北京邮电大学软件学院

School Of software Engineering Of BUPT

Operating Systems

Lecture 7 : Memory Management

Jinpengchen

Email: jpchen@bupt.edu.cn

Catalog Description

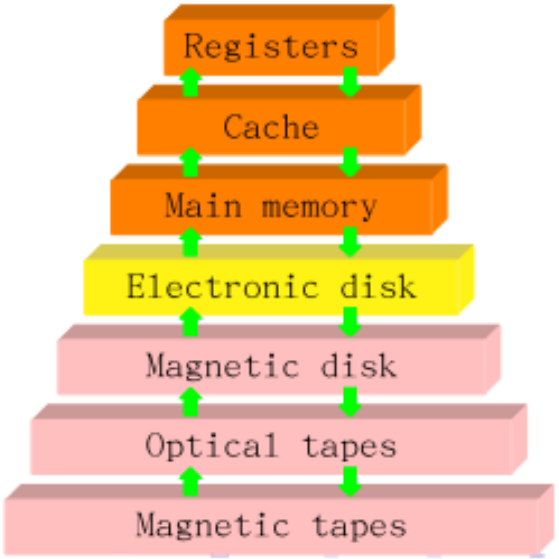
- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

Background

- 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



Background

- 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

Background

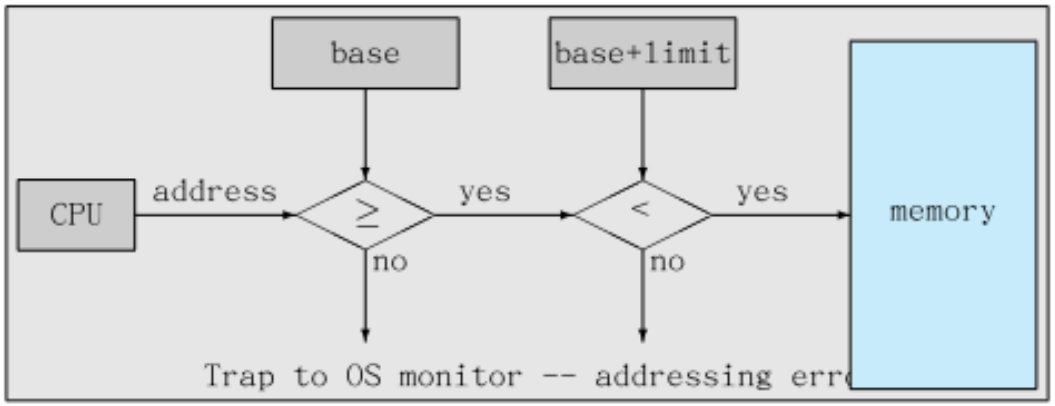
- Caching
  - Caching (高速缓存技术)
    - ✓ Copying information into faster storage system
    - ✓ When accessing, first check in the cache,
      if In: use it directly
      Not in: get from upper storage system, and leave a copy in the cache
  - Using of caching
    - ✓ Registers provide a high-speed cache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

Background

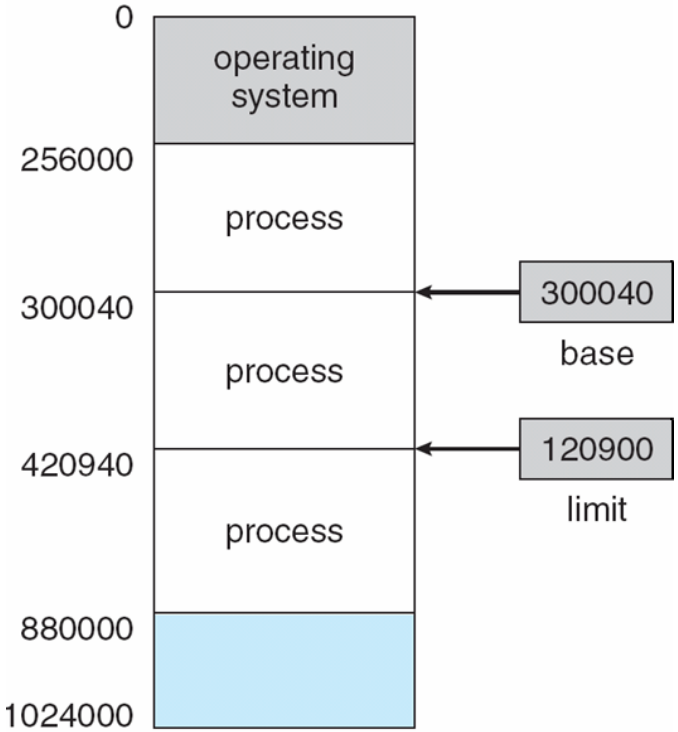
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles(several CPU clocks)
- Cache sits between main memory and CPU registers
- Protection of memory required to ensure correct operation (How?)

