0117401: Operating System
计算机原理与设计
Chapter 8: Main Memory

陈香兰

xlanchen@ustc.edu.cn

http://staff.ustc.edu.cn/~xlanchen

Computer Application Laboratory, CS, USTC @ Hefei
Embedded System Laboratory, CS, USTC @ Suzhou

May 6, 2015

温馨提示：

为了您和他人的工作学习，
请在课堂上关机或静音。

不要在课堂上接打电话。

提纲

1. background
2. Contiguous Memory Allocation (连续内存分配)
3. Swapping
4. Paging (分页)
5. Structure of the Page Table
6. Segmentation (分段)
7. Segmentation with paging (段页式)
8. 小结和作业

Outline

1. background
   - Storage hierarchy
   - Memory protection
   - Program execution, loading & linking

Outline

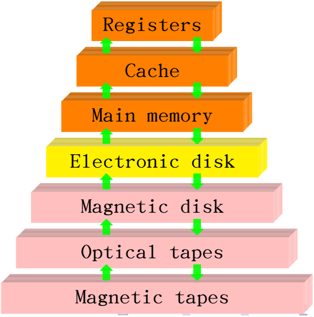
1. background
   - Storage hierarchy
   - Memory protection
   - Program execution, loading & linking

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

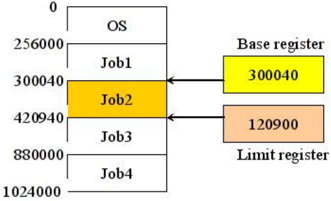
- Caching (高速缓存技术)
  - Copying information into faster storage system
  - When accessing, first check in the cache,
    - if In: use it directly
    - Not in: get from upper storage system, and leave a copy in the cache
- Using of caching
  - Registers provide a high-speed cache for main memory
  - Instruction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

Outline

1. background
   - Storage hierarchy
   - Memory protection
   - Program execution, loading & linking

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



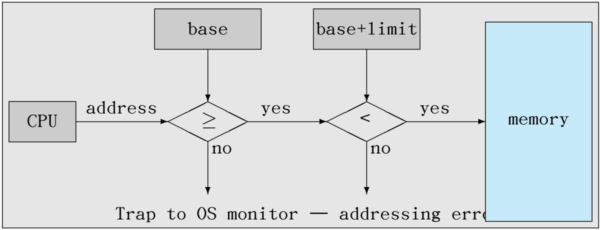


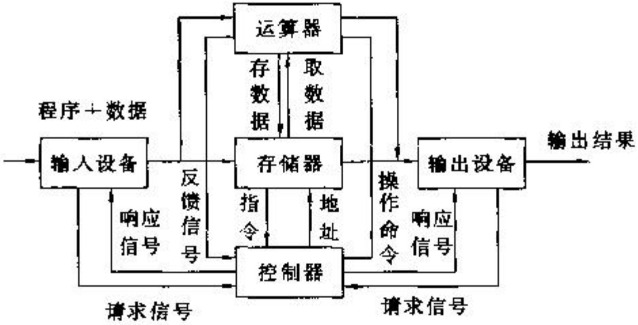
Figure: Hardware address protection with base and limit registers

Outline

1. background
   - Storage hierarchy
   - Memory protection
   - Program execution, loading & linking

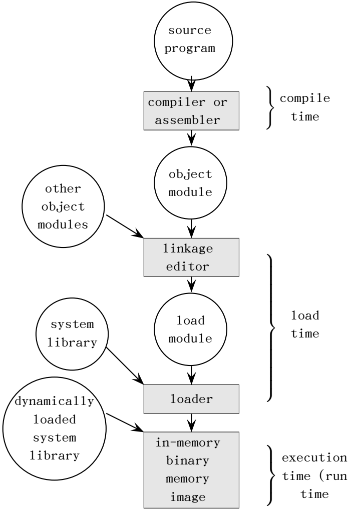
Program execution, loading & linking

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

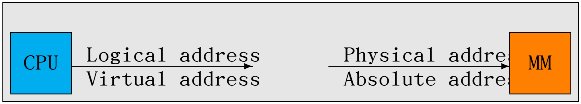


Program execution, loading & linking

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



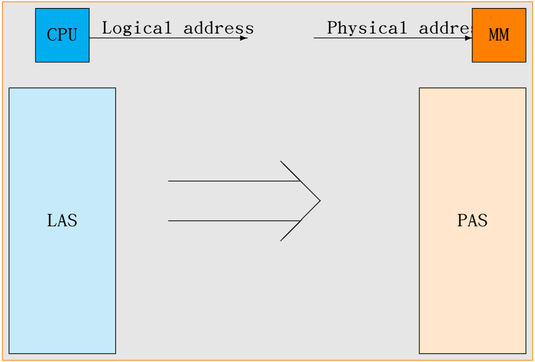
Address Types



- Absolute address (绝对地址): Address seen by the memory unit
  ALSO: Physical address (物理地址)
- Relative address (相对地址)
  ALSO: Linear address (线性地址)
- Logical address (逻辑地址): Generated by the CPU
  ALSO: Virtual address (虚拟地址)

Logical vs. Physical Address Space

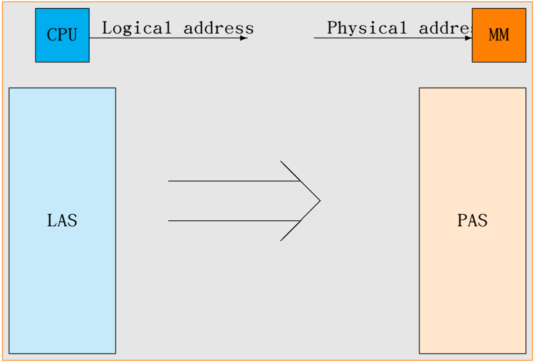
- Logical address space:
  the set of all logical addrs generated by a program
- Physical address space:
  the set of all physical addrs



- WHEN can the absolute address can be decided?

Logical vs. Physical Address Space

- Logical address space:
  the set of all logical addrs generated by a program
- Physical address space:
  the set of all physical addrs

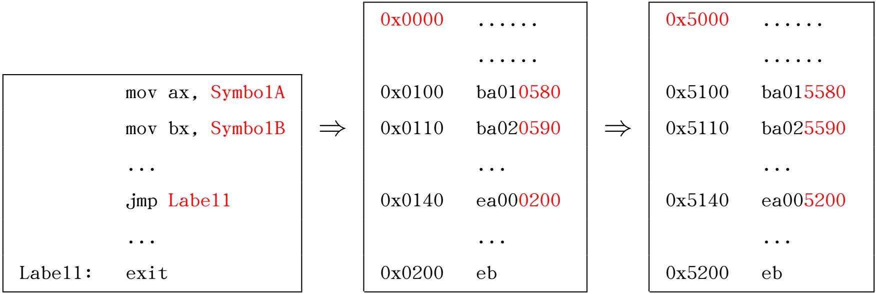


- WHEN can the absolute address can be decided?

Example

if program was loaded at 0x5000, the

real codes processor execute are:



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

⇒

```
0x0000    ......
          ......
0x0100    ba010580
0x0110    ba020590
          ...
0x0140    ea000200
          ...
0x0200    eb
```

⇒

```
0x5000    ......
          ......
0x5100    ba015580
0x5110    ba025590
          ...
0x5140    ea005200
          ...
0x5200    eb
```

relative address

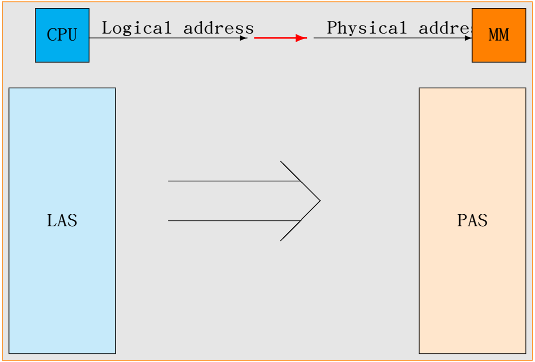
LA = PA

Address Binding

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
- Address binding of instructions and data to memory addresses can happen at three different stages
  1. Compile time:
     If memory location known a priori, absolute code (绝对代码) can be generated;
     Must recompile code if starting location changes;
     Example: MS-DOS .COM-format programs
  2. Load time:
     Must generate relocatable code (可重定位代码) if memory location is not known at compile time
  3. 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)

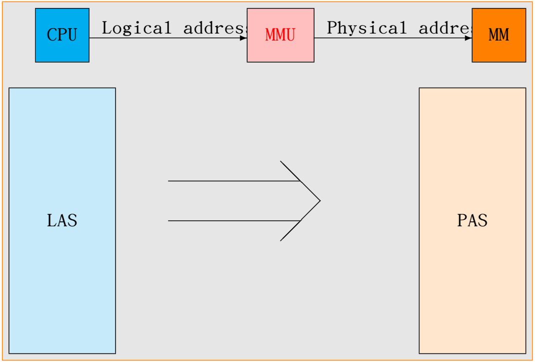
Address Binding

1. In compile-time and load-time address-binding schemes:
   - Logical addr = physical addr



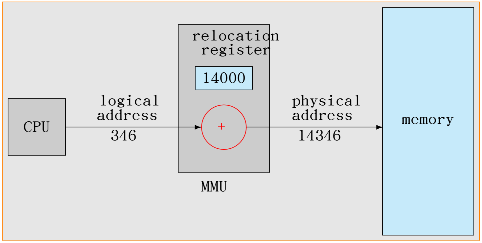
Address Binding

2. In execution-time address-binding scheme:
   - Logical addr $\neq$ physical addr;
   - need MMU



Memory-Management Unit (MMU)

- MMU: Hardware device that maps virtual to physical address
- Example: dynamic relocation using a relocation register
  - the value in the relocation register (重定位寄存器) is added to every address generated by a user process at the time it is sent to memory



- The user program deals with logical addresses [0, MAX); it never sees the real physical addresses [R+0, R+MAX)

Program loading & linking

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

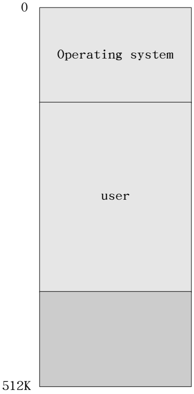
- For better memory space utilization
  1. Dynamic loading
  2. Dynamic linking
  3. Overlays
  4. Swapping
  5. ...

