操作系统原理与设计

第9章 VM1（虚存1）

陈香兰

中国科学技术大学计算机学院

2009年09月01日

提纲

Background I

- 指令必须被装载到内存中运行
- 前面的解决方案
  - **To place the entire logical address in physical memory**
  - Overlays（覆盖）
  - Dynamic loading
  - Dynamic linking
- 然而
  - 有时候作业很大；有时候作业个数很多
  - 若从物理上扩展内存，代价太高
- 思路：
  从逻辑上扩展内存

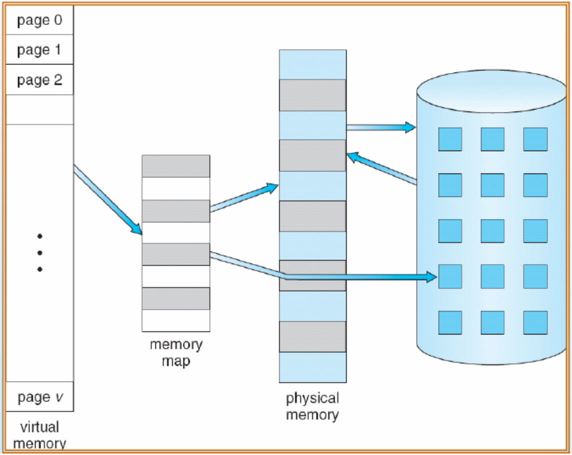
Background II

- 虚存技术的引入
  - 程序的有些部分根本得不到或者只有很少的运行的机会，例如某些错误处理程序
  - 有些数据根本得不到访问的机会
  - **程序的局部性原理**(locality of reference)，1968，Denning
    - 时间局部性、空间局部性
  - 思路：部分装入、按需装入、置换
- 虚拟存储器： **是指具有请求调度功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统**
  - 逻辑容量：
    从系统角度看：内存容量＋外存容量
    从进程角度看：地址总线宽度范围内；内存容量＋外存容量
  - 运行速度：接近内存
  - 每位成本：接近外存

Background III

- **Virtual memory** : separation of user logical memory from physical memory.
    - **Only part** of the program needs to be in memory for execution
    - Logical address space can therefore be **much larger than** physical address space
    - Allows address spaces to be shared by several processes
    - Allows for more efficient process creation
- Virtual memory can be implemented via:
    - Demand **paging**
        - 以分页技术为基础，加上
        - 请求调页（pager）功能和页面置换功能
        - 与对换相比，页面置换中
          换入换出的基本单位是页，而不是整个进程
    - Demand **segmentation**

Virtual Memory That is Larger Than Physical Memory



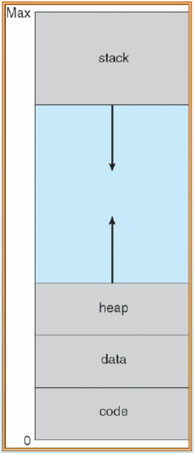
虚拟存储器的特征

- 多次性：最重要的特征
  - 一个作业被分成多次装入内存运行
- 对换性
  - 允许在进程运行的过程中，（部分）换入换出
- 虚拟性
  - 逻辑上的扩充

- 虚拟性是以多次性和对换性为基础的。
- 多次性和对换性是建立在离散分配的基础上的

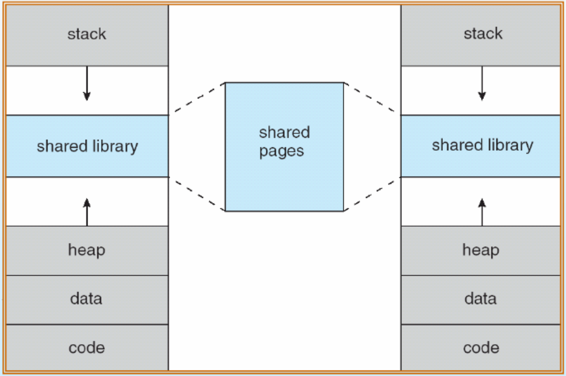
Virtual-address Space

- the **virtual address space** of a process refers to **the logical (or virtual) view of how a process is stored in memory**.
  - Typically: 0~xxx & exists in contiguous memory
- **In fact**, the physical memory are organized (partitioned) in **page frames** & the page frames assigned to a process **may not be contiguous**⇒MMU