Background

- Memory protection
  - A pair of base and limit registers define the logical address space
    - ✓ 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



Hardware address protection base and limit register



Background

- Binding of Instructions and Data to Memory
  - Symbolic addresses to relocatable addresses
  - Relocatable addresses to absolute addresses
  - Address binding of instructions and data to memory addresses can happen at three different stages
    - ✓ Compile time:(absolute code)

      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

Background

- Binding of Instructions and Data to Memory
  - ✓ Load time: (relocatable code)

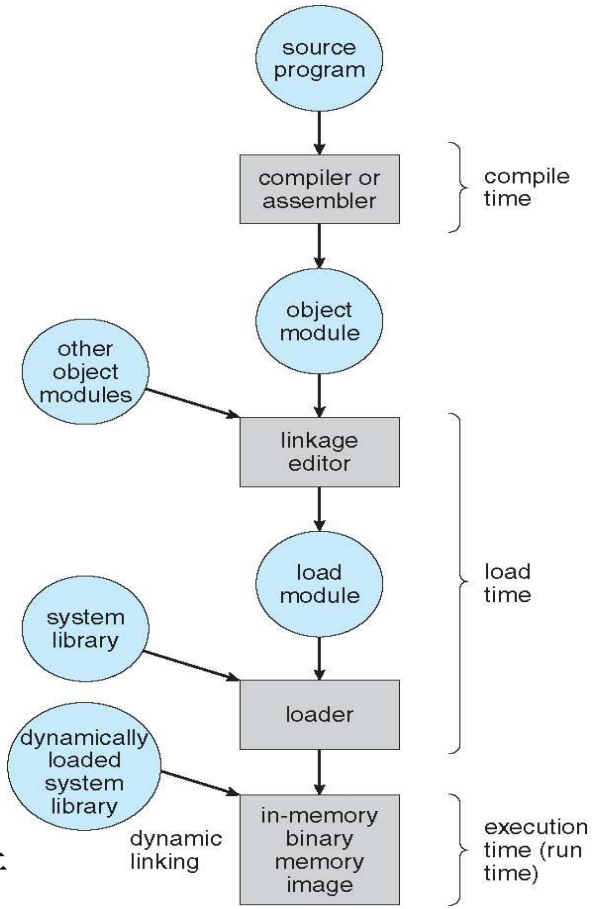
    Must generate relocatable code (可重定位代码) if memory location is not known at compile time
  - ✓ Execution time:

    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)

