操作系统原理与设计

第8章 Main Memory1

陈香兰

中国科学技术大学计算机学院

2009年11月

提纲

Background

- Storage hierarchy
- memory protection
- Program execution, loading & linking

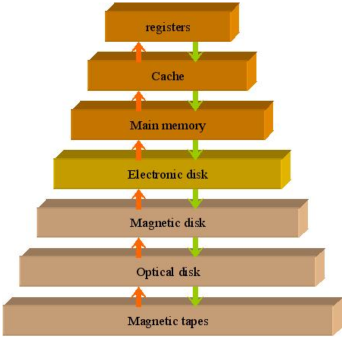
Outline

Storage hierarchy I

- 存储器是计算机系统的重要组成部分
  - 容量、价格和速度之间的矛盾
  - 内存、外存；易失性和永久性
  - 内存，是稀缺资源
- 在现代计算机系统中，存储通常采用层次结构来组织

Storage hierarchy

- Storage systems in a computer system can be organized in a hierarchy
  - Speed, access time
  - Cost per bit
  - Volatility



Memory VS. register

- **Same**: **Access directly for CPU**
  - Register name
  - Memory address
- **Different: access speed**
  - Register, one cycle of the CPU clock
  - Memory, Many cycles (2 or more)
- **Disadvantage**:
  - CPU needs to stall frequently & this is intolerable
- **Remedy**
  - **cache**

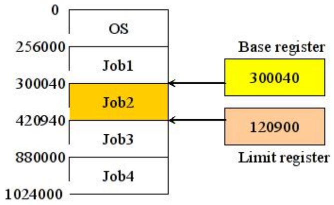
Caching

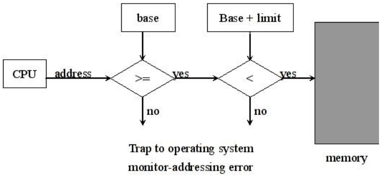
- **Caching** (高速缓存技术)
  - Copying information into faster storage system
  - When accessing, first check in the **cache**,
    - if **In**: use it directly
    - **Not in**: get from upper storage system, and leave a copy in the cache
- Using of caching
  - Registers provide a high-speed cache for main memory
  - **Instruction cache & data cache**
  - Main memory can be viewed as a fast cache for secondary storage
  - ...

Outline

Memory protection

- **Base register protection scheme**
  - Base register＋Limit register
  - Memory outside is protected
  - OS has unrestricted access to both monitor and user’ s memory
  - Load instructions for the base/limit registers are privileged

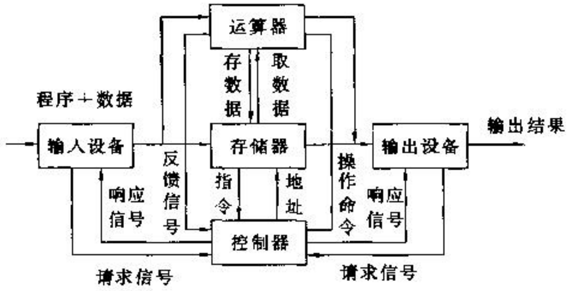




Outline

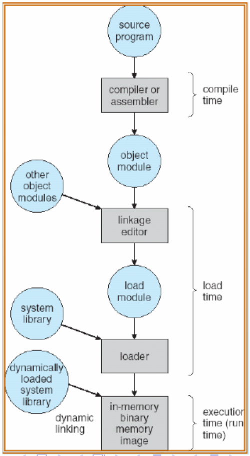
Program execution, loading & linking I

- Von Neumann architecture (冯 · 诺依曼体系结构)
  - Program must be brought into memory
  - Main memory is usually too small