Some benifits

1. Shared Library Using Virtual Memory

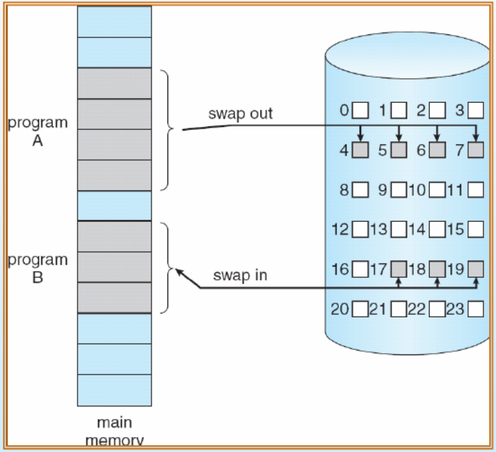


2. Shared memory
3. speeding up process creation

Demand Paging

- Do not load the entire program in physical memory at program execution time. NO NEED!
- Bring a page into memory only when it is **needed**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is **needed** ⇒ reference to it
  - invalid reference ⇒abort
  - not-in-memory ⇒bring to memory
- Swapper VS. pager
  - A swapper manipulates the entire processes
  - **Lazy swapper**
    never swaps a page into memory unless page will be needed
    - Swapper that deals with individual pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



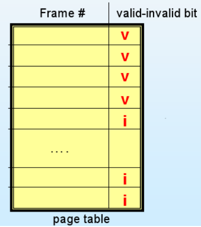
Outline

HW support

1. **the modified page table mechanism**
2. **page fault**
3. **address translation**
4. secondary memory (as swap space)

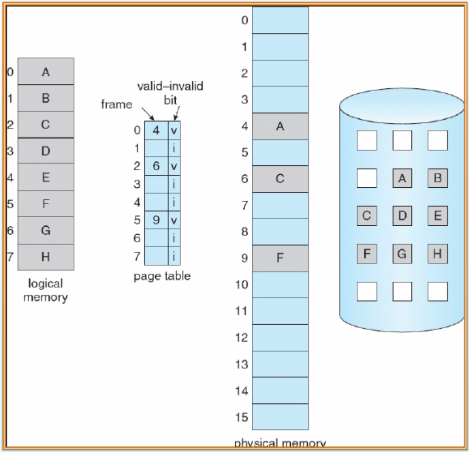
1) the modified page table mechanism

1. Valid-Invalid Bit (PRESENT bit)
   - With each page table entry a valid-invalid bit is associated
     - v ⇒ in-memory, i ⇒ not-in-memory
   - Initially valid-invalid bit is set to i on all entries
   - During address translation, if valid-invalid bit in page table entry is i ⇒ page fault



2. reference bits (for pager out)
3. modify bit (or dirty bit)
4. secondary storage info (for pager in)

Page Table When Some Pages Are Not in Main Memory



2) Page Fault I

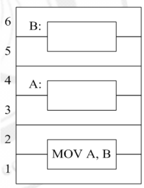
- If there is a reference to a page, first reference to that page will trap to OS:

  page fault

- Page fault trap（缺页异常）
  - Exact exception (trap)
    Restart the process in exactly the same place and state.
    Re-execute the instruction which triggered the trap
- 一条指令在执行期间可能产生多次缺页异常

2) Page Fault II

- example: One instruction & 6 page faults



- 缺页异常可能在任何一次访存操作中产生
- 一条指令可能产生多次缺页.
  取指时 以及 存取操作数时

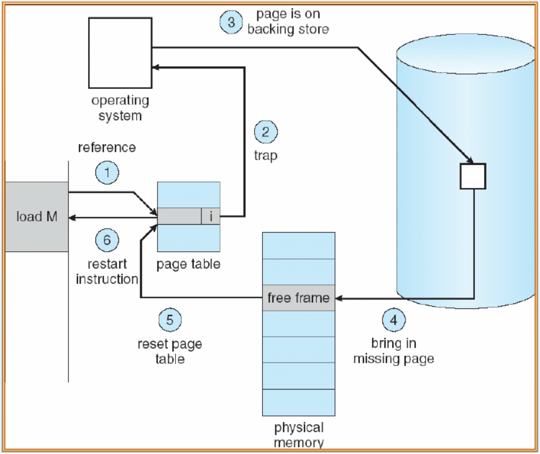
Page Fault Handling:

1. OS looks at **an internal table** to decide:
   - Invalid reference ⇒ abort
   - Just not in memory ⇒

2) Page Fault III

2. Get empty frame
3. Swap page into frame
    - pager out & pager in
4. Modify the internal tables & Set validation bit = v
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



3) address translation

- 在前面所讲的分页地址转换结构中，增加了缺页中断的处理

resume the execution

- Before OS handling the page fault, the state of the process must be saved（保存现场）
  - e.g. record its register values, PC
- The saved state allows the process to be resumed from the line where it was interrupted.（恢复现场）

- Note: distinguish the following 2 situation
  - illegal reference⇒the process is terminated
  - page fault⇒ load in or pager in