Program loading

- 3 modes
  1. Absolute loading mode
  2. Relocatable loading mode
  3. Dynamic run-time loading

Program loading

1. Absolute loading mode (绝对装入方式)
   - Compiling:
     - Absolute code with absolute addresses
   - Loading:
     - Must be loaded into the specified address
     - Loading address = absolute address
   - Execution:
     - Logical address = absolute address
   - Suitable for simple batch systems (单道系统)

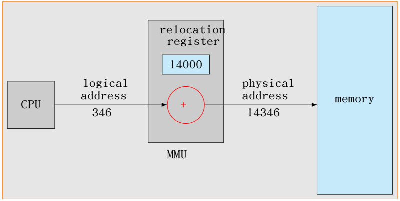


Program loading

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.
    - Compiling:
        - Relocatable code with relative addresses
    - Loading:
        - According to loading address, relative addresses in file is modified to absolute addresses in memory
        - This is called relocation (重定位)
        - Static relocation (静态重定位): because the address binding is completed one-time at load time, and will not be changed after
    - Execution:
        - Logical address = absolute address
    - Suitable for multiprogramming systems (多道系统)

Program loading

2. Relocatable loading mode (可重定位装入方式)
   - The in-memory location of a program may be changed, that is, the physical addresses is changed during execution
     - Example: swapping
   - How? ⇒ To postpone the relocation time to real execution dynamic run-time relocation (动态运行时重定位)
     - Loading:
       Addresses in memory = relative address
     - Execution:
       Logical address = relative address
     - need MMU with relocation register

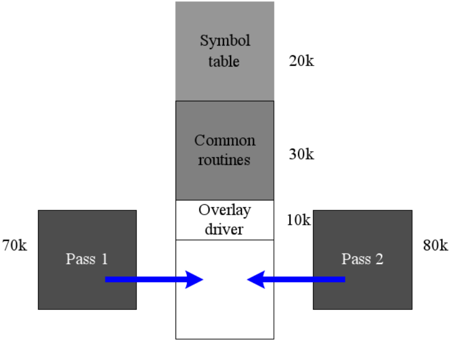


Program loading

3. Dynamic Loading (动态运行时装入方式)
   - Based on the principle of locality of reference (局部性原理)
     - The main program is loaded into memory and is executed
     - Routine is not loaded until it is called
   - Loading while execution: need the relocatable linking loader
     - before loading: relocatable code
     - while calling and not in:
       load the desired routine, update the program' s address tables and the control is passed to the newly loaded routine
   - Advantage:
     - Better memory-space utilization;
     - unused routine is never loaded.
   - Useful when large amounts of code are needed to handle infrequently occurring cases
     - Example: Error routine
   - No special support from OS is required
     - Due to the users
     - Special library routines that implementing dynamic loading are needed

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

- source files $\xrightarrow{\text{compiling}}$ object modules $\xrightarrow{\text{linking}}$ loadable modules
- according to the time of linking
  1. static linking (静态链接方式)
  2. load-time dynamic linking (装入时动态链接)
  3. run-time dynamic linking (运行时动态链接)

Program linking

1. static linking (静态链接方式)
   - Before loading, all object modules and required libraries are linked into one loadable binary program image.
     - In object modules and (static) libraries: relative address
     - Exist external calls or references to external symbols (functions or variables):
       object modules ←→object modules; object modules →libraries
   - While linking:
     - relative addresses are modified:
       multiple relative address spaces → one relative address space
     - External calls and references are deliminated
   - Disadvantage:
     Each program on a system must include a copy of required libraries
     (or at leasted required routines)
     - Example: language libraries

Program linking

2. load-time dynamic linking (装入时动态链接)
   - Linking while loading:
     - External calls and references are deliminated
       According to external calls and references, the loading program find the required object modules and libraries, and load them into memory
     - Relative addresses are modified:
       multiple relative address spaces → one relative address space
   - Advantage:
     - Easy to modify and update the object modules and libraries
     - Easy to share the object modules and libraries

Program linking

3. Dynamic Linking (运行时动态链接)
   - Every execution time, the set of executed modules of a program may different
     - load all? on demand?
     - Linking postponed until execution time
   - While linking:
     - A stub is included in the image for each library-routine references
     - The stub is a small piece of code, used to locate the appropriate memory-resident library routine
   - During execution:
     - 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
   - Advantage:
     - short load time; less memory space

Outline

2 Contiguous Memory Allocation (连续内存分配)

Contiguous Memory Allocation (连续内存分配)

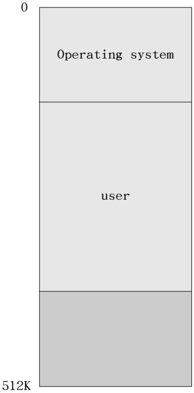
Contiguous Memory Allocation (连续内存分配)

Each process is contained in a single contiguous section of memory

1. Monoprogramming memory allocation (单一连续)
2. Multiple-partition allocation
    1. 固定分区
    2. 动态分区

Monoprogramming memory allocation (单一连续分配)

- Monoprogramming memory allocation (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



Monoprogramming memory allocation (单一连续分配)

- Memory protection scheme
  1. Use MMU, for example

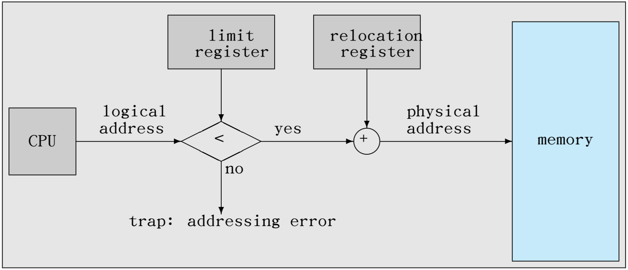


Figure: Hardware support for relocation and limit registers

  2. Maybe not use any protection

Multiple-partition allocation (多分区分配)

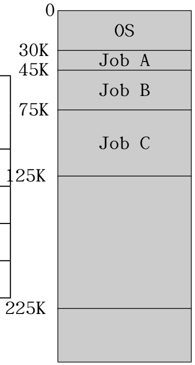
- Make several user processes reside in memory at the same time.
    - User partition is divided into n partitions
    - Each partition may contain exactly one process
        - When a partition is free, a process in input queue is selected and loaded into the free partition
        - When a process terminates, the partition becomes available for another process
    - The degree of multiprogramming (多道程序度) is bound by the number of partions.

1. Fixed-partition (固定分区)
2. Dynamic-partition (动态分区)

Fixed-sized-partition scheme (固定分区)

- 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

partition number	size (KB)	start addr (KB)	state
1	15	30	allocated
2	30	45	allocated
3	50	75	allocated
4	100	125	allocated

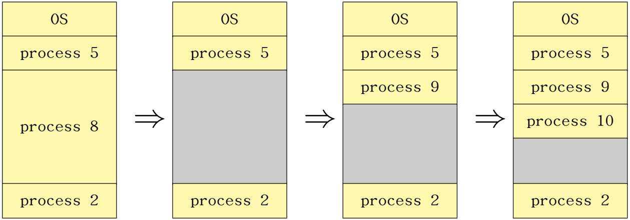


Fixed-sized-partition scheme（固定分区）

- Disadvantage
  - Poor memory utility
  - Internal fragmentation & external fragmentation
    - Internal Fragmentation（内部碎片）
      Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
    - External Fragmentation（外部碎片）
      Total memory space exists to satisfy a request, but it is not contiguous

Dynamic partition scheme （动态分区）

- 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.