Background

- Multistep Processing of a User Program



.C

.OBJ

.EXE

.h/.dll

Loadlibrary
GetProcAddress

Windows作为DLL实现文件：

- ✓ Activex控件（.ocx）文件
- ✓ 控制面板（.cpl）文件
- ✓ 设备驱动程序（.drv）文件

……

Background

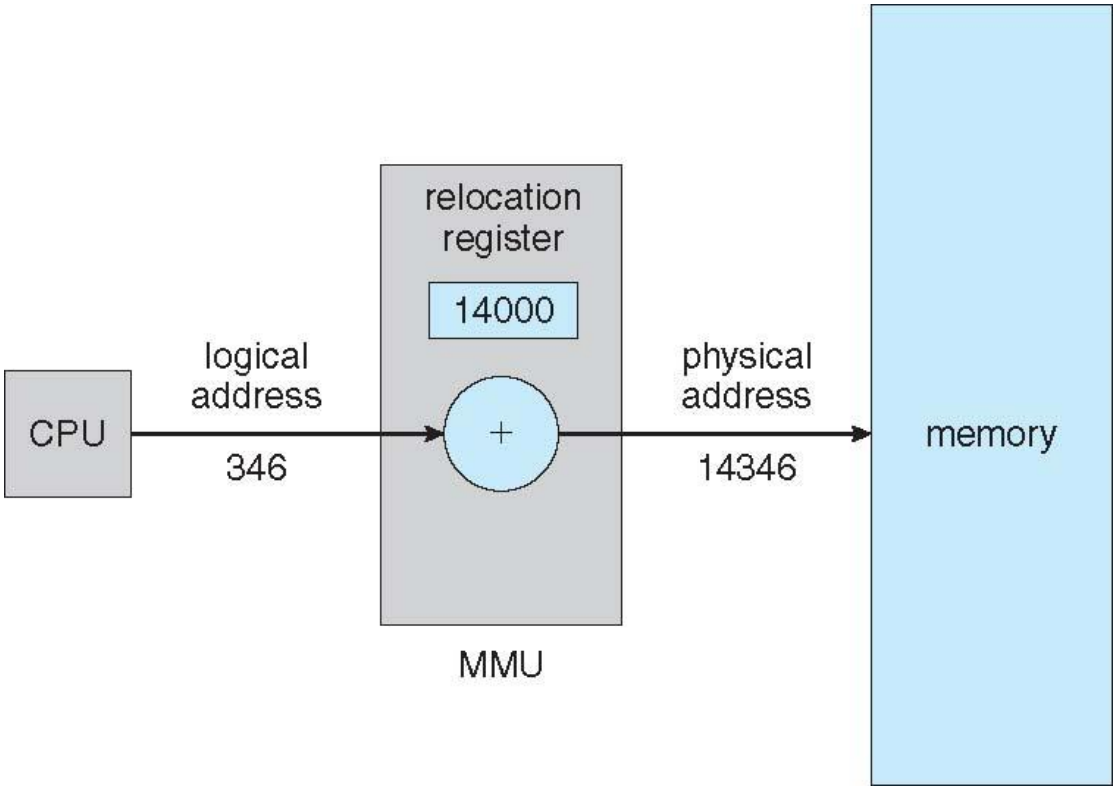
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.
- Logical address - generated by the CPU; also referred to as virtual address
- Physical address - address seen by the memory unit
- Logical and physical addresses are the same
  - ✓ in compile-time address-binding schemes
  - ✓ in load-time address-binding schemes;
- logical (virtual) and physical addresses differ
  - ✓ in execution-time address-binding schemes.

Background

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

Background

- Dynamic Relocation Using a Relocation Register



Background

- 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(覆盖)
    - ✓ Swapping
    - ✓ ...

Background

- Program loading
  - 3 modes
    - ✓ Absolute loading mode
    - ✓ Relocatable loading mode
    - ✓ Dynamic run-time loading

Background

- Absolute loading mode
  - 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)

Background

- 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:
    - ✓ Execution: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
    - ✓ Logical address = absolute address
  - Suitable for multiprogramming systems (多道系统)

Background

- 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.

Background

- Dynamic Loading (动态运行时装入方式)
  - 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

Background

- Program linking
  - source files —compiling→ object modules —linking→ loadable modules
  - 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)

Background

- 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 delimitated
  - Disadvantage:
    - ✓ Each program on a system must include a copy of required libraries(or at least required routines)

      Example: language libraries

Background

- load-time dynamic linking (装入时动态链接)
  - Linking while loading:
    - ✓ External calls and references are delimitated

      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

Background

- 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 and less memory space

Catalog Description

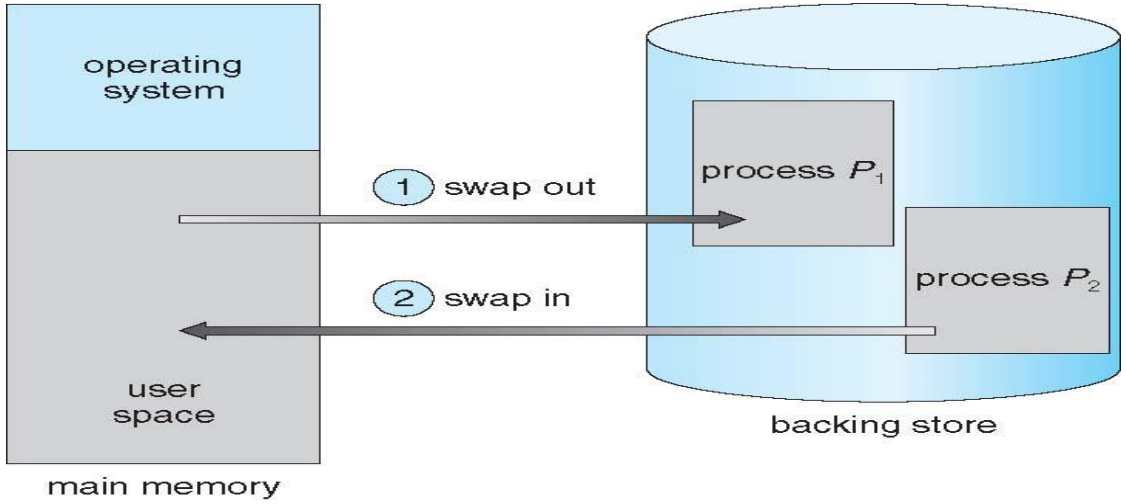
- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