Program execution, loading & linking II

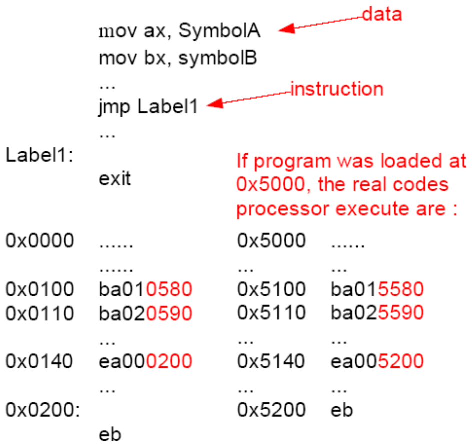
- Program must be placed within a process for it to be executed
    - User programs: Where to place the program?
    - several steps



Address Types

- Absolute address 绝对地址
  - Address seen by the memory unit
  - Physical address 物理地址
- Relative address 相对地址
  - Linear address 线性地址
- Logical address 逻辑地址
  - Generated by the CPU
  - Virtual address 虚拟地址
- **When can the absolute address can be decided?**

Example



```
        mov ax, SymbolA
        mov bx, symbolB
        ...
        jmp Label1
        ...
Label1:
        exit
```

If program was loaded at 0x5000, the real codes processor execute are :

```
0x0000  ......           0x5000  ......
        ......           ...     ...
0x0100  ba010580         0x5100  ba015580
0x0110  ba020590         0x5110  ba025590
        ...              ...     ...
0x0140  ea000200         0x5140  ea005200
        ...              ...     ...
0x0200:                  0x5200  eb
        eb
```

Address Binding

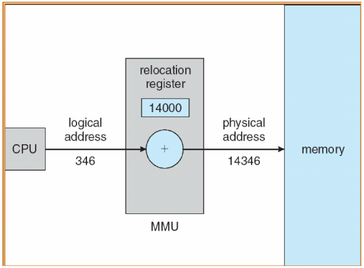
- **Address binding** of instructions and data to memory addresses can happen at **three** different stages
    - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes; MS-DOS .COM-format programs
    - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
    - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- in compile-time and load-time address-binding schemes
  - Logical addr $=$ physical addr
- in execution-time address-binding scheme
  - Logical addr $\neq$ physical addr
  - need **MMU**

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
    - Dynamic relocation using a relocation register



- The user program deals with **logical** addresses; it never sees the **real** physical addresses

Program loading & linking

Shall we put the entire program & data of a process in physical memory before the process can be executed?

- For better memory space utilization
  - Dynamic loading
  - Dynamic linking
  - Overlays
  - Swapping
  - ...

Program loading I

- 3 modes
  - Absolute loading mode
  - relocatable loading mode
  - dynamic run-time loading

1. **Absolute loading mode (绝对装入方式)**
   - compiling →absolute code with absolute addresses
   - loading →必须装入到指定的地址
   - 无需对程序和数据的地址进行修改
   - 适用于单道系统
2. **relocatable loading mode (可重定位装入方式)**
   - mostly, the loading address can not be known at compile time, but only be decided at load time.

Program loading II

- compiling → relocatable code with relative addresses
- loading → must relocate
  - 通常把在装入时对目标程序中指令和数据的修改过程称为重定位(relocation)。
- 由于地址变换是在装入时一次性完成的，以后不再改变，故称为**静态重定位 (static relocation)**
- 可用于多道系统

dynamic run-time relocation (动态运行时重定位)

- 有时候，程序会在内存中移动位置
  - 例如对换
- 需要能支持在运行过程中动态改变程序在内存中的位置
- 方法：
  推迟重定位时机
  即从相对地址到绝对地址的转换推迟到程序真正执行时才进行

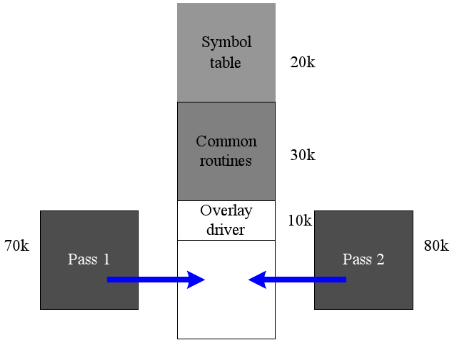
Program loading III

- 因此，装入内存中的代码和数据的地址仍然是相对地址
- 需要重定位寄存器的支持

3. **Dynamic Loading (动态运行时装入方式)**
    - 根据程序运行的局部性
        - Routine is not loaded until it is called
    - Better memory-space utilization;
      unused routine is never loaded.
    - Useful when large amounts of code are needed to handle infrequently occurring cases
        - Error routine
    - No special support from OS is required implemented through program design
        - Due to the users