Dynamic partition scheme （动态分区）

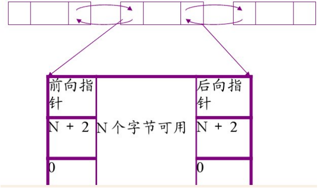
- OS maintains information about:
  1. Allocated partitions
  2. Free partitions (hole)

Example:

1. Free partitions table: need extra memory to store the table

Partition number	partition size	start address	state

2. Free partitions list: can make use of the free partitions to store links and partition infomation



Dynamic partition scheme （动态分区）

- Dynamic Storage-Allocation Problem:
  How to satisfy a request of size n from a list of free holes
  1. First-fit（首次适应）: Allocate the first hole that is big enough
  2. Next-Fit（循环首次适应）: Allocate the next hole that is big enough
  3. 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
  4. 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

Dynamic partition scheme （动态分区）

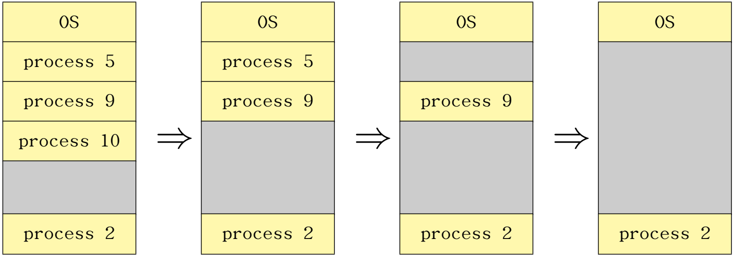
- Partition allocation operation(分配操作)
  - Suppose u is the requested partition, the size is u.size
  - Find a suitable partition m based on an algorithm (above-mentioned), the size is m.size, we have

$$m.size \geq u.size$$

  - Let min_size be the partition’s minimal size allowed
    - If $m.size - u.size > min\_size$, partition m is divided into two partitions,
      one is for partition u, the other is added into free partitions
    - Otherwise, let partition m be partition u
  - The first address of partition u is returned
  - The max size of internal fragmentations $\leq$ min_size

Dynamic partition scheme （动态分区）

- Partition deallocation operation (or free, 回收/释放操作): suppose the size is dealloc_size

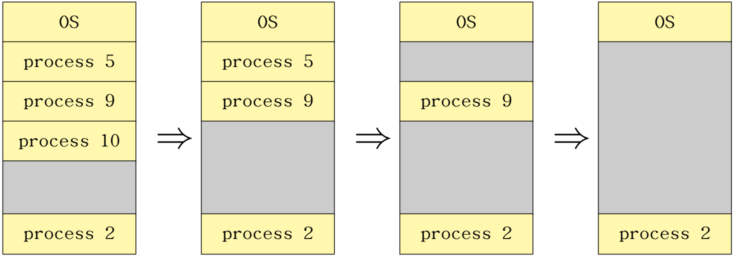


1. Combine with the prev free neighbor
   - Only need to expand the size of the prev neighbor partition

$$\text{prev.size} += \text{dealloc\_size}$$

Dynamic partition scheme （动态分区）

- Partition deallocation operation (or free, 回收/释放操作): suppose the size is dealloc_size

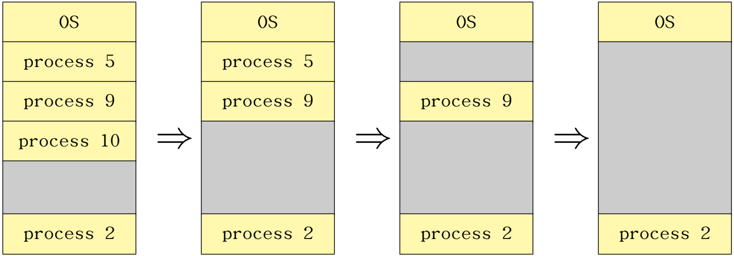


2. Combination with the next free neighbor
    - Only need to modify the start address and the size of the next neighbor partition

$$\begin{cases} next.start\_addr & -= dealloc\_size \\ next.size & += dealloc\_size \end{cases}$$

Dynamic partition scheme （动态分区）

- Partition deallocation operation (or free, 回收/释放操作): suppose the size is dealloc_size

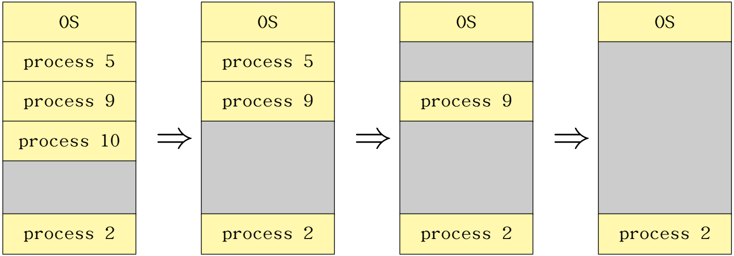


3. Combination with both neighbors
   - Expand the size of the prev neighbor partition, and delete the next partition item

$$\text{prev.size} += \text{dealloc\_size} + \text{next.size}$$

Dynamic partition scheme （动态分区）

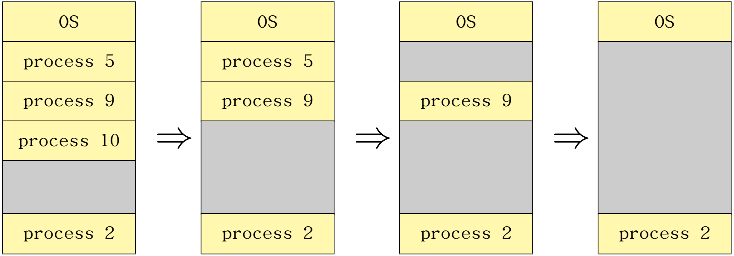
- Partition deallocation operation (or free, 回收/释放操作): suppose the size is dealloc_size



4. No free neighbor, no combination
   - Build a new partition item, fill-in related information, and then insert it into free partitions (structure)

Dynamic partition scheme （动态分区）

- Partition deallocation operation (or free, 回收/释放操作): suppose the size is dealloc_size



OS
process 5
process 9
process 10
process 2
⇒
OS
process 5
process 9
process 2
⇒
OS
process 9
process 2
⇒
OS
process 2

4. No free neighbor, no combination
   - Build a new partition item, fill-in related information, and then insert it into free partitions (structure)

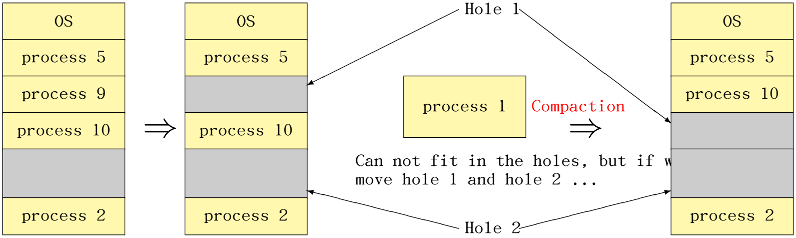
- 上述过程中，根据链表的维护规则，可能需要调整相应表项在空闲链表中的位置

Dynamic partition scheme （动态分区）

- 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（紧凑）
  - 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; Solution:
    1. Latch job in memory while it is involved in I/O
    2. Do I/O only into OS buffers



Can not fit in the holes, but if w
move hole 1 and hole 2 ...

- 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

3 Swapping

- Swapping (对换)

Outline

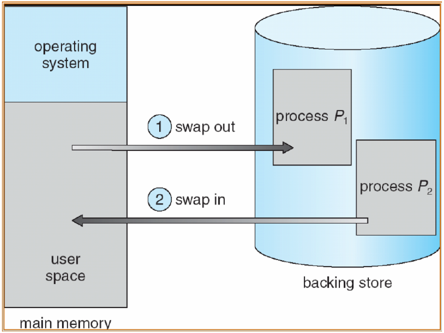
3 Swapping

- Swapping （对换）

Swapping（对换）

- Swapping（对换）
  A process (or segment, data, etc.) can be swapped temporarily out of memory to a backing store and then brought back memory for continued execution.

- Advantage: memory utilization↑
- First used in CTSS, MIT:
  - Single user + time slice + swapping
- Unit of swapping:
  - Process: whole swapping; process swapping
  - Page, segment: partly swapping



Swapping（对换）

- Swapping requires:
  1. Management of backing store（对换空间）
  2. Swap out (or roll out)
  3. Swap in (or roll in)

1. Backing store

   Fast disk large enough to accommodate copies of all memory images for all users;

   Must provide direct access to these memory images

   - In order to speed-up, consider the contiguous allocation, and ignore the fragmentation problem
   - Need to provide data structure to manage the free disk block
     - Similar to dynamic partition allocation

Swapping（对换）

2. Process swap out
   - Step 1: select a process to be swapped out
     - 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.
   - Step 2: swap out
     - Determine the content to be swapped out
       (1) Code and data segments that are non-sharable
       (2) Code & data segments that are sharable: counter (计数器)
     - Allocate spaces on backing store, swap out, and modify the related data structures

Swapping（对换）

3. Process swap in
   - Step 1: select a process to be swapped in
     - Process with static ready state(静止就绪状态) + other principles
     - Ready queue: all ready processes on backing store or in memory
   - Step 2: allocate memory space and swap in
     - If memory is available, ...
     - Otherwise, free memory by swapping out other processes

Swapping（对换）

- Context switch(上下文切换) with swapping
  1. Swapped in & out COST TOO MUCH!