Swapping

- 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
  - Unit of swapping:
    - ✓ Process: whole swapping; process swapping
    - ✓ Page, segment: partly swapping
  - Swapping requires:
    - ✓ Management of backing store (对换空间)
    - ✓ Swap out (or roll out)
    - ✓ Swap in (or roll in)

Swapping

- Swapping (对换)
  - 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



Swapping

- Swapping (对换)
  - 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

- Swapping (对换)
  - 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
  - Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
  - Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

Catalog Description

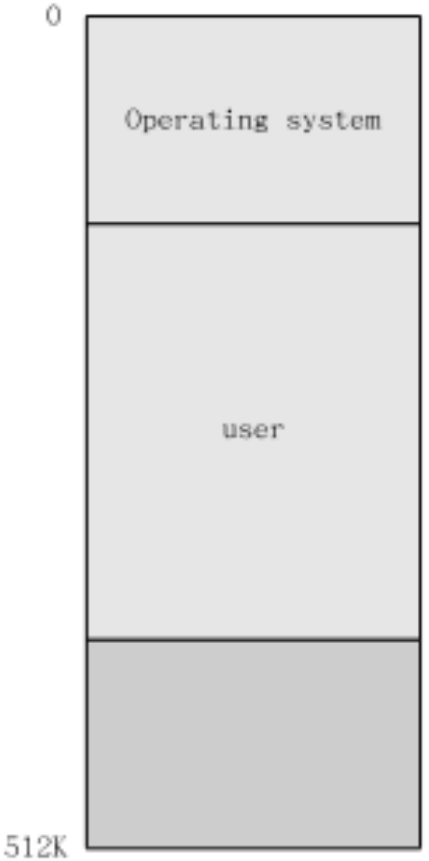
- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

Contiguous Memory Allocation

- Contiguous Memory Allocation （连续内存分配）
  - Each process is contained in a single contiguous section of memory
  - Monoprogramming memory allocation (单一连续)
  - Multiple-partition allocation
    - ✓ 固定分区
    - ✓ 动态分区