Overlays 覆盖技术

- Keep in memory only those are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- **Implemented by user**, no special support needed from OS, programming design of overlay structure is complex



Overlays for a two -pass assembler

Program linking I

- 多个源程序→编译→多个目标模块/库。
  需要链接成可装入模块
- according to the time of linking
  - static linking (静态链接方式)
  - load-time dynamic linking (装入时动态链接)
  - run-time dynamic linking (运行时动态链接)
- **static linking**
  - 在程序运行之前，先将各目标模块以及它们所需要的库函数，链接成一个完整的装配模块，以后不再拆开。
    - 各目标模块内：相对地址
    - 存在目标模块之间的调用关系
    - 外部调用符号
  - 要解决的问题：

Program linking II

- 修改相对地址：多个相对地址空间→一个统一的相对地址空间
- 变换外部调用符号

- **load-time dynamic linking**
  - 在装入时，根据外部模块调用事件，使用装入程序去寻找相应的外部目标模块，并装入，在装入时，修改相对地址
  - 优点：
    - 便于修改和更新
    - 便于实现对目标模块的共享

- **运行时动态链接(Dynamic Linking)**
  - 程序的每次运行，可能要执行的目标模块集是不同的
    - 全部装入？按需装入？
  - Linking postponed until execution time

Program linking III

- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- OS needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries– **shared libraries**
- 优点：程序运行前的装入速度加快；省空间

Contiguous Memory Allocation

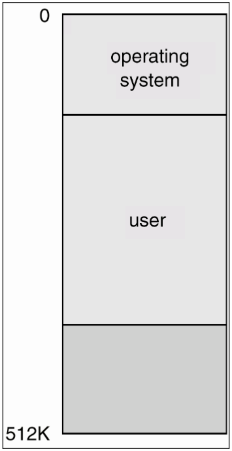
Contiguous Memory Allocation

Each process is contained in a single contiguous section of memory

- 单一连续
- Multiple-partition allocation
  - 固定分区
  - 动态分区

单一连续分配

- the most simple method
- at most one process at a time
- Main memory usually divided into two partitions:
  - Resident OS, usually held in **low** memory with **interrupt vector**
  - User processes then held in **high** memory



protection

1. MMU

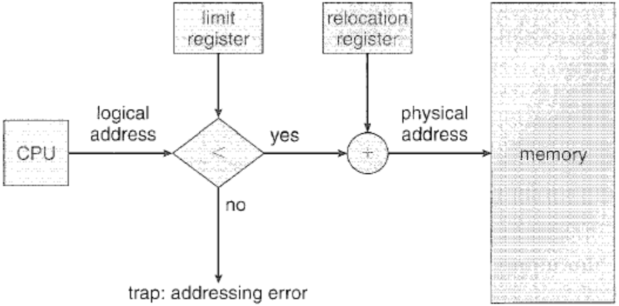


Figure 8.6 Hardware support for relocation and limit registers.

2. Maybe not use any protection

Multiple-partition allocation

- make several user porcesses reside in memory at the same time.
    - User partition is divided into *n* **partitions**
    - Each partition may contain exactly one process
        - when a partition is free, a process in input queue is selected and loaded into the free partition
        - when a process terminates, the partition becomes available for another process
    - the degree of multiprogramming is bound by **the number of partions**.
- **fixed-partition** VS. **dynamic-partition**

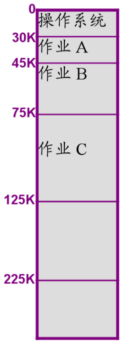
Fixed-sized-partition scheme (固定分区) I

- The simplest multi-partition method: IBM OS/360 (MFT)
  - the memory is divided into several fixed-sized partitions
  - partition size: **equal** VS. **not equal**
  - Data Structure & allocation algorithm

