OPERATING SYSTEMS

Introduction to Mem. Management

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

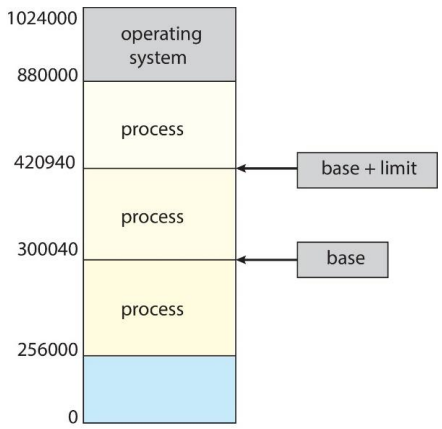
- 目录
  - 基本概念
  - 内存管理要求
    - 重新定位
    - 保护
    - 分享
    - 逻辑组织
    - 物理组织
  - 内存分区分配
    - 固定分区
    - 可变分区
    - 伙伴系统

Basic Concepts

- **背景**
  - 用户程序必须（从磁盘）放入内存并放入进程中才能运行。
    - 内存单元只能感测<address, read>请求流或<address, data, write>请求流。
    - 主存储器和寄存器是CPU可以直接访问的唯一存储器。
      - 寄存器在一个（或小于一个）CPU时钟周期内访问
      - 主存储器访问可能需要许多周期，从而导致暂停，因此
      - 高速缓存位于主内存和CPU寄存器之间。
  - 内存管理是由操作系统和硬件执行的任务，以满足主内存中的多个进程的需求。
    - 需要保护内存以确保正确操作。

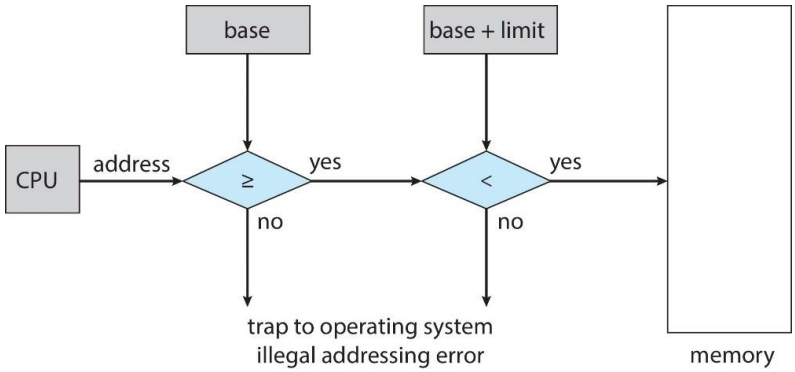
Basic Concepts

- **硬件地址保护**
  - 一对基址寄存器和界限寄存器定义进程的逻辑地址空间。



Basic Concepts

- **硬件地址保护**
  - CPU必须检查在用户模式下生成的每个内存访问，以确保该用户访问的内存地址在基址和界限之间。

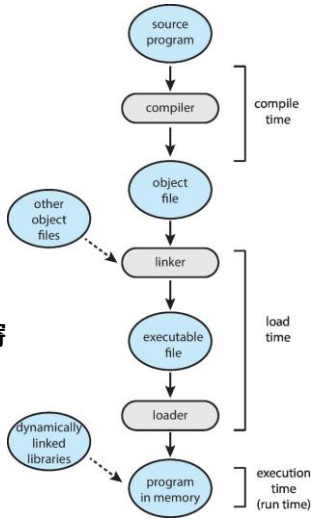


Basic Concepts

- **地址绑定**
  - 磁盘上的程序在准备进入内存执行时形成一个输入队列。
    - 如何分配用户进程的第一个物理地址？总是在0...00？
  - 此外，地址在程序生命周期的不同阶段以不同的方式表示。
    - 源代码地址通常是符号化的。
    - 编译后的代码地址绑定到可重新定位的地址。
      - 例如，“本模块开头的14字节”。
    - 链接器或加载程序将可重定位地址绑定到绝对地址。
      - 例如，14+71000=71014.
    - 每个绑定将一个地址空间映射到另一个地址空间。

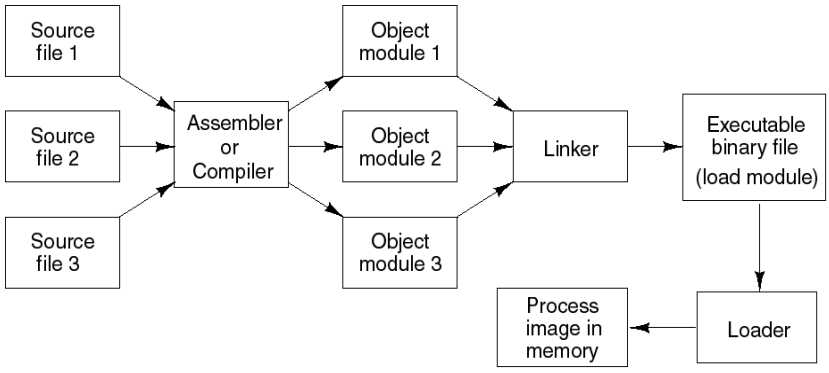
Basic Concepts

- **地址绑定**
  - 指令和数据到内存地址的地址绑定可以发生在用户程序多步处理的三个不同阶段。
    - **编译时：如果内存位置已知，则可以生成绝对代码；如果起始位置更改，则需要重新编译。**
    - **加载时：如果在编译时内存位置未知，则在加载时生成可重定位代码。**
    - **执行时：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。**
      - 地址映射需要硬件支持（例如，基寄存器和界限寄存器）。