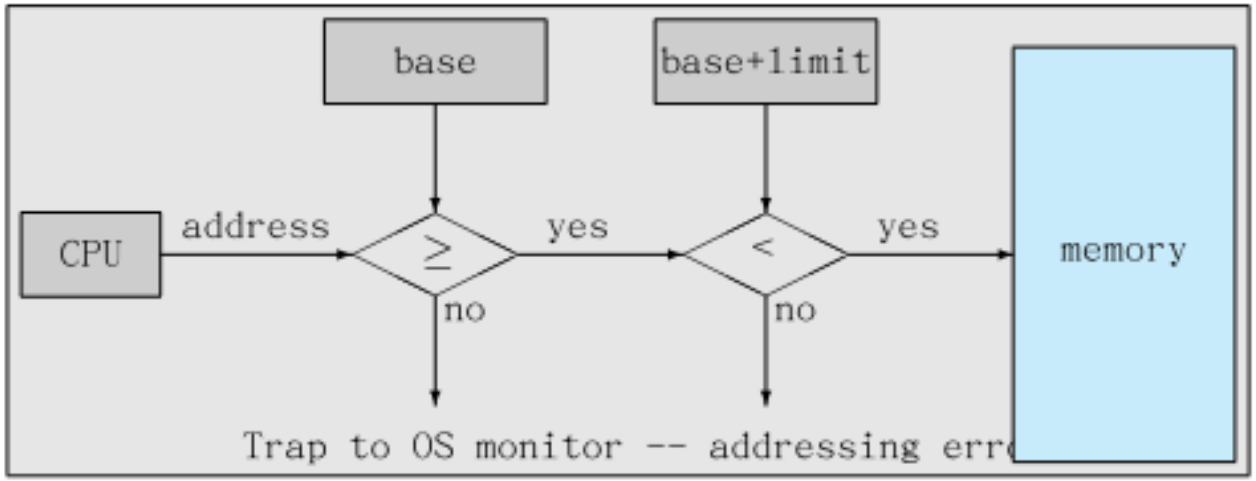
Contiguous Memory Allocation

- Contiguous Memory Allocation （连续内存分配）
  - Monoprogramming memory allocation (单一连续)
    - ✓ 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



Contiguous Memory Allocation

- Contiguous Memory Allocation （连续内存分配）
  - Monoprogramming memory allocation （单一连续）
    - ✓ Memory protection scheme
      - Use MMU, for example



      - May not use any protection

Contiguous Memory Allocation

- Contiguous Memory Allocation (连续内存分配)
  - 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

      1) When a partition is free, a process in input queue is selected and loaded into the free partition

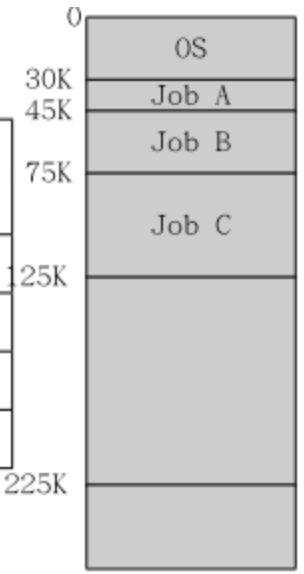
      2) When a process terminates, the partition becomes available for another process

      - The degree of multiprogramming (多道程序度) is bound by the number of partitions.
  - Fixed-partition (固定分区)
  - Dynamic-partition (动态分区)

Contiguous Memory Allocation

- Contiguous Memory Allocation (连续内存分配)
  - 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



Contiguous Memory Allocation

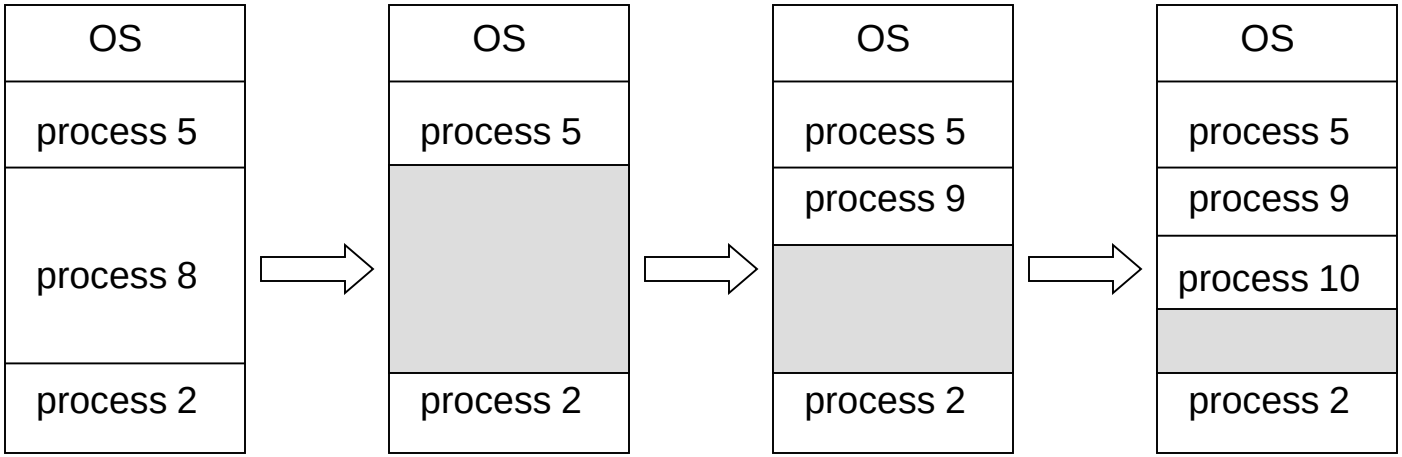
- Fixed-sized-partition scheme (固定分区)
  - Disadvantage
    - ✓ Poor memory utility
    - ✓ Internal fragmentation
      - Internal Fragmentation (内部碎片)

      Allocated memory may be slightly larger than requested memory;

      this size difference is memory internal to a partition, but not being used

Contiguous Memory Allocation

- 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.

