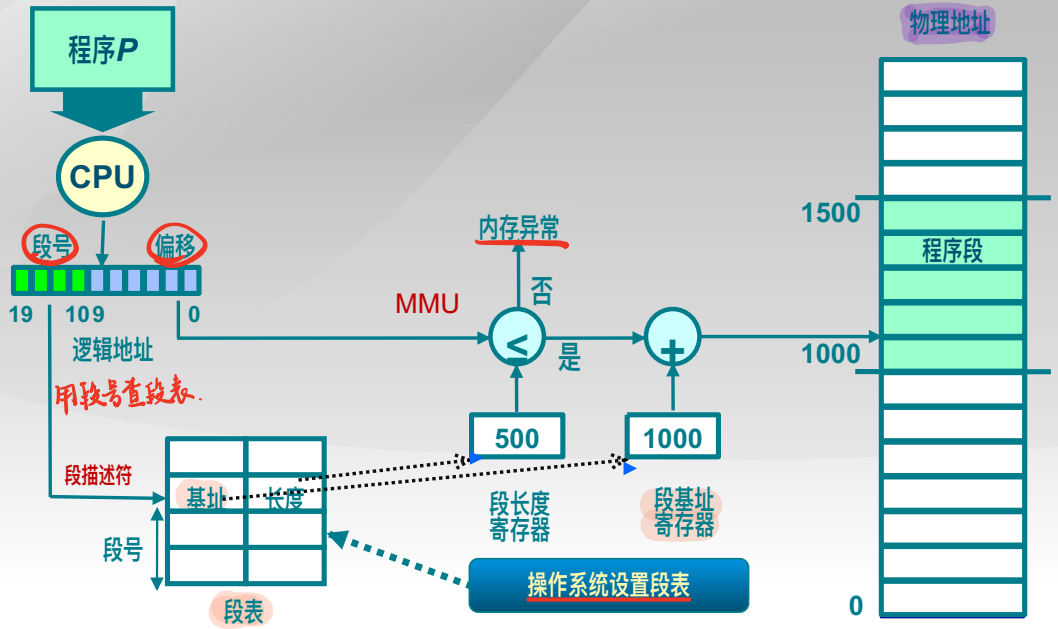
- 段的概念
  - 段表示访问方式和存储数据等属性相同的一段地址空间
  - 对应一个连续的内存“块”
  - 若干个段组成进程逻辑地址空间
- 段访问：逻辑地址由二元组(s, addr)表示
  - s — 段号
  - addr — 段内偏移