Basic Concepts

- **地址绑定**
  - 用户程序的多步处理

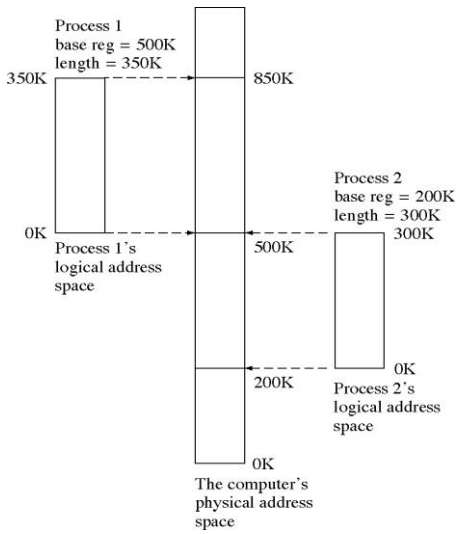


Basic Concepts

- **逻辑地址空间与物理地址空间**
  - 将逻辑地址空间绑定到单独物理地址空间的概念是正确内存管理的核心。
    - 逻辑地址–由CPU生成；也称为虚拟地址，是对独立于内存物理组织的内存位置的引用。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–内存单元看到的绝对地址，是主内存中的物理位置。物理地址空间是程序生成的所有物理地址的集合。
  - 编译时和加载时地址绑定方案中，逻辑地址和物理地址相同；执行时地址绑定方案中，逻辑地址和物理地址有所不同。
    - 编译器生成代码中，所有内存引用都是逻辑地址。
    - 相对地址是逻辑地址的一个示例，其中地址表示为相对于程序中某个已知点（例如，开始点）的位置。

Basic Concepts

- **逻辑地址空间与物理地址空间**
  - 逻辑和物理地址空间

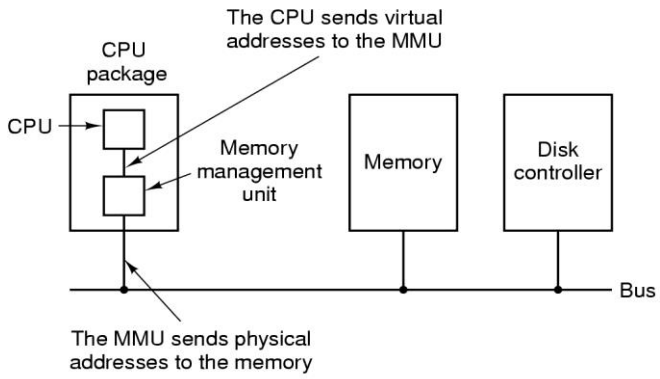


Basic Concepts

- **内存管理单元（MMU）**
  - 内存管理单元是在运行时将虚拟地址映射到物理地址的硬件设备。
  - 在MMU方案中，用户进程在发送到内存时生成的每个逻辑地址都会增加基址寄存器中的值
    - 基址寄存器现在称为重定位寄存器
    - 英特尔80x86上的MS-DOS使用了4个重定位寄存器
  - 用户程序只处理逻辑（虚拟）地址，它从未看到过真实的（物理）地址
    - 执行时绑定在引用内存中的位置时发生。
    - 逻辑地址绑定到物理地址。

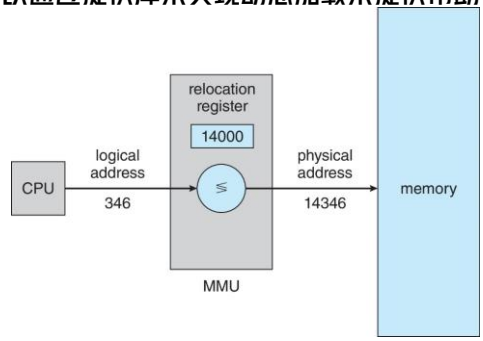
Basic Concepts

- **内存管理单元（MMU）**
  - CPU、MMU和内存



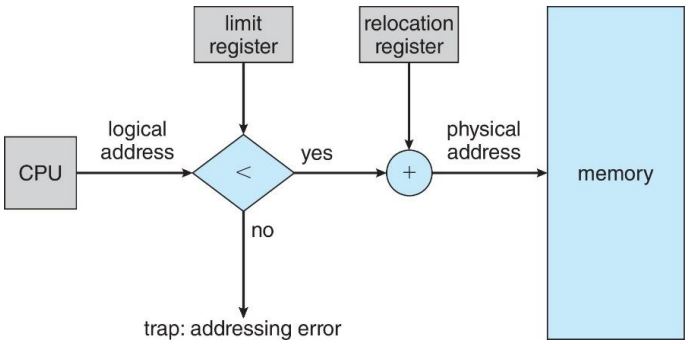
Basic Concepts

- **使用重定位寄存器的动态重定位**
  - 在调用例程之前，不会加载该例程。
    - 所有例程以可重定位加载格式保存在磁盘上
      - 不用的例程从不加载-更好的内存空间利用率
      - 当需要大量代码来处理不经常发生的情况时，此功能非常有用
  - 不需要操作系统的特别支持
    - 通过程序设计实现
    - OS可以通过提供库来实现动态加载来提供帮助



Basic Concepts

- **对重定位和界限寄存器的硬件支持**
  - 基址寄存器现在称为重定位寄存器



Basic Concepts 15 / 61

地址的硬件转换动态

- 当一个进程被分配到运行状态时，一个重定位/基址寄存器将加载该进程的起始物理地址。
- 限制/界限寄存器将加载进程的结束物理地址。
- 当遇到相对地址时，将其与基址寄存器的内容相加，以获得与限制/界限寄存器的内容相比较的物理地址。
- 这提供了硬件保护：每个进程只能访问其进程映像中的内存。