Assume:
process size 10MB, disk transfer rate 40MB/sec, average latency 8ms

Transfer time =10MB / (40MB/sec) = 1/4 sec = 250 ms

Swap time = 258 ms

Swap out & in = 516

- Major part of swap time is transfer time;
  Total transfer time is directly proportional to the amount of memory swapped
- For efficient CPU utilization, the execution time must be long relative to the swap time.
  For RR scheduling, time quantum should >> 516ms

Swapping（对换）

- Context switch(上下文切换) with swapping
  1. Swapped in & out COST TOO MUCH!
  2. Problems exist for processes swapping with pending I/O (similar to the I/O problem of compaction（紧凑))
     - Solution 1: never swap processes with pending I/O
     - Solution 2: only execute I/O operation via OS buffers
- Modified versions of swapping are found on many systems
  - i.e., UNIX, Linux, and Windows

Discrete Memory Allocation (离散内存分配)

1. paging (分页)
   - internal fragmentation <one page
2. segmentation (分段)
   - logical
3. combined paging & segmentation (段页式)

Outline

4 Paging（分页）

- Basic Method
- Hardware support
- Memory Protection（内存保护）
- Shared Pages（页共享）

Outline

4 Paging（分页）
- Basic Method
- Hardware support
- Memory Protection（内存保护）
- Shared Pages（页共享）

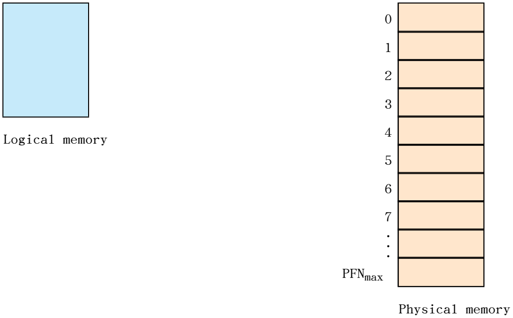
Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method

  1. Divide physical memory into fixed-sized blocks called frames (物理页框): size is power of 2, 512B–8,192B
     - Page Frame Number (物理页框号, PFN): $0, 1, \ldots, \mathrm{PFN}_{\max}$

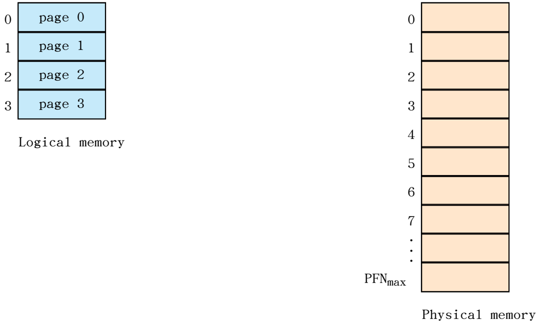
Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
  1. Divide physical memory into fixed-sized blocks called frames（物理页框）: size is power of 2, 512B—8,192B
     - Page Frame Number（物理页框号，PFN）: $0, 1, ..., \mathrm{PFN}_{\mathrm{max}}$



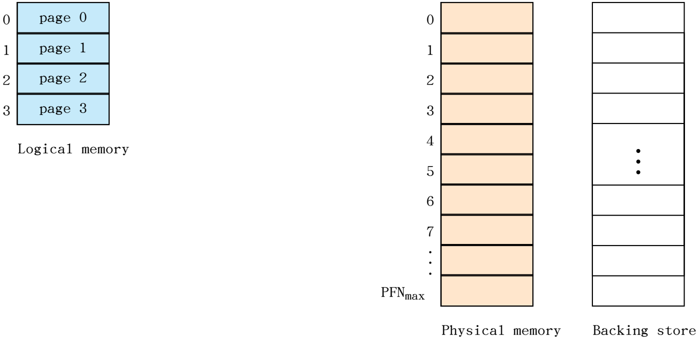
Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
    2. Divide logical memory into blocks of same size called pages（逻辑页，页）
        - Logical Frame Number（逻辑页框号，LFN）: $0, 1, \ldots, \mathrm{LFN}_{max}$



Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
  3. The backing store is also divided into fixed-sized blocks of same size as frames

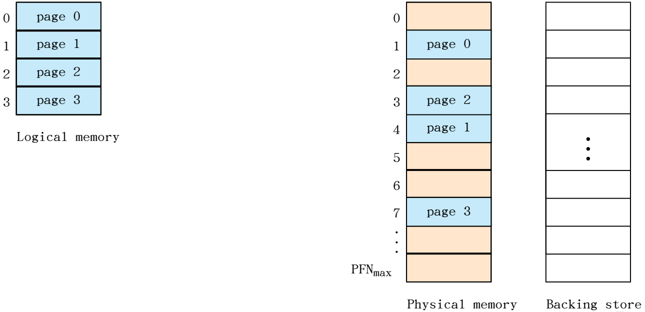


Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
    4. Need hardware and software support for paging
        1. Keep track of all free frames

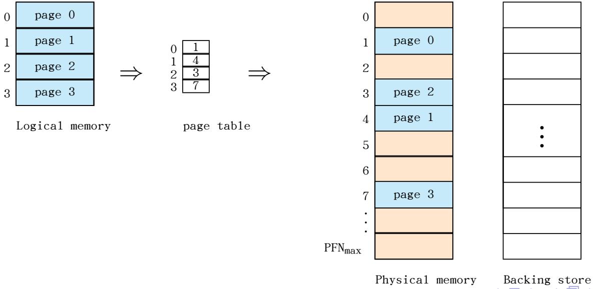
Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
  4. Need hardware and software support for paging
     1. Keep track of all free frames
     2. To run a program of size n pages, need to find n free frames and load program



Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
  4. Need hardware and software support for paging
     1. Keep track of all free frames
     2. To run a program of size n pages, need to find n free frames and load program
     3. Set up a page table to translate logical to physical addresses for each process



Logical memory

0	page 0
1	page 1
2	page 2
3	page 3

⇒

page table

0	1
1	4
2	3
3	7

⇒

Physical memory

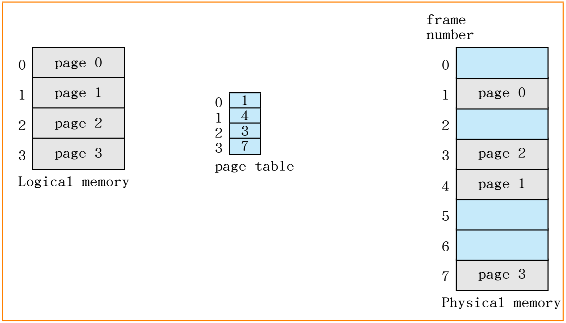
0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3
⋮	
PFN_{max}	

Backing store

Paging（分页）

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
  1. Divide physical memory into fixed-sized blocks called frames (物理页框): size is power of 2, 512B—8,192B
     - Page Frame Number (物理页框号, PFN): $0, 1, ..., \mathrm{PFN}_{\max}$
  2. Divide logical memory into blocks of same size called pages (逻辑页，页)
     - Logical Frame Number (逻辑页框号, LFN): $0, 1, ..., \mathrm{LFN}_{\max}$
  3. The backing store is also divided into fixed-sized blocks of same size as frames
  4. Need hardware and software support for paging
     1. Keep track of all free frames
     2. To run a program of size n pages, need to find n free frames and load program
     3. Set up a page table to translate logical to physical addresses for each process
- Internal fragmentation $<$ page size

Paging Model of Logical and Physical Memory



Logical memory

0	page 0
1	page 1
2	page 2
3	page 3

page table

0	1
1	4
2	3
3	7

Physical memory

frame number	
0	
1	page 0
2	
3	page 2
4	page 1
5	
6	
7	page 3

Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number (p), LFN
  - Page offset (d)
- How to get p and d?

Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number (p), LFN
  - Page offset (d)

- How to get p and d?
  - Let

    A: An address, either logical address or physical address

    L: The size of a page or page frame

    p and d: The corresponding number of the page (frame), and page offset

$$\begin{cases} p = A \ \% \ L \\ d = A \bmod L \end{cases}$$

Address Translation Scheme

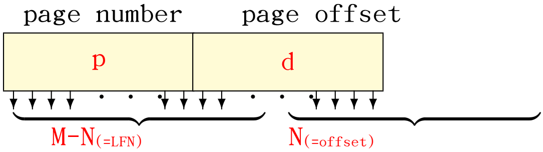
- Address generated by CPU is divided into:
  - Page number (p), LFN
  - Page offset (d)

- How to get p and d?
  - Suppose L = 2^N:

$$\begin{cases} p = A \text{ right\_shrift } N,\ \text{即}A\text{的高}(M-N)\text{位} \\ d = A\text{的低端}N\text{位} \end{cases}$$

Address Translation Scheme

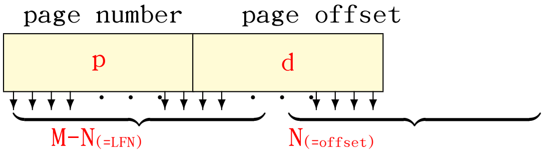
- Address generated by CPU is divided into:
  - Page number (p), LFN
  - Page offset (d)



For given logical address space 2^m and page size 2^n

Address Translation Scheme

- Address generated by CPU is divided into:
  - Page number (p), LFN
  - Page offset (d)



For given logical address space 2^m and page size 2^n

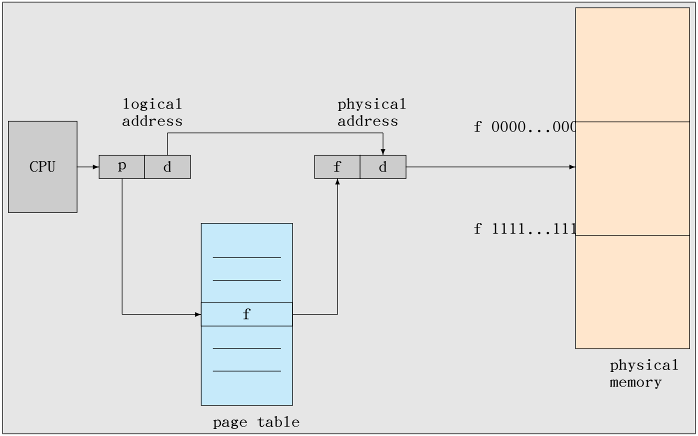
- For 32bits system & 4KB page size, $M = 32$, $N = 12$, $M - N = 20$