分区号	大小（KB）	起始地址 (K)	状态
1	15	30	已分配
2	30	45	已分配
3	50	75	已分配
4	100	125	已分配

固定分区使用表

❖ 分配算法



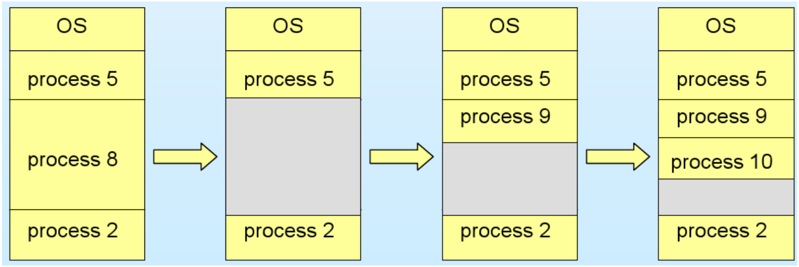
Fixed-sized-partition scheme (固定分区) II

- disadvantage
  - **poor memory utility**
  - Internal fragmentation & external fragmentation
    - **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
    - **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- ? dynamic partition

dynamic partition （动态分区） I

- **Hole** – block of available memory
  - Initially, all memory is considered one large hole;
  - When a process arrives, a hole large enough is searched. If found, the memory is allocated to the process as needed, the rest memory of the partition is keep available to satisfy future requests.
  - holes of various size are scattered throughout memory
- OS maintains information about:
  - a) allocated partitions b) free partitions (hole)

dynamic partition （动态分区） II

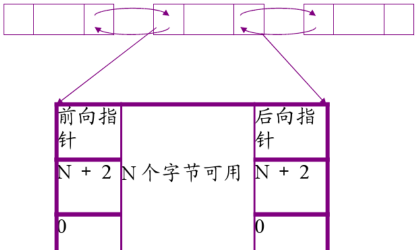


dynamic partition （动态分区） III

- **空闲分区表**，占用额外的空间

序号	分区大小	起始地址	状态

- **空闲分区链**，利用空闲分区



Dynamic Storage-Allocation Problem

- How to satisfy a request of size *n* from a list of free holes
  - **First-fit**（首次适应）：Allocate the first hole that is big enough
  - 循环首次适应
  - **Best-fit**（最佳适应）：Allocate the **smallest** hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**（最差适应）：Allocate the **largest** hole; must also search entire list
    - Produces the largest leftover hole
- **First-fit and best-fit better than worst-fit in terms of speed and storage utilization**

分区分配操作

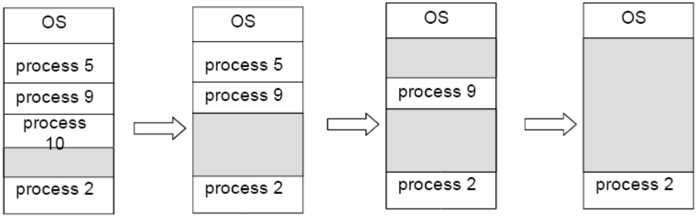
- 分配
  - 设请求的分区大小为u.size;
  - 利用某种分配算法，找到待分配的分区，大小为m.size
  - 根据上述分区分配算法，有m.size>u.size
  - 判断m.size-u.size与min_size的大小
    min_size为事先约定的最小分区大小
    - >，分割，分割出来的分配出去，余下的加入空闲数据结构
    - 否则，直接分配
  - 将分配到的分区的首地址返回
- 可以看出，动态分区分配方式中内部碎片最大不超过min_size

分区回收操作 I

- 回收，要考虑合并
  - 向前合并
    - 只需修改前一个空闲分区表项中的大小
  - 向后合并（图）
    - 只需修改后一个空闲分区表项中的起始地址和大小
  - 与前后同时合并
    - 修改前一个空闲分区表项中的大小，并取消后一个分区表项
  - 无相邻空闲分区，无需合并
    - 建立一个新的表项，填写相关信息，插入