Basic Concepts 16 / 61

动态链接

- *静态链接*–加载程序将系统库和程序代码组合到二进制程序映像中
- *动态链接*–链接延迟到执行时进行。
  - 一小段代码（存根）用于定位适当的内存驻留库例程。
  - 存根将自身替换为例程的地址，并执行例程。
  - 操作系统检查例程是否在进程的内存地址中。
    - 如果没有，将其添加到地址空间
  - 动态链接对于共享/公共库特别有用–需要全面的操作系统支持
    - 例如标准的C语言库
- 考虑修补系统库的适用性
  - 可能需要进行版本控制

Basic Concepts

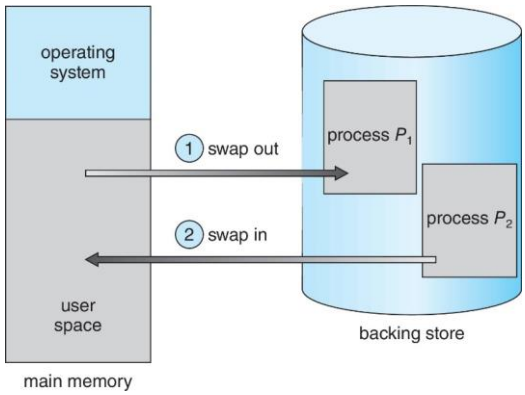
- **交换**
  - 进程可以暂时从内存中交换到备份存储，然后带回内存继续执行。
    - 通过支持交换，进程的总内存空间可以超过实际物理内存大小
  - *备份存储*–快速大磁盘，足以容纳所有用户的所有内存映像副本；必须提供对这些内存映像的直接访问。
  - *换出、换入*–交换变量，用于基于优先级的调度算法；低优先级进程被调出，以便可以加载和执行高优先级进程。
  - 交换时间的主要部分是传输时间。
    - 总传输时间与交换的内存量成正比。
  - 系统维护一个就绪队列，其中包含磁盘上具有内存映像的就绪运行进程。

Basic Concepts

- **交换**
  - 交换出的进程是否需要像以前一样交换回相同的物理地址？
    - 这取决于地址绑定方法。
      - 考虑到关联进程内存空间的挂起I/O
  - 许多系统（如UNIX、Linux和Windows）上都有经过修改的交换版本
    - 交换通常被禁用
    - 如果分配的内存量超过阈值，则启动
    - 一旦内存需求降低到阈值以下，将再次禁用

Basic Concepts

- **交换**
  - 交换的示意图



Basic Concepts

- **交换**
  - 上下文切换时间，包括交换。
    - 如果要获得CPU的下一个进程（目标进程）不在内存中，需要交换出一些进程，然后把目标进程交换入内存中。
      - 上下文切换时间可能非常长。
    - 考虑100MB进程交换到硬盘，传输速率为50Mb/s：
      - 交换时间为100/50（秒）= 2000（毫秒）
      - 加上同等规模的换入流程
      - 上下文切换交换组件的总时间为4000ms（4秒）
    - 如果我们知道实际需要使用多少内存，可减少交换内存的大小以减少交换时间。
      - 通知OS内存使用的系统调用

        request_memory()
        release_memory()

Basic Concepts

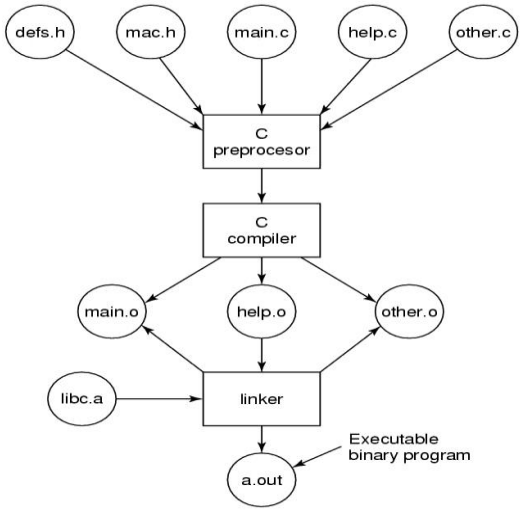
- **交换**
  - 上下文切换时间，包括交换。
    - 交换的其他约束条件
      - 挂起的I/O–无法调出，因为I/O将发生在错误的进程上。
      - 或者总是将I/O传输到内核空间，再传输到I/O设备
        - 称为双缓冲，增加了开销
    - 现代操作系统中不使用标准交换。
      - 但常用修改后的版本
        - 仅在可用内存极低时交换

Basic Concepts

- **交换**
  - 在移动系统上交换
    - 通常不受支持
      - 基于闪存(Flash)
        - 空间较小
        - 有限的写入周期数
        - 移动平台上闪存和CPU之间的吞吐量低
    - 如果内存不足，则使用其他方法释放内存。
      - iOS要求应用自动放弃分配的内存
        - 丢弃只读数据，需要时再从闪存重新加载
        - 释放失败可能导致进程终止
      - 如果可用内存不足，Android会终止应用程序，但首先会将应用程序状态写入闪存以快速重启。
      - 这两个操作系统都支持分页（稍后将讨论）