Example: $A = 0x\underbrace{1\ 2\ 3\ 4\ 5}_{p}\ \underbrace{6\ 7\ 8}_{d}$

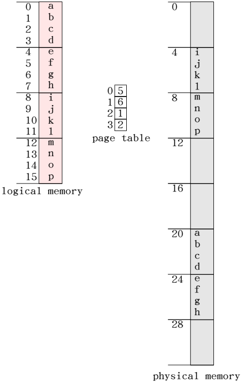
Address Translation Scheme: Paging Hardware

- Paging Hardware:

$$\underbrace{\text{LFN}\,(p) + \text{offset}\,(d)}_{\text{Logical address}} \rightarrow \underbrace{\text{PFN}\,(f) + \text{offset}\,(d)}_{\text{Physical address}}$$



Paging Example



32-byte memory with 4-byte pages

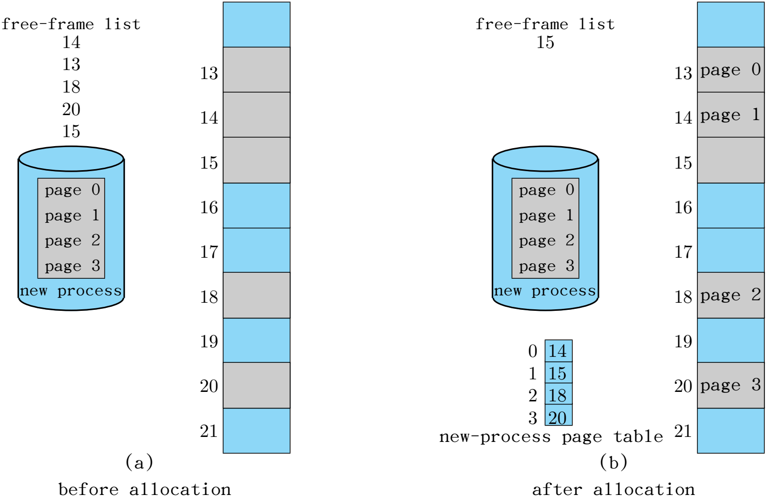
- What if read logical address 9?

Free Frames

- Since OS is managing physical memory, it must be aware of the allocation details of physical memory
  - which frames are allocated
  - which frames are available
  - how many total frames
  - ...

Free Frames

- **Frame table**: one entry for each physical page frame



Outline

4 Paging（分页）

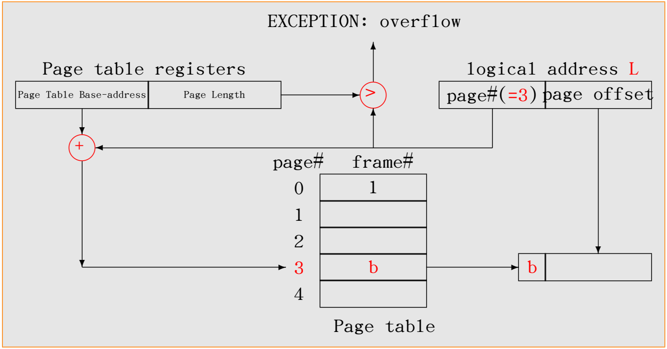
- Basic Method
- Hardware support
- Memory Protection（内存保护）
- Shared Pages（页共享）

Hardware support

- Special hardware (software) is needed to implement page table
  1. Basic paging hardware
  2. Paging hardware with TLB

Hardware support

1. Implementation of Page Table : basic paging hardware
    - Page table is kept in main memory
        - Page-table base register (PTBR) points to the page table
        - Page-table length register (PRLR) indicates size of the page table



Hardware support

1. Implementation of Page Table : basic paging hardware
   - Page table is kept in main memory
     - Page-table base register (PTBR) points to the page table
     - Page-table length register (PRLR) indicates size of the page table
   - Context switch?
     - Each process is associated with a page table.
     - Page table must be switched, too.

Hardware support

1. Implementation of Page Table : basic paging hardware
   - Effective memory-Access Time (EAT，有效访问时间)
     - Every data/instruction access requires two memory accesses.

     1. One for the page table
     2. One for the data/instruction.

Hardware support

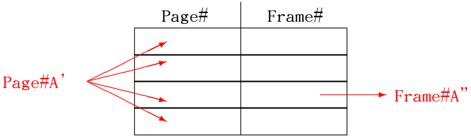
1. Implementation of Page Table : basic paging hardware

   - Effective memory-Access Time (EAT, 有效访问时间)
     - Every data/instruction access requires two memory accesses.

     1. One for the page table
     2. One for the data/instruction.
   - Solution to two memory access problem:
     - A special fast-lookup hardware cache called
       associative memory or
       translation look-aside buffers (TLBs)

Hardware support

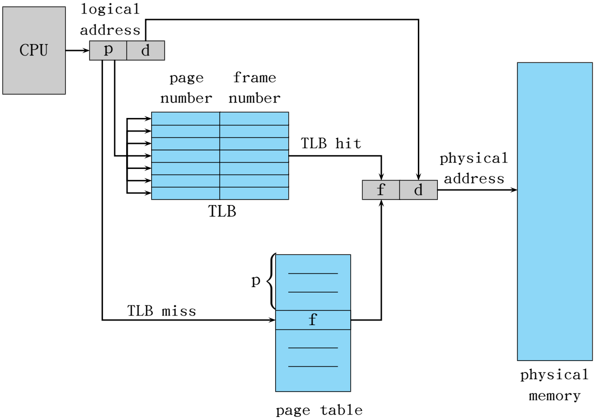
2. Paging Hardware With TLB
   - Associative Memory
     - Each register: a key & a value
     - Parallel search (high speed)
     - Expensive, typically 8~2048 entries



   Address translation (A' , A")
     - If A' is in associative register, get frame # out
     - Otherwise get frame # from page table in memory

Hardware support

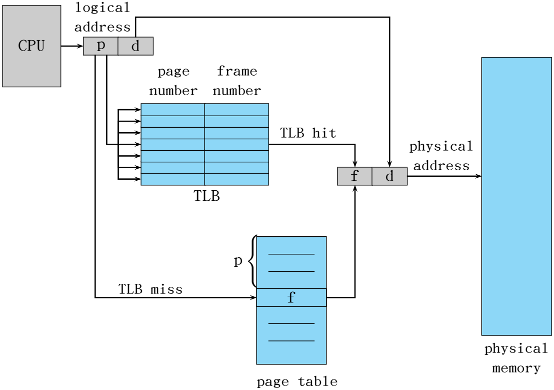
2. Paging Hardware With TLB



- Context Switch?
  - TLB must be flushed after context is switched!

Hardware support

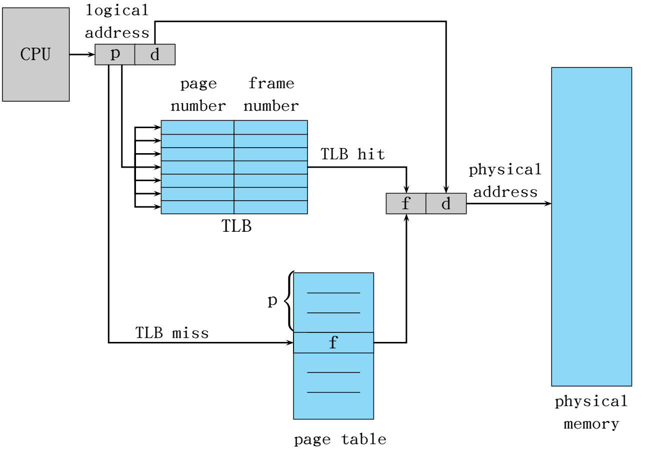
2. Paging Hardware With TLB



- Some TLBs store address-space identifiers (ASIDs) in each TLB entry
  - Uniquely identifies each process to provide address-space protection for that process

Hardware support

2. Paging Hardware With TLB



- NOTE: CACHE VS. TLB

TLB Miss

- TLB miss（TLB缺失）
  - If the page number is not in the associative registers
    - Get & store
- Hit ratio（命中率）
  - The percentage of times that a page number is found in the associative registers
  - Ratio related to number of associative registers

1. What will be happened after context is swiched?
2. TLB replacement algorithm

Effective Access Time（有效访问时间）

- If
  - Associative Lookup = ϵ time unit
  - Assume memory cycle time is t microsecond
  - Hit ratio = α
- Then Effective Access Time (EAT)

$$\begin{aligned} \mathrm{EAT} &= (\mathrm{t}+\epsilon)\,\alpha + (2\mathrm{t}+\epsilon)\,(1-\alpha) \\ &= 2\mathrm{t}+\epsilon-\mathrm{t}\alpha \end{aligned}$$

- If $\epsilon = 20\mathrm{ns}$，$\mathrm{t} = 100\mathrm{ns}$，$\alpha 1 = 80\%$，$\alpha 2 = 98\%$：
  - If TLB hit: $20 + 100 = 120\mathrm{ns}$
  - If TLB miss: $20 + 100 + 100 = 220\mathrm{ns}$
  - $\mathrm{EAT1} = 120 * 0.8 + 220 * 0.2 = 140\mathrm{ns}$
  - $\mathrm{EAT2} = 120 * 0.98 + 220 * 0.02 = 122\mathrm{ns}$

Outline

4 Paging (分页)

- Basic Method
- Hardware support
- Memory Protection (内存保护)
- Shared Pages (页共享)