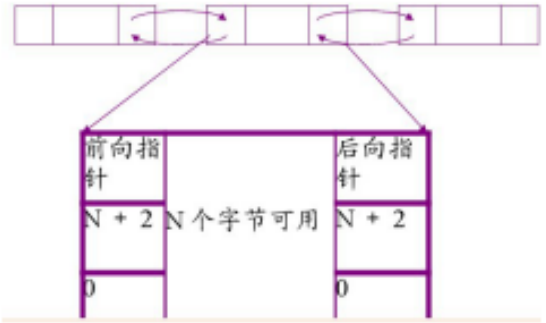


Contiguous Memory Allocation

- Dynamic partition scheme（动态分区）
  - OS maintains information about:
    - ✓ Allocated partitions
    - ✓ Free partitions (hole)
  - Example:
    - ✓ Free partitions table: need extra memory to store the table

Partition number	partition size	start address	state

- ✓ Free partitions list: can make use of the free partitions to store links and partition information



Contiguous Memory Allocation

- 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
    - ✓ Next-Fit（循环首次适应）: Allocate the next hole that is big enough
    - ✓ 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 are better than worst-fit in terms of speed and storage utilization

Contiguous Memory Allocation

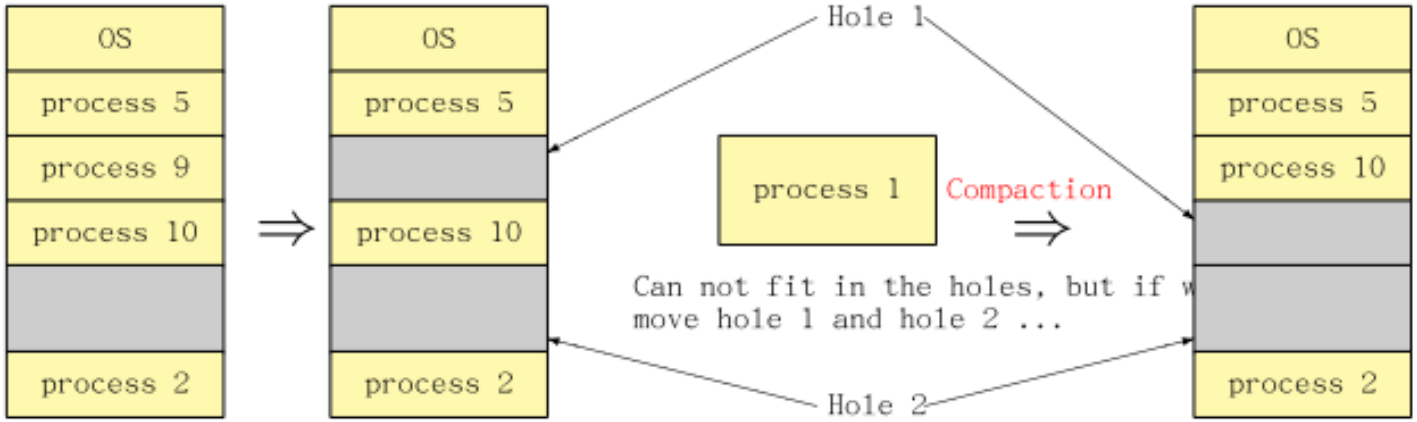
- For job A, 18 K, allocation comparing

Contiguous Memory Allocation

- Dynamic partition scheme
  - Advantage
    - ✓ 分区的个数、位置和大小都是随进程的进出而动态变化，非常灵活，避免了在固定分区中因分区大小不当所造成的的内碎片，提高了内存利用率
  - 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
  - External Fragmentation(外部碎片)
    - ✓ total memory space exists to satisfy a request, but it is not contiguous
    - ✓ 50-percent rule, given N allocated blocks, another 0.5 N blocks

Contiguous Memory Allocation

- Fragmentation
  - 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)