Basic Concepts

- **一个C编译示例**



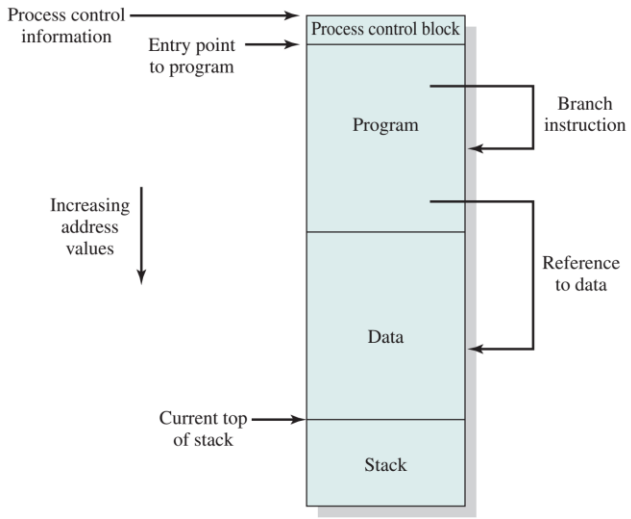
Basic Concepts

- **一个C编译示例**
  - public公共名称可由其他对象模块使用。
  - external外部名称在其他对象模块中定义。
    - 包括将这些名称作为操作数的指令列表
  - 重定位字典
    - 具有操作数为地址的指令列表（因为它们是可重定位的）
  - 只有代码和数据将加载到物理内存中。
    - 其余部分由链接器使用，然后移除。
  - 堆栈仅在加载时分配。

模块结束
重定位字典
数据
机器代码
external外部名称表
public公共名称表
模块标识id

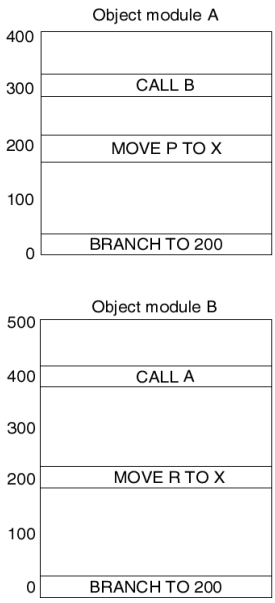
Basic Concepts

- 一个C编译示例
  - 处理进程的需求



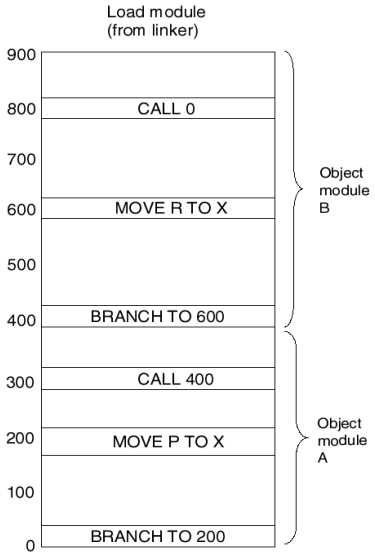
Basic Concepts

- 一个C编译示例
  - 最初，每个对象模块都有自己的地址空间。
  - 所有地址都相对于模块的开头。



Basic Concepts

- **一个C编译示例**
  - 静态链接
    - 链接器使用对象模块中的表将模块链接到单个线性可寻址空间。
    - 新地址是相对于加载模块开头的地址。

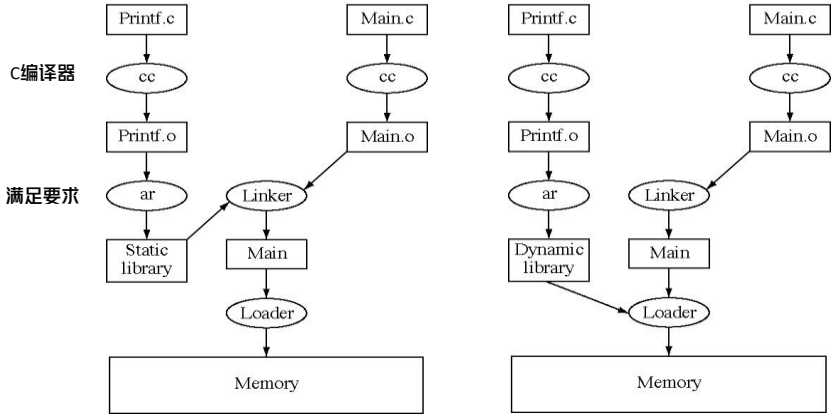


Basic Concepts

- **一个C编译示例**
  - 动态链接
    - 一些外部模块的链接在创建加载模块（可执行文件）后完成。
    - 加载时动态链接：
      - 加载模块包含对加载时解析的外部模块的引用。
    - 运行时动态链接：
      - 调用外部模块中定义的过程时，解析对外部模块的引用。
      - 从不加载未使用的过程。
      - 进程启动得更快。

Basic Concepts

- **一个C编译示例**
  - 静态链接与动态链接



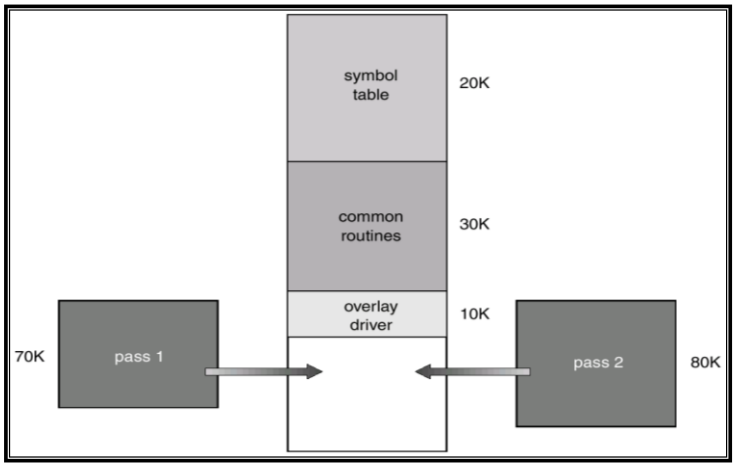
(a) 静态链接　　(b) 动态链接

Basic Concepts

- **程序大小 vs. 内存大小**
  - 当程序大小大于分配给它的内存或分区（存在的或可以分配的）总量时，该怎么办？
  - 物理内存管理有两种基本的解决方案：
    - 覆盖层
    - 动态链接（库，即DLL）
  - 覆盖层
    - 在任意给定阶段/时间里，在内存中只保留程序中需要的部分指令和数据。
    - 覆盖只能用于适合此模型的程序，例如， 像编译器一样的多道(multi-pass)程序。
    - 覆盖层由程序员设计/实现
      - 需要覆盖层驱动程序
    - 不需要操作系统的特别支持
      - 但覆盖层结构的程序设计比较复杂。