Memory Protection（内存保护）

- If page size 2^n, page & frame is aligned at 2^n, so ⋯

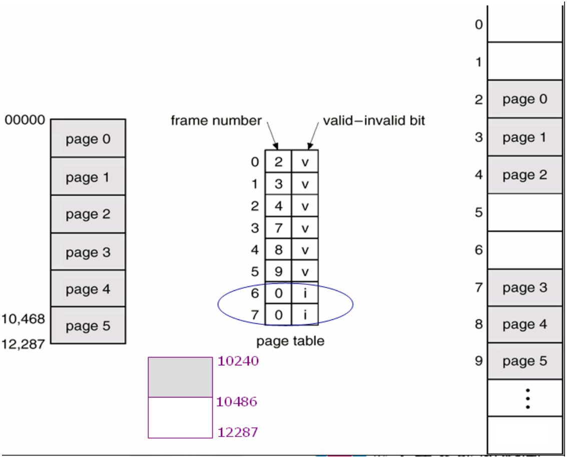
  $$\text{Example:}\quad A = 0x\ \underbrace{1\ 2\ 3\ 4\ 5}_{p}\ \underbrace{6\ 7\ 8}_{d}$$

  Only 0x12345 is stored in the page table entry.
- Memory protection implemented by associating protection bit with each frame
  - Provide read only, read-write, execute-only protection or⋯
  - Valid-invalid bit attached to each entry in the page table:
    - ' valid' indicates that the associated page is in the process' logical address space, and is thus a legal page
    - ' invalid' indicates that the page is not in the process' logical address space

Memory Protection（内存保护）

- Example: Valid (v) or Invalid (i) Bit In A Page Table

- Address space 2^{14}, Page size 2KB; Process size (0~10468)
- Page 5 has internal fragmentation
- PTLR=6, Page 6 & 7 are invalid



	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

Outline

4 Paging (分页)

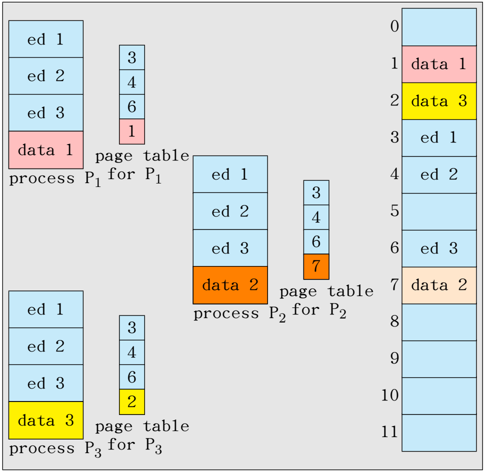
- Basic Method
- Hardware support
- Memory Protection (内存保护)
- Shared Pages (页共享)

Shared Pages（页共享）

- Shared code
  - One copy of read-only (reentrant, 可重入) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
    - WHY?
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages（页共享）

- Shared Pages Example



Outline

5 Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables (哈希页表)
- Inverted Page Tables (反置页表)

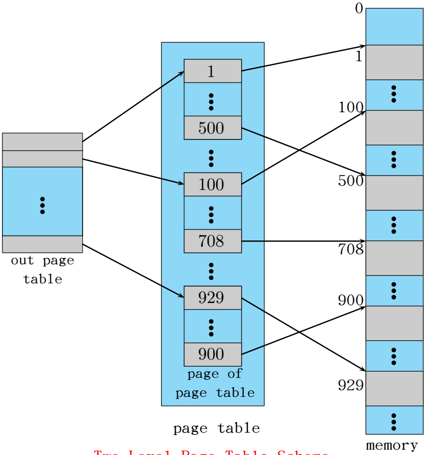
Outline

5 Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables (哈希页表)
- Inverted Page Tables (反置页表)

Hierarchical Page Tables

- Break up the LAS into multiple page tables
  - Need directories
  - A simple technique is a two-level page table



Two-Level Page-Table Scheme

Two-Level Paging Scheme

- On 32-bit machine with 4K page size, a logical address is divided into
  - Page number: 20 bits & page offset: 12 bits
  - Since the page table is paged, the page number is further divided into:
    - A 10-bit page number & a 10-bit page offset
  - Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

Where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Paging Scheme

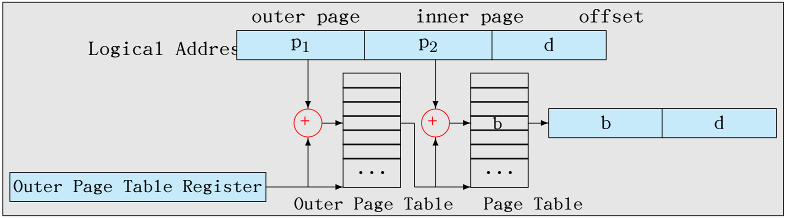
- Example

$$A = 0x\ \underbrace{1\ 2\ 3\ 4\ 5}_{p}\ \underbrace{6\ 7\ 8}_{d}$$

$$p = 0x12345 = \overbrace{\underbrace{0001}_{1}\ \underbrace{0010}_{2}\ \underbrace{0011}_{3}}^{p_2 = 0x48}\ \overbrace{\underbrace{0100}_{4}\ \underbrace{0101}_{5}}^{p_1 = 0x345}$$

Two-Level Paging Scheme

- Address-Translation Scheme



Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_2	d
32	10	10	12

Performance of multi-level page tables

- Assume memory cycle time is t microsecond,
  If Level number $=$ L, then

$$\mathrm{EAT} = (\mathrm{L}+1)\mathrm{t}$$

- If using TLB, Assume Associative Lookup = ϵ time unit, Hit ratio = α

$$\begin{cases} \mathrm{EAT} = \alpha\left(\mathrm{t}+\epsilon\right)+\left(1-\alpha\right)\left(\left(\mathrm{L}+1\right)\mathrm{t}+\epsilon\right) \\ \mathrm{t} = 100\mathrm{ns} \\ \epsilon = 20\mathrm{ns} \\ \alpha = 0.98 \\ \mathrm{L} = 3 \end{cases}$$

$$\Rightarrow \begin{aligned} \mathrm{EAT} &= 0.98 \times 120 + 0.02 \times 420 \\ &= 126\mathrm{ns} \end{aligned}$$

which is only a 26% slowdown in memory access time.

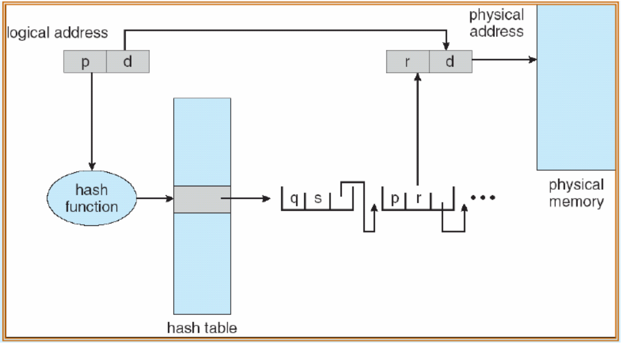
Outline

5 Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables (哈希页表)
- Inverted Page Tables (反置页表)

Hashed Page Tables (哈希页表)

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table.
  This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match.
  If a match is found, the corresponding physical frame is extracted.



Outline

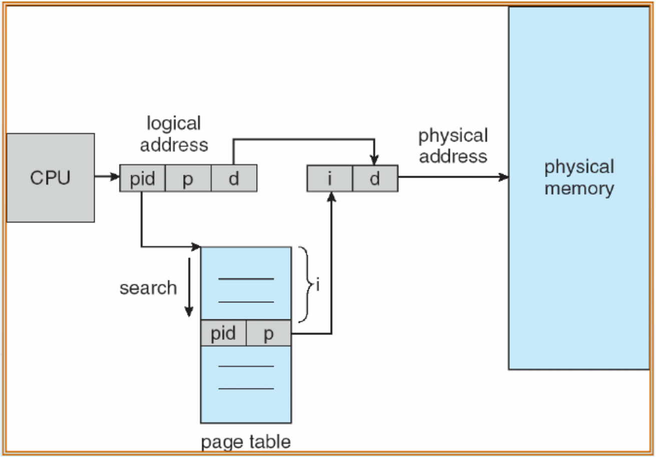
5 Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables (哈希页表)
- Inverted Page Tables (反置页表)

Inverted Page Table (反置页表)

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

Inverted Page Table Architecture



Discrete Memory Allocation（离散内存分配）

1. paging（分页）
   - internal fragmentation <one page
2. segmentation（分段）
   - logical
3. combined paging & segmentation（段页式）

Outline

6 Segmentation (分段)

- Basic Method
- Hardware

Outline

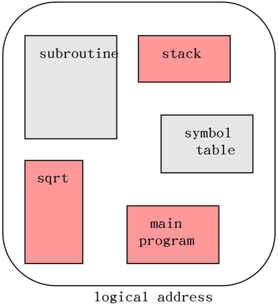
6. Segmentation (分段)
   - Basic Method
   - Hardware

Segmentation (分段)

- Segmentation: supporting user view of memory

- A program is a collection of segments.
  A segment is a logical unit such as:
  main program, procedure,
  function, method,
  object, local variables,
  global variables, common block,
  stack, symbol table,
  arrays

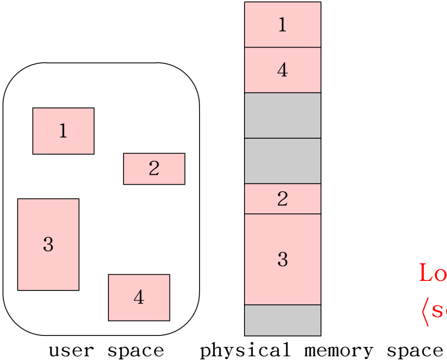
User's View of a Program



Logical address space

- A collection of segments, each segment $< name, \; length >$
  - 2-D address space
- A logical address consists of a two tuple
  - $< seg - name, \; offset >$, or
  - $< seg - num, \; offset >$
- Compiler automatically constructs segments reflecting the input program.
  - Pascal compiler
  - FORTRAN compiler
  - C compiler, such as gcc, ...