Outline

Performance of Demand Paging

- let $p =$ Page Fault Rate $(0 \leq p \leq 1.0)$
  - if $p = 0$, no page faults
  - if $p = 1.0$, every reference is a fault
- Effective Access Time (EAT)

$$
\begin{aligned}
EAT &= (1 - p) \times memory\ access \\
&\quad + p \times page\ fault\ time
\end{aligned}
$$

$$
\begin{aligned}
page\ fault\ time &= page\ fault\ overhead \\
&\quad + swap\ page\ out \\
&\quad + swap\ page\ in \\
&\quad + restart\ overhead
\end{aligned}
$$

Demand Paging Example I

- Memory access time $= 200ns$
- Average page-fault service time $= 8ms$

$$
\begin{aligned}
EAT &= (1-p) \times 200 + p \times 8ms \\
&= (1-p) \times 200 + p \times 8,000,000 \\
&= 200 + p \times 7,999,800
\end{aligned}
$$

Demand Paging Example II

- If one access out of 1,000 causes a page fault, then

$$
\begin{aligned}
p &= 0.001 \\
EAT &= 8.2\mu s
\end{aligned}
$$

  This is a slowdown by a factor of 40!!
- if we want performance degration to be less then 10%, then

$$
\begin{aligned}
EAT = 200 + p \times 7,999,800 &< 200\,(1 + 10\%) = 220 \\
p \times 7,999,800 &< 20 \\
p &< 20/7,999,800 \approx 0.0000025
\end{aligned}
$$

减少缺页处理时间的方法

- To keep the fault time low
  1. Swap space, faster then file system
  2. Only dirty page is swapped out, or
  3. Demand paging only from the swap space, or
  4. Initially demand paging from the file system, swap out to swap space, and all subsequent paging from swap space
- Keep the fault rate extremely low
  - Localization of program executing
    - Time, space

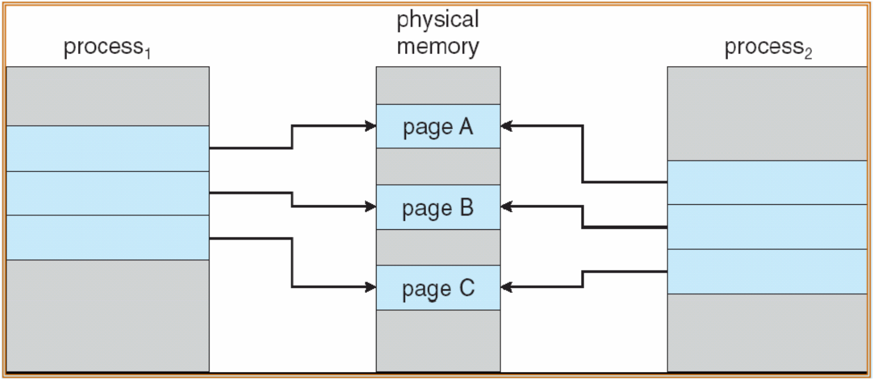
Process Creation

- Virtual memory allows other benefits during process creation:
  - **Copy-on-Write**
  - Memory-Mapped Files (later)

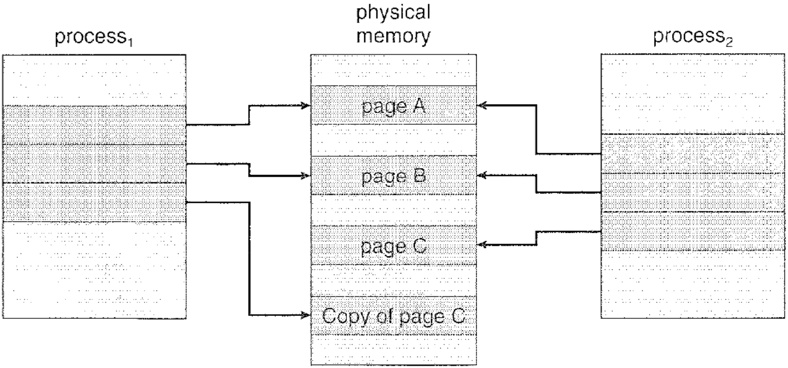
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially **share** the same pages in memory
- If either process **modifies** a shared page, only then is the page copied
- COW allows **more efficient process creation** as only modified pages are copied
- Free pages are allocated from a pool of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What happens if there is no free frame?

- Page replacement
  find some page in memory, but not really in use, swap it out
  - algorithm
  - performance
    want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

小结

1. Background
2. Demand Paging
   - Basic Concepts (HW support)
   - Performance of Demand Paging
3. Copy-on-Write
4. 小结和作业

谢谢！