Basic Concepts

- **程序大小 vs. 内存大小**
  - 覆盖层



Two-Pass二道汇编程序的覆盖

Basic Concepts

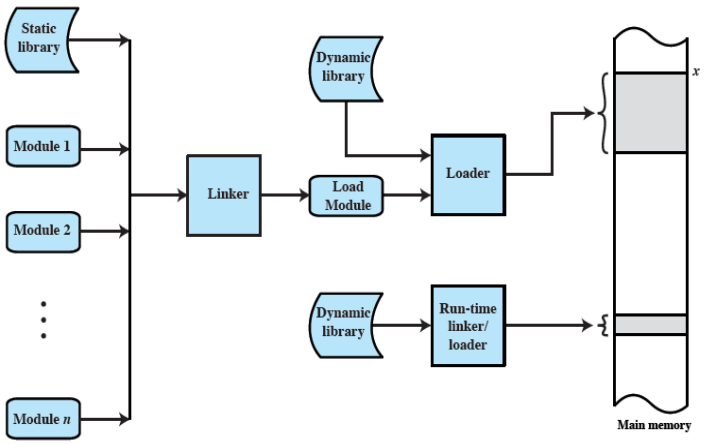
- **程序大小 vs. 内存大小**
  - 动态链接
    - 当需要大量代码来处理不经常发生的情况时，动态链接非常有用。
      - 除非调用例程，否则不加载例程。
    - 更好的内存空间利用率
      - 从不加载未使用的例程
    - 不需要操作系统提供太多支持
      - 通过程序设计实现。

Basic Concepts

- **程序大小 vs. 内存大小**
  - 动态链接的优势
    - 无需修改的可执行文件可以使用其他版本的外部模块。
    - 多个进程可链接到同一个外部模块。
      - 节省磁盘空间
    - 同一个外部模块只需在主存中加载一次。
      - 进程可以共享代码并节省内存。
    - 示例：
      - Windows的动态链接库: .DLL文件
      - Unix/Linux的共享库: .SO文件

Basic Concepts

- **程序大小 vs. 内存大小**
  - 动态链接/加载场景



Memory Management Requirements

- **内存管理要求**
  - 需要有效地分配内存，以便将尽可能多的进程打包到内存中，避免所有进程都在等待I/O，而CPU处于空闲状态的情况。
  - 需要以下方面的额外支持：
    - 重新定位
    - 保护
    - 分享
    - 逻辑组织
    - 物理组织

Memory Management Requirements

- **内存管理要求**
  - 重新定位
    - 当程序执行时，程序员无法知道程序将放在内存中的什么位置。
    - 由于交换/压实，进程可能（通常）在主内存中重新定位：
      - 交换使操作系统具有更大的就绪执行进程池。
      - 压实(合并，压紧) 使操作系统具有更大的连续内存来放置程序。
  - 保护
    - 未经许可，进程不能引用另一进程中的内存位置。
    - 在编译/加载时无法检查程序中的地址，因为程序可能会被重新定位。
    - 硬件必须在执行时检查地址引用。

Memory Management Requirements

- **内存管理要求**
  - 分享
    - 必须允许多个进程在不影响保护的情况下访问主内存的公共部分。
      - 最好允许每个进程访问程序的同一副本，而不是拥有自己的单独副本。
      - 协作进程可能需要共享对同一数据结构的访问。
  - 逻辑组织
    - 用户编写的程序中具有不同特征的模块。
      - 指令模块仅可执行。
      - 数据模块为只读或读/写。
      - 有些模块是私有的，有些是公共的。
    - 为了有效地处理用户程序，操作系统和硬件应支持基本形式的模块，以提供所需的保护和共享。

Memory Management Requirements

- **内存管理要求**
  - 物理组织
    - 外部存储器是程序和数据的长期存储器，而主存储器保存当前正在使用的程序和数据。
    - 在内存层次结构的这两个级别之间移动信息是内存管理的主要关注点。
      - 将此责任留给应用程序程序员是非常低效的。

Memory Partitioning

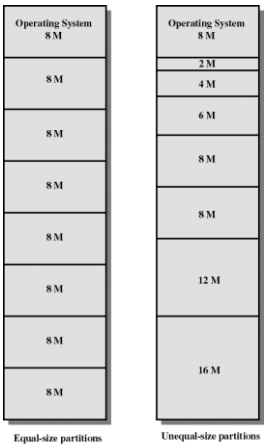
- **连续分配**
  - 在连续存储分配中，执行中的进程必须完全加载到主内存中（如果不使用覆盖）
  - 主存通常分为两个（split）或多个（division）分区：
    - 常驻操作系统，通常用中断向量保存在低内存分区中
    - 然后将用户进程保存在高内存分区中
  - 重定位寄存器用于保护用户进程彼此不受影响，以及防止更改操作系统代码和数据。
    - 基址寄存器包含最小物理地址的值。
    - 界限寄存器包含逻辑地址范围–每个逻辑地址必须小于界限寄存器。
    - MMU动态映射逻辑地址。