Logical View of Segmentation



- Each segment is a logically integrated unit.
- Each segment is of variable length.
- Elements within one segment is addressed from the beginning of the segment.

Logical address = ⟨segment#, offset⟩

Outline

6 Segmentation (分段)

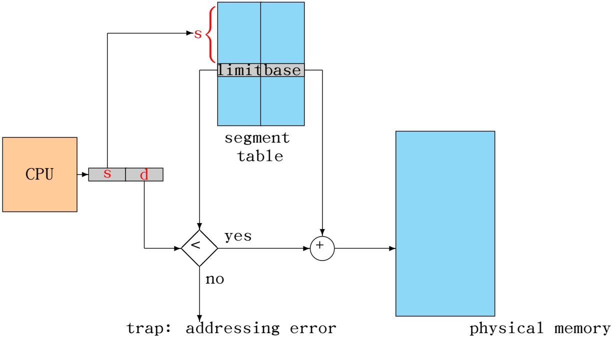
- Basic Method
- Hardware

Segmentation Architecture

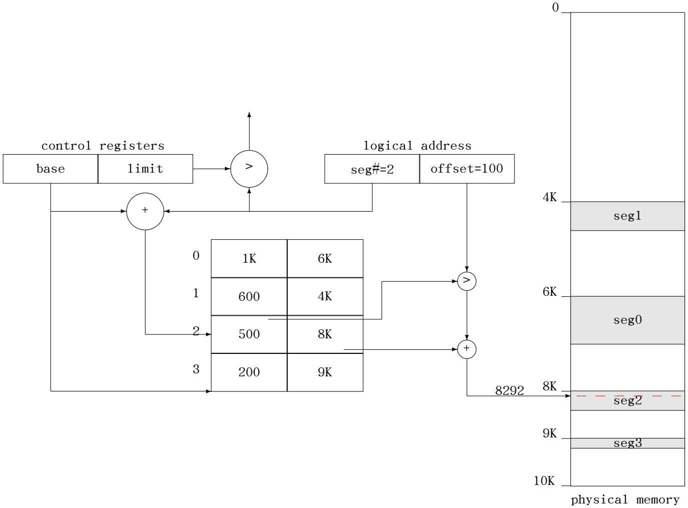
- Segmentation Architecture
  - Segment table(段表) — maps 2-D LA → 1-D PA; Each table entry has:
    1. Base — contains the starting physical address where the segments reside in memory
    2. Limit — specifies the length of the segment
  - Segment-table base register (STBR) points to the segment table’s location in memory
  - Segment-table length register (STLR) indicates number of segments used by a program;

    $$\textit{segment number } \boldsymbol{s} \textit{ is legal if } \boldsymbol{s} < \textit{STLR}$$

Segmentation Architecture

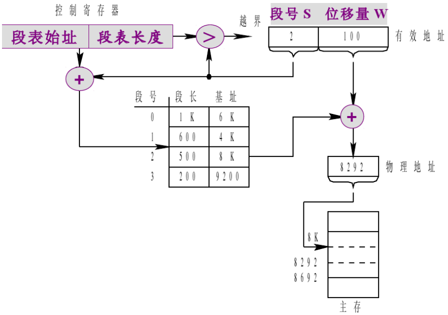


Segmentation Architecture



0	1K	6K
1	600	4K
2	500	8K
3	200	9K

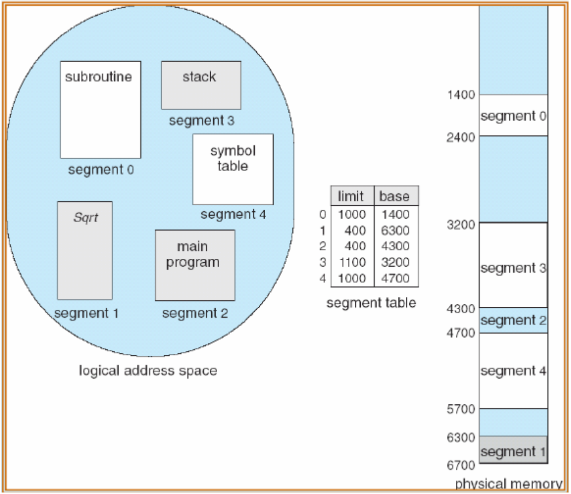
Segmentation Architecture



段号	段长	基址
0	1K	6K
1	600	4K
2	500	8K
3	200	9200

- Protection
  - With each entry in segment table associate:
    - Validation bit = 0 ⇒ illegal segment
    - Read/write/execute privileges
  - Protection bits associated with segments; Code sharing occurs at segment level
  - Since segments vary in length, memory allocation is a dynamic storage-allocation problem
    - First-fit, Best-fit, ...
    - Externel fragmentation, compaction, ...

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

- $<$ segment2, 53 $>\rightarrow$
  4300 + 53 = 4353
- $<$ setment3, 852 $>\rightarrow$
  3200 + 852 = 4052
- What about
  $<$ segment1, 536 $>$?

Differences between paging and segmentation

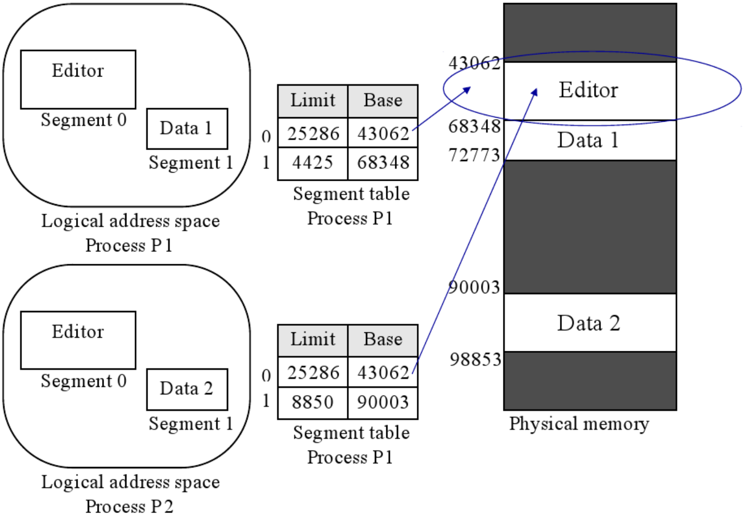
- Motivation and purpose
  - Paging：system-oriented, discrete physically, reduce external & internal fragmentation, memory utility↑
    - Page is the physical unit of infomation
  - Segmentation: user-oriented, discrete logically, satisfy the user’s need
    - Segment is the logical unit of information with relatively complete meaning
- Size
  - Paging: size is fixed, depends on hardware
  - Segmentation: size is not fixed, depends on the program and decided while compiling
- Dimension
  - Paging：1-D
  - Segmentation：2-D, segment name (number) + segment offset

Advantages of segmentation

1. Easy programming
    - Logically, easy to locate
    - Dynamic, by segment table
2. Sharing
    - Shared segments
    - Same segment number
3. Protection
    - Use segment table entry
    - Protection bit
        - Read-only, execute-only, read/write
        - Validation bit, 0=illegal segment
4. Dynamic linking
5. Growing dynamically (动态增长)

Protection & sharing & linking at segment level

Sharing



Editor

Segment 0

Data 1

Segment 1

Logical address space
Process P 1

	Limit	Base
0	25286	43062
1	4425	68348

Segment table
Process P 1

Editor

Segment 0

Data 2

Segment 1

Logical address space
Process P 2

	Limit	Base
0	25286	43062
1	8850	90003

Segment table
Process P 1

Physical memory

Outline

7 Segmentation with paging (段页式)

- Example: The Intel Pentium

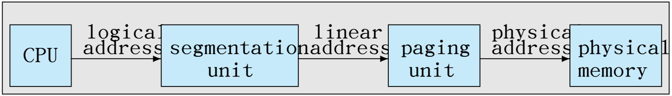
Outline

7 Segmentation with paging (段页式)

- Example: The Intel Pentium

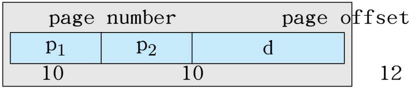
Example: The Intel Pentium

- Supports both pure segmentation & segmentation with paging



CPU → logical address → segmentation unit → linear address → paging unit → physical address → physical memory

1. CPU generates logical addresses
2. Logical address given to segmentation unit which produces linear addresses
3. Linear address given to paging unit which generates physical address in main memory
    - Paging units form equivalent of MMU



page number		page offset
p_1	p_2	d
10	10	12

Intel Pentium Segmentation

- Intel segmentation
  - Logical address = segment : offset
- 6 16-bits segment registers: cs, ss, ds, es, fs and gs
  - cs: code segment register
  - ss: stack segment register
  - ds: data segment register
- Since 80386, Intel microprocessors using two different address translation scheme
  1. Real-mode (实模式)(20-bits address space)
  2. Protection-mode (保护模式)(32-bits address space)

Intel Pentium Segmentation

1. Real-mode (实模式)(20-bits address space)
   - Segment registers store segment base addresses, but only 16 bits
     Therefore, segment base addresses must 4-bits aligned (example: 0xABCD0)