单地址实现方案

“段基址+段内偏移”实现方案

段访问的硬件实现

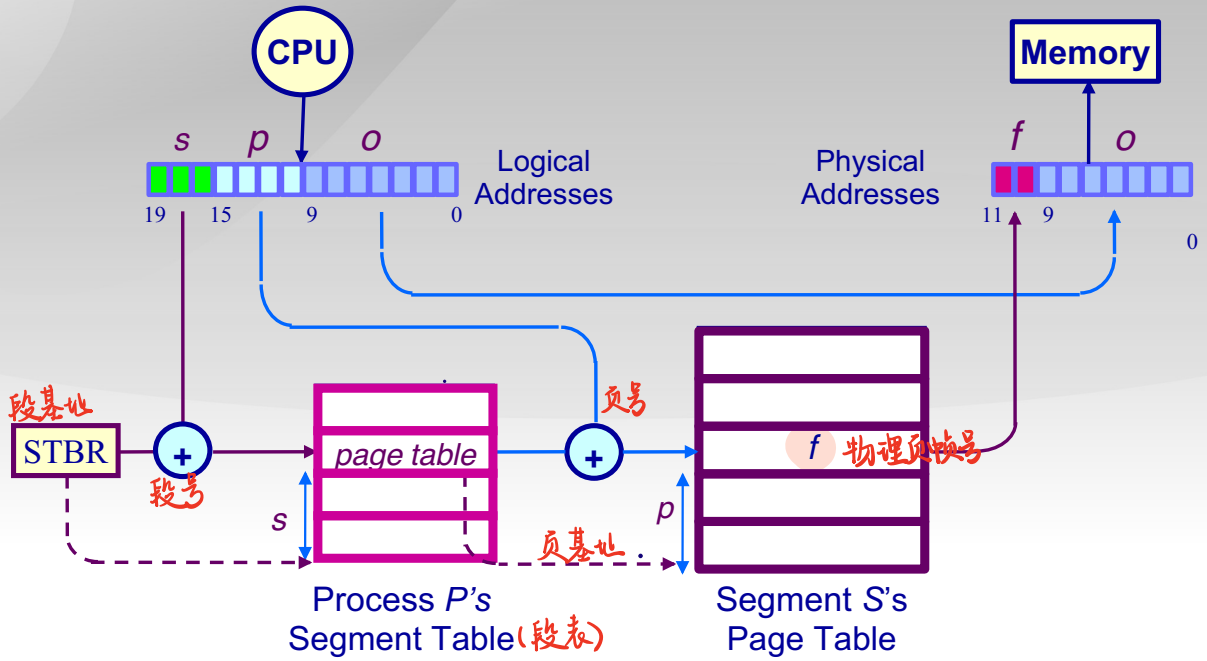

程序P
CPU
段号
偏移
19
109
0
逻辑地址
用段号查段表.
段描述符
基址
长度
段号
段表
MMU
内存异常
否
是
<
+
500
1000
段长度
寄存器
段基址
寄存器
操作系统设置段表
物理地址
1500
程序段
1000
0

段页式存储管理

- 段式存储管理支持方便的数据保护
- 页式存储管理的存储利用率高，且地址转换方便
- 如何组合段式存储和页式存储各自的优点?

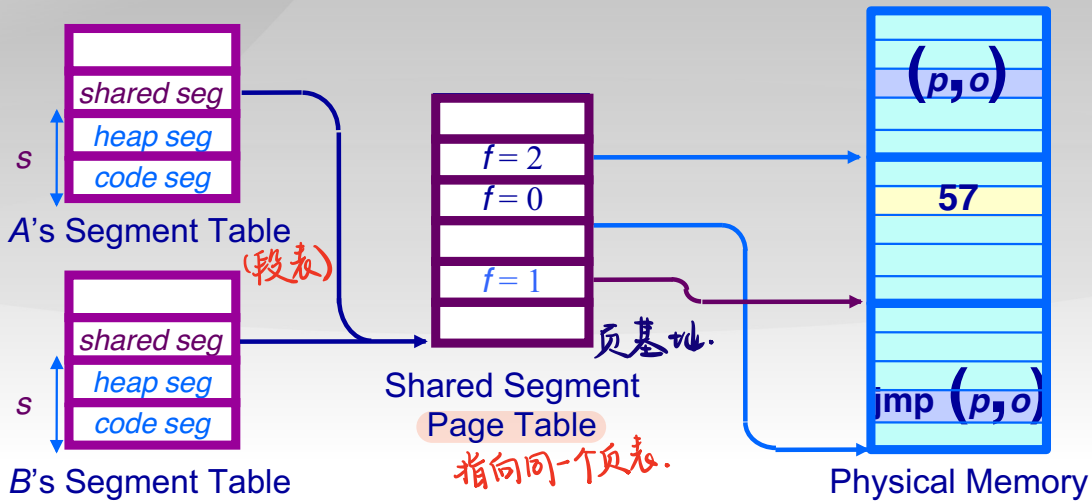
段页式存储管理

- 在页式存储的基础上再加一组指向页表的间接索引

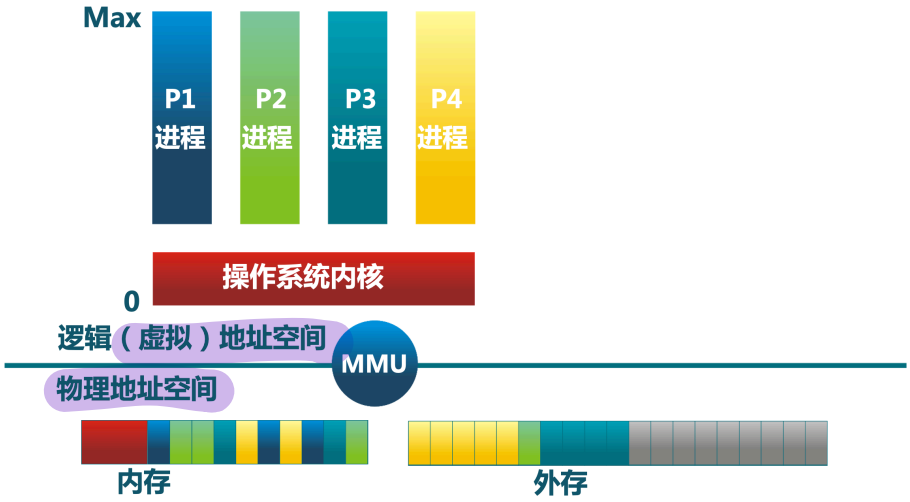


基于段页式存储的内存共享

- 通过段表项的共享，方便地实现该段所有页面的共享



回顾



- 通过页表来实现隔离与共享
  - 运行的应用程序之间的隔离与共享
  - 应用与内核之间的隔离与共享
  - 便于非连续内存管理
  - RISC-V Privileged Architecture Version 1.10 (RV32/64)
  - The RISC-V Reader 第 10.6 节