Memory Partitioning

- **物理内存管理技术**
  - 尽管以下简单（基本）内存管理技术没有在现代操作系统中使用，但它们为以后正确讨论虚拟内存奠定了基础。
    - 固定/静态分区分配
    - 可变/动态分区分配
    - 简单页式分配
    - 简单段式分配

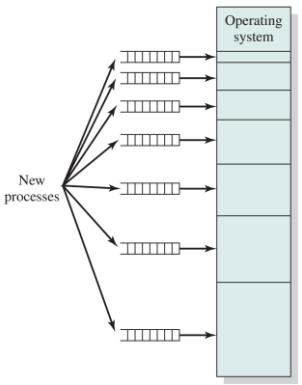
Fixed Partitioning

- **固定分区**
  - 在固定分区方案中，主存被划分为一组称为分区的非重叠内存区域
  - 固定分区（也称为静态分区）的大小可以相等或不等。
  - 程序分配后，分区内的剩余空间称为内部碎片。



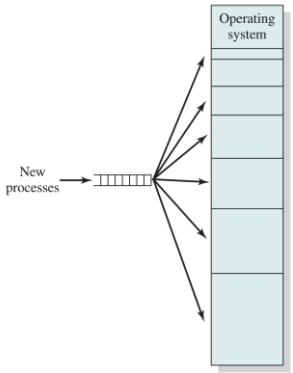
Fixed Partitioning

- **带分区的分配算法**
  - 大小不等的分区，使用多个队列：
    - 将每个进程分配给适合它的最小分区
    - 每个分区大小都有一个队列
    - 尝试最小化内部碎片
    - 问题：某些队列可能为空，而某些队列可能满载。



Fixed Partitioning

- **带分区的分配算法**
  - 大小不等的分区，使用单个队列：
    - 将进程加载到内存时，将选择容纳该进程的最小可用分区
    - 以牺牲内部碎片为代价提高多道程序的程度



Fixed Partitioning

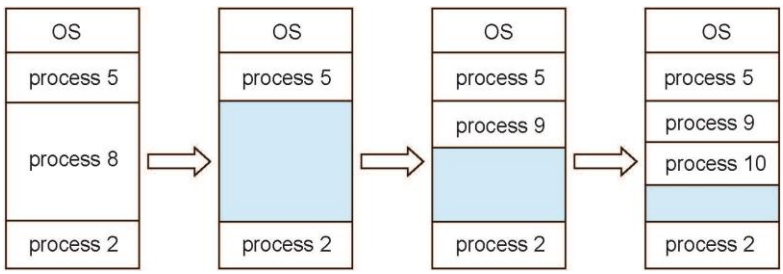
- **固定分区分配的动态**
  - 任何大小小于或等于分区大小的进程都可以加载到分区中。
  - 如果所有分区都被占用，操作系统可以将进程从分区中交换出去。
  - 程序可能太大，无法装入分区。在这种情况下，程序员必须用覆盖层设计程序。

Fixed Partitioning

- **关于固定分区的评论**
  - 内存使用效率低下。任何程序，无论多么小，都会占用整个分区。这可能导致严重的内部碎片。
  - 大小不等的分区减轻了这些问题，但它们仍然。。。
  - 在IBM早期的OS/MFT（固定任务数的多道程序）中使用了相同大小的分区。

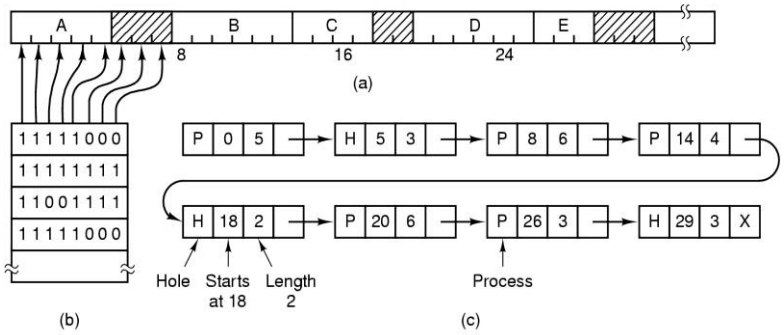
Variable Partitioning

- **可变分区**
  - 多道程序的程度受分区数量的限制。
  - 为提高效率而改变分区大小（根据给定进程的需要调整大小）。
  - 孔–空闲分区; 各种大小的孔散布在内存中。
  - 当一个进程到达时，从一个足以容纳它的孔中分配内存。
  - 进程退出释放其分区，相邻的空闲分区合并。
  - 操作系统维护以下信息：a)已分配的分区。b) 空闲分区（孔）。



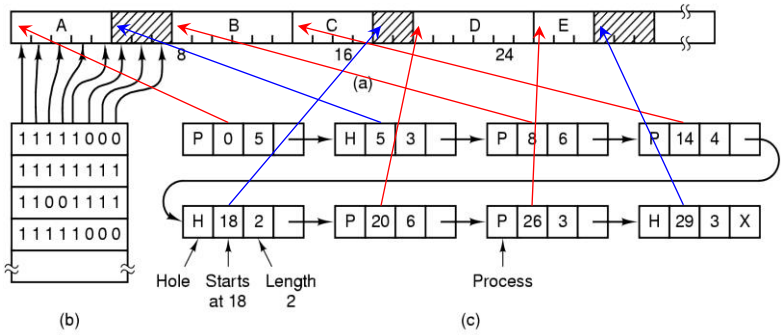
Variable Partitioning

- **管理已分配和空闲分区**
  - 示例：具有5个进程A、B、C、D、E和3个孔的内存。
    - 刻度线显示内存分配单元。
    - 阴影区域（位图中的0）是空闲的。