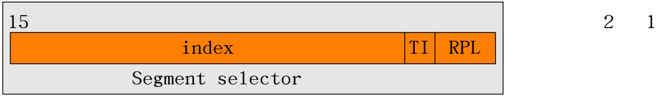
$$\begin{aligned} \text{physical address} &= \text{logical address} \\ &= \text{value in segment register} \times 16 + \text{offset} \end{aligned}$$

Intel Pentium Segmentation

2. Protection-mode (保护模式)(32-bits address space)
    - 16-bits segment registers + GDT/LDT
    - GDT/LDT and segment descriptor (段描述符)
        - Global descriptor table, GDT (全局描述符表)
        - Local descriptor table, LDT (局部描述符表): for process
        - GDT/LDT: One 8-bytes segment descriptor for each segment
        - GDT and LDT are also stored in memory
        - Registers GDTR and LDTR store the base address of a GDT and LDT, respectively

Intel Pentium Segmentation

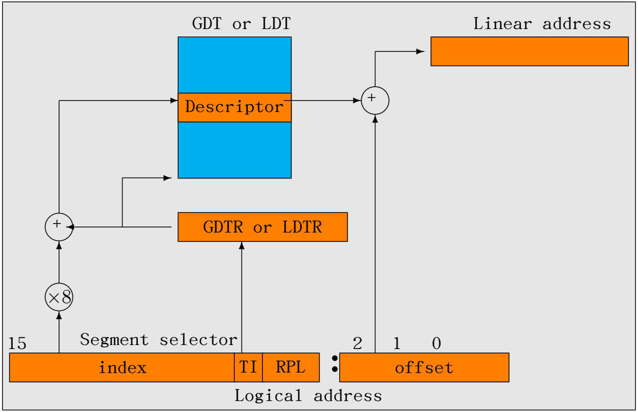
2. Protection-mode (保护模式)(32-bits address space)
   - Segment selector (段选择子): The value in segment register, 16-bits



1. Index: 13 bits, the index of corresponding segment descriptor in GDT/LDT
2. Table Indicator, TI-bit: 1 bit, GDT? LDT?
3. Request privilege level, RPL-bits: 2 bits

Intel Pentium Segmentation

2. Protection-mode (保护模式)(32-bits address space)
   - Linear address = segment base + offset



   - Linear address ⇒Physical address: paging or not

Intel Pentium Segmentation

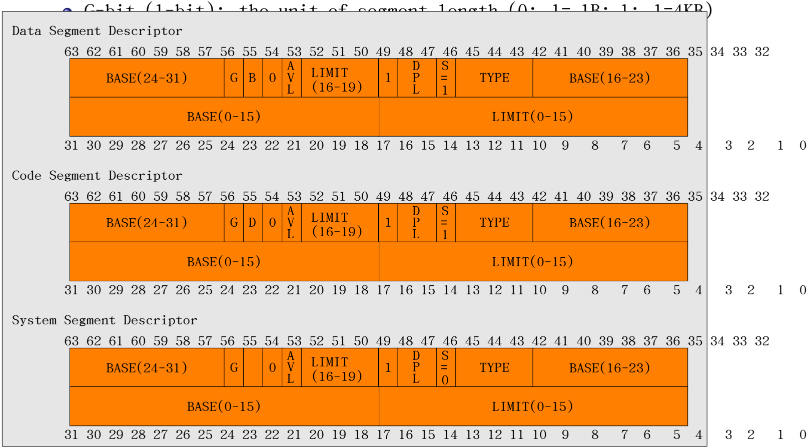
2. Protection-mode (保护模式)(32-bits address space)
   - Types of segment descriptors:
     1. Data Segment Descriptor (数据段描述符): for data/stack segments
     2. Code Segment Descriptor (代码段描述符): for code segments
     3. Task State Segment Descriptor (任务状态段描述符)
     4. LDT Descriptor (LDT描述符)
     5. System Segment Descriptor (系统段描述符)

Intel Pentium Segmentation

2. Protection-mode (保护模式)(32-bits address space)
   - Contents of segment descriptors:
     - Base (32-bits): Segment start address in physical memory
     - Limit (20-bits): for segment length
     - G-bit (1-bit): the unit of segment length (0: 1= 1B; 1: 1=4KB)
     - S-bit (1-bit): system segment (0) or not (1)
     - Type (4-bits): for code/data/tss/ldt/etc
     - DPL-bits (2-bits): descriptor privilege level of the segment (00b~11b)
     - Segment present flag (1-bit): present (1) or not (0)
     - ...

Intel Pentium Segmentation

2. Protection-mode (保护模式)(32-bits address space)
   - Contents of segment descriptors:
     - Base (32-bits): Segment start address in physical memory
     - Limit (20-bits): for segment length
     - G-bit (1-bit): the unit of segment length (0: 1= 1B; 1: 1=4KB)



Data Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE(24-31) | G | B | 0 | AVL | LIMIT (16-19) | 1 | DPL | S=1 | TYPE | BASE(16-23)

BASE(0-15) | LIMIT(0-15)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Code Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE(24-31) | G | D | 0 | AVL | LIMIT (16-19) | 1 | DPL | S=1 | TYPE | BASE(16-23)

BASE(0-15) | LIMIT(0-15)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

System Segment Descriptor

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE(24-31) | G | | 0 | AVL | LIMIT (16-19) | 1 | DPL | S=0 | TYPE | BASE(16-23)

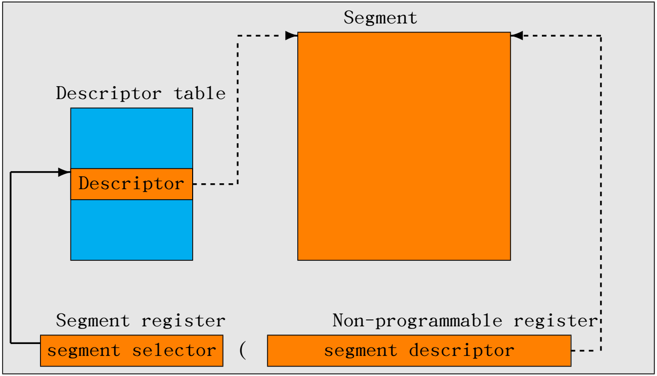
BASE(0-15) | LIMIT(0-15)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

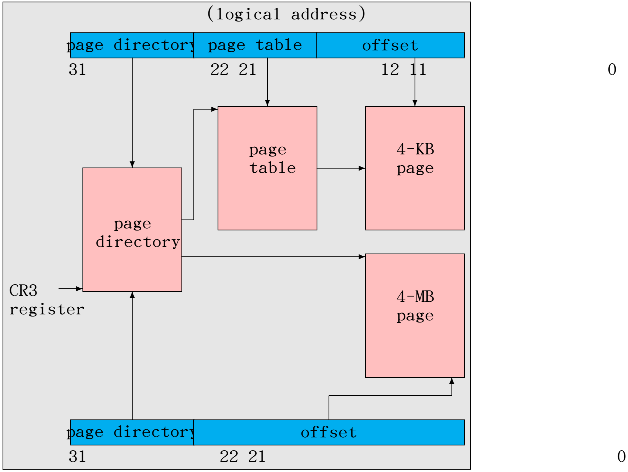
Intel Pentium Segmentation

2. Protection-mode (保护模式)(32-bits address space)

- Selector and the quick access to descriptor



Pentium Paging Architecture



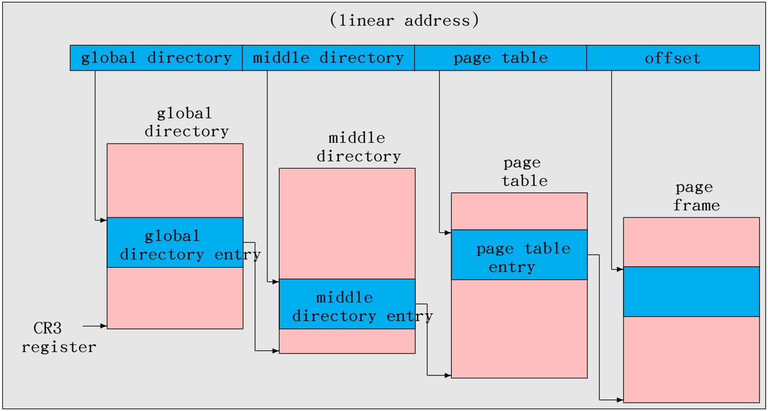
Linux on Pentium Systems

- Linux does not rely on segmentation and uses it minimally.
- Only 6 segments
  - __KERNEL_CS, __KERNEL_DS
  - __USER_CS, __USER_DS
    - shared by all processes
    - all processes use the same logical address
  - A Task-state segment (TSS)
  - A default LDT segment, shared by all processes, usually not used

(allow processes to create its own LDT replacing the default LDT)

Linear Address in Linux

- Linear address in Linux is broken into four parts with three-level paging



Outline

8 小结和作业

小结

1. background
   - Storage hierarchy
   - Memory protection
   - Program execution, loading & linking
2. Contiguous Memory Allocation (连续内存分配)
3. Swapping
   - Swapping (对换)
4. Paging (分页)
   - Basic Method
   - Hardware support
   - Memory Protection (内存保护)
   - Shared Pages (页共享)
5. Structure of the Page Table
   - Hierarchical Paging
   - Hashed Page Tables (哈希页表)
   - Inverted Page Tables (反置页表)
6. Segmentation (分段)
   - Basic Method
   - Hardware
7. Segmentation with paging (段页式)
   - Example: The Intel Pentium
8. 小结和作业

作业

- 参见课程主页

谢谢！