  - 上述过程中，根据链表的维护规则，可能需要调整相应表项在空闲链表中的位置

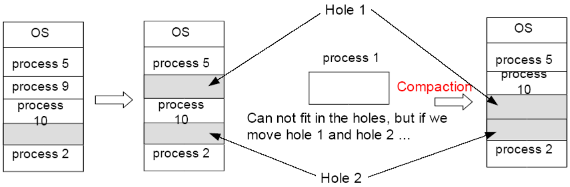
分区回收操作 II



- disadvantage
    - 随着分配的进行，空闲分区可能分散在内存的各处
    - 尽管有回收，但内存仍然被划分的越来越碎，形成大量的外部碎片
- solution
    - compaction (紧凑)

compaction (紧凑)

- Reduce **external fragmentation** by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if **relocation** is **dynamic**, and is done at execution time (运行时的动态可重定位技术)
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers



- 动态重定位分区分配算法：
  引入紧凑和动态重定位技术的动态分区分配算法

Outline

Swapping (对换) I

- 最早用于MIT的CTSS中
  - 单用户＋时间片＋对换
- 对换是指
  把内存中暂时不能运行的进程，或暂时不用的程序和数据，换出到外存上，以腾出足够的内存空间，把已具备运行条件的进程，或进程所需要的程序和数据，换入内存
- 能提高内存利用率
- 对换的单位：
  - 进程：整体对换；进程对换
  - 页、段：部分对换

Swapping (对换) II

- 对换技术需要实现三个方面的功能
  - 对换空间的管理
  - 进程的换出
  - 进程的换入
- **Backing store**
  fast disk large enough to accommodate copies of all memory images for all users;
  must provide direct access to these memory images
  - 为提高速度，考虑连续分配方式，忽略碎片问题
  - 需提供数据结构对空闲盘块进行管理
    - 方法类似动态分区分配方法

Swapping (对换) III

- **进程的换出**
  - 第一步：选择被换出的进程
    - RR scheduling:
      swapped out when a quantum expires
    - Priority-based scheduling: Roll out, roll in
      Lower-priority process is swapped out so higher-priority process can be loaded and executed.
  - 第二步：换出
    - 确定要换出的内容
      非共享的程序和数据段的换出
      共享的程序和数据段的换出：计数器
    - 申请对换空间，换出，并修改相关数据结构

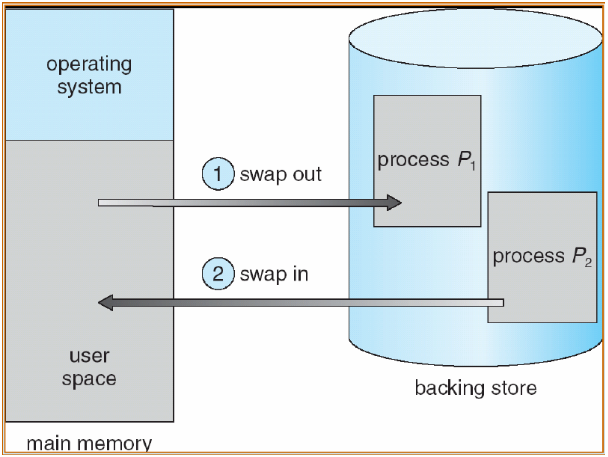
Swapping (对换) IV

- 进程的换入
  - 第一步：选择被换入的进程
    - 考虑“静止就绪状态”的进程＋其他原则
  - 第二步：申请内存并换入
    - 申请成功
    - 申请失败：利用对换技术腾出内存

Swapping (对换) V

- Context switch
  - Swapped in & out cost too much
  - Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
  - Assume: process size 1MB, disk transfer rate 5MB/sec, average latency 8ms
    - Transfer time =1MB / (5MB/sec) = 1/5 sec = 200 ms
    - Swap time = 208 ms
    - Swap out & in = 416
  - For RR scheduling, time quantum should >> 416ms
  - Problems exist for pending I/O processes swapping
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



谢谢！