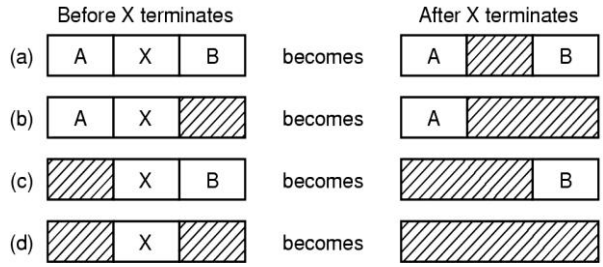
Variable Partitioning

- **管理已分配和空闲分区**
  - 示例：具有5个进程A、B、C、D、E和3个孔的内存。
    - 刻度线显示内存分配单元。
    - 阴影区域（位图中的0）是空闲的。



Variable Partitioning

- **管理已分配和空闲分区**
  - 示例：空闲分区合并
    - 阴影区域是空闲的。



Variable Partitioning

- **碎片**
  - 内部碎片
    - 分配的内存可能略大于请求的内存；其差即未被使用的内存，因在分区内部无法分配给其他进程使用。
  - 外部碎片
    - 可用分区的总大小满足请求，但可用内存不是连续的。
  - 首次适应分析表明，给定分配的N个块，0.5N个块因碎片而浪费。
    - 例如，1/3可能没有使用->50%规则。
  - 通过压缩减少外部碎片。
    - 压缩(compaction)意味着对内存内容进行整理，使得所有可用内存合并到一个（或多个）大块中。
    - 只有重新定位是动态的，并且是在执行时进行的，才有可能进行压缩。
    - I/O问题：
      - 当作业涉及I/O时，将其锁定在内存中。
      - 仅对于操作系统缓冲区执行I/O操作。

Variable Partitioning

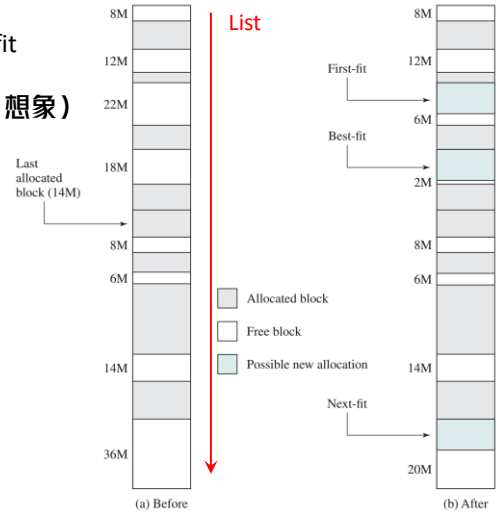
- **关于可变分区的评论**
  - 分区的长度和数量是可变的。
  - 每个进程都被精确地分配了所需的内存。
  - 最终在主存储器中形成孔。这可能导致外部碎片。
  - 压缩用于移动进程，使其相邻；从而所有可用内存聚合在一个块中
  - 在IBM的OS/MVT（可变任务数的多道程序）中使用。

Variable Partitioning

- **动态存储分配问题**
  - 为满足对空闲孔列表中某些大小的内存请求，可以应用4种基本分配算法：
    - 首次适应(First-fit): 分配搜索到的第一个足够大的孔。
    - 循环首次适应(Next-fit): 与首次适应的逻辑相同，但搜索总是从上一个分配的孔开始（需要保持指向该孔的指针），以循环方式进行。
    - 最佳适应(Best-fit): 分配足够大的最小孔；必须搜索整个列表，除非按大小排序。它将产生最小的剩余孔。
    - 最坏适应(Worst-fit): 分配最大孔；也必须搜索整个列表。它将产生最大的剩余孔。
  - 就速度和存储利用率而言， 首次适应和最佳适应要优于最坏适应。

Variable Partitioning

- **动态存储分配问题**
  - 实例
    - 通过以下算法选择哪个空闲块来分配16MB的进程。
      - 首次适应First-fit
      - 循环首次适应Next-fit
      - 最佳适合Best-fit
      - 最坏适合Worst-fit（想象）



Variable Partitioning

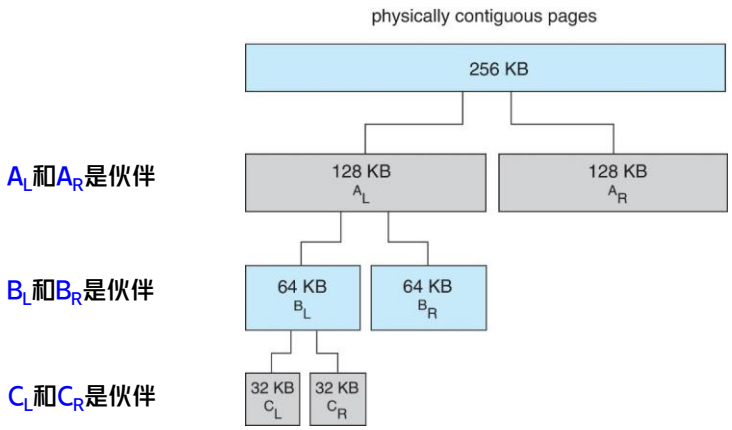
- **动态存储分配问题**
  - 关于布局算法的评论
    - 首次适应有利于在自由孔列表的开始附近进行分配。它往往比循环首次适应产生更少的碎片。
    - 循环首次适应通常会导致在内存末尾分配最大的块。
    - 最佳适应搜索最小块，留下的碎片尽可能小——
      - 主内存很快会形成太小的孔，无法容纳任何进程
      - 压缩通常需要更频繁地进行
    - 就速度和存储利用率而言，最坏适应最差。

Buddy System

- **伙伴系统分配**
  - 伙伴(Buddy)系统(Harry Markowitz, 1963)是一种合理的折衷方案，可以克服固定和可变分区方案的缺点。
    - 内存分配使用2的幂分配；满足以2的幂为单位的内存请求。
    - 请求的大小如非2的幂，则向上取满足请求的2的最小次幂分配。例如，一个11KB的请求用一个16KB的分区来满足。
  - 内存块的大小为2^k，其中 $l \le k \le u$，且：
    - 可分配块的最小大小为2^l
    - 可分配块的最大大小为2^u（通常是可用的整个内存）
  - 该方法的修改版的被应用于Unix SVR4/Linux的内核内存分配中。

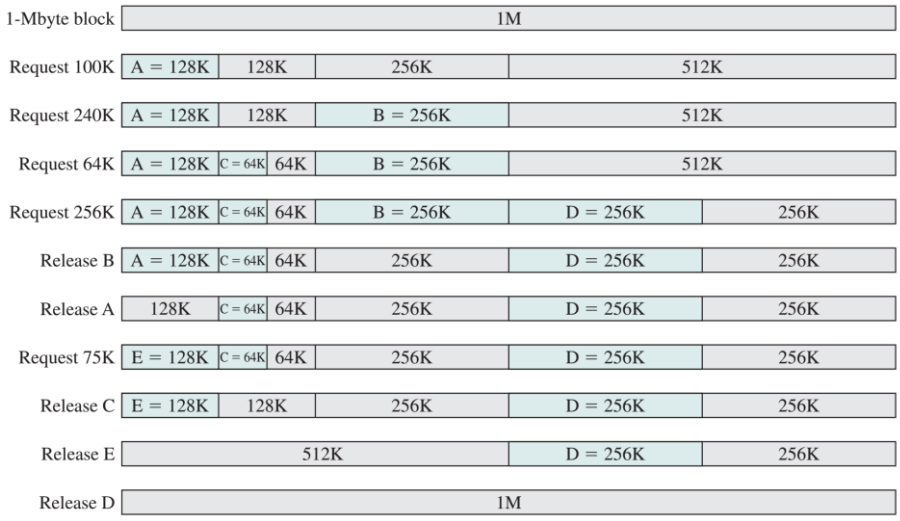
Buddy System

- **伙伴系统分配**
  - 伙伴系统中的伙伴



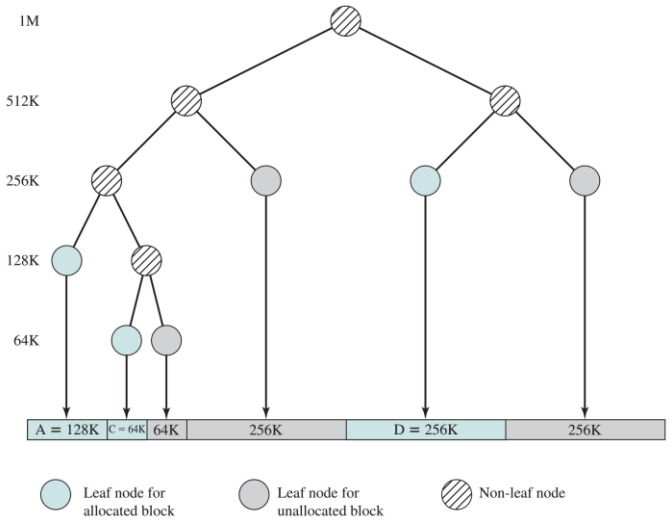
Buddy System

- **伙伴系统分配**
  - 伙伴系统的例子



Buddy System

- **伙伴系统分配**
  - 伙伴系统的树表示



Buddy System

- **伙伴系统分配**
  - 伙伴系统的动态
    - 假设我们从大小为 2^u 的整个块开始。
    - 当请求的大小为 S, $S \le 2^u$：
      - 如果 $2^{u-1} < S \le 2^u$, 则将大小为 2^u 的整个块分配给S.
      - 否则，将这个 2^u 大小的块分成两个伙伴，每个伙伴的大小为 2^{u-1} 。
      - 如果 $2^{u-2} < S \le 2^{u-1}$, 则将2个2^{u-1}大小的伙伴中的一个分配给S.
      - 否则，两个伙伴中的一个会再次分裂。
    - 重复此过程，直到生成大于或等于 S 的最小块。
    - 每当两个伙伴都未分配时，他们就会合并成更大的一块。

Buddy System

- **伙伴系统分配**
  - 伙伴系统的动态
    - 操作系统维护多个孔列表：
      - i-list是尺寸为2^i的孔列表。
      - 每当i-list中出现一对伙伴时，它们就会从该列表中删除，并合并到(i+1)-list列表中的一个孔中。
    - 要求分配大小S，使得
      $2^{i-1} < S \le 2^i$
      - 首先检查i-list
      - 如果i-list为空，则检查 (i+1)-list.....（然后呢？）

Buddy System

- **伙伴系统分配**
  - 伙伴系统的评论
    - 当好友系统使用的内存大小M为2的幂时，效率最高：
      - $M = 2^u$ (字节)，其中u为整数。
      - 那么每个块的大小是2的幂。
      - 最小的块大小为1。
    - 平均而言，内部碎片化率为25%
      - 每个内存块的占用率至少为50%
    - 程序不会在内存中移动：
      - 简化内存管理。