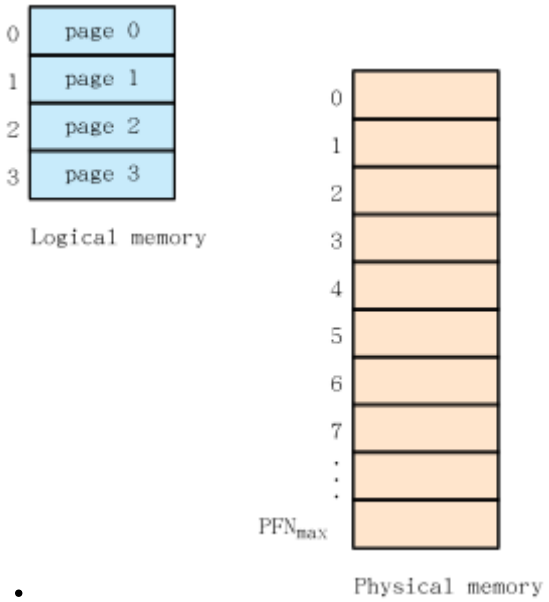


Catalog Description

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

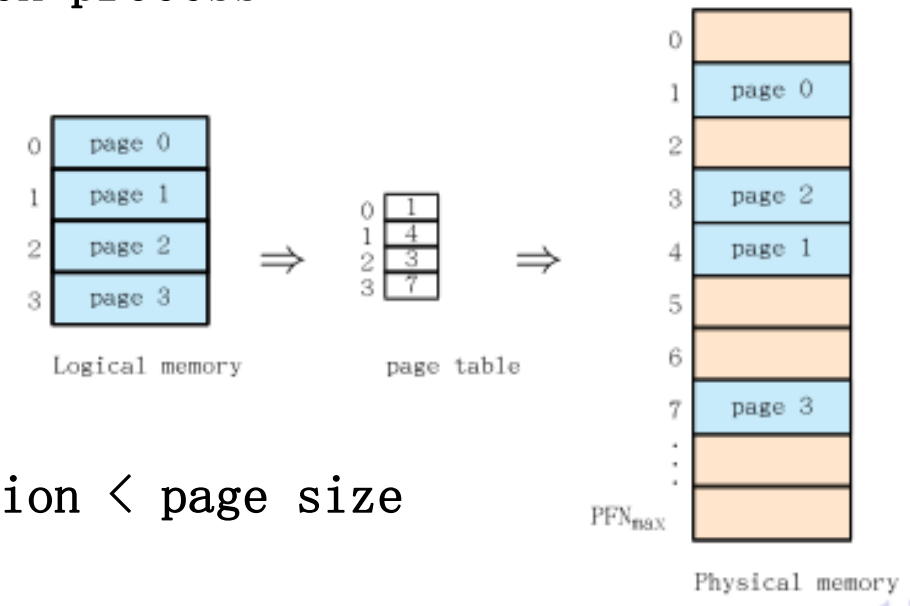
Paging

- LAS of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Basic Method
  - 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, ..., PFNmax
  - 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, ..., LFNmax



Paging

- Basic Method
  - Need hardware and software support for paging
    - ✓ Keep track of all free frames
    - ✓ To run a program of size n pages, need to find n free frames and load program
    - ✓ Set up a page table to translate logical to physical addresses for each process



  - Internal fragmentation < page size

Paging

- Address Translation Scheme
  - Page number (p),LFN
  - Page offset (d)
  - How to get p and d?

Paging

- Address Translation Scheme
  - Page number (p),LFN
  - Page offset (d)
  - How to get p and d?
    - ✓ Let

      A: An address, either logical address or physical address

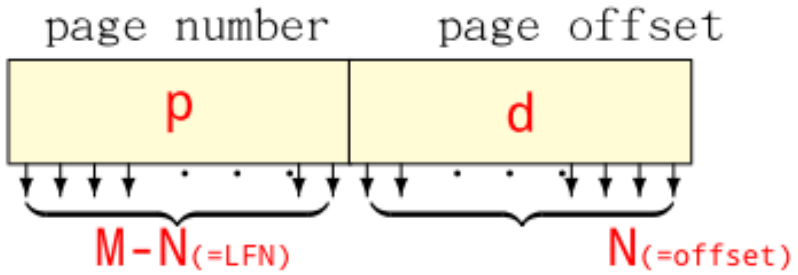
      L: The size of a page or page frame

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

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

Paging

- How to get p and d?
  - ✓ Suppose L = 2^N :
    - p = A right_shift N, 即A的高(M-N)位
    - d = A的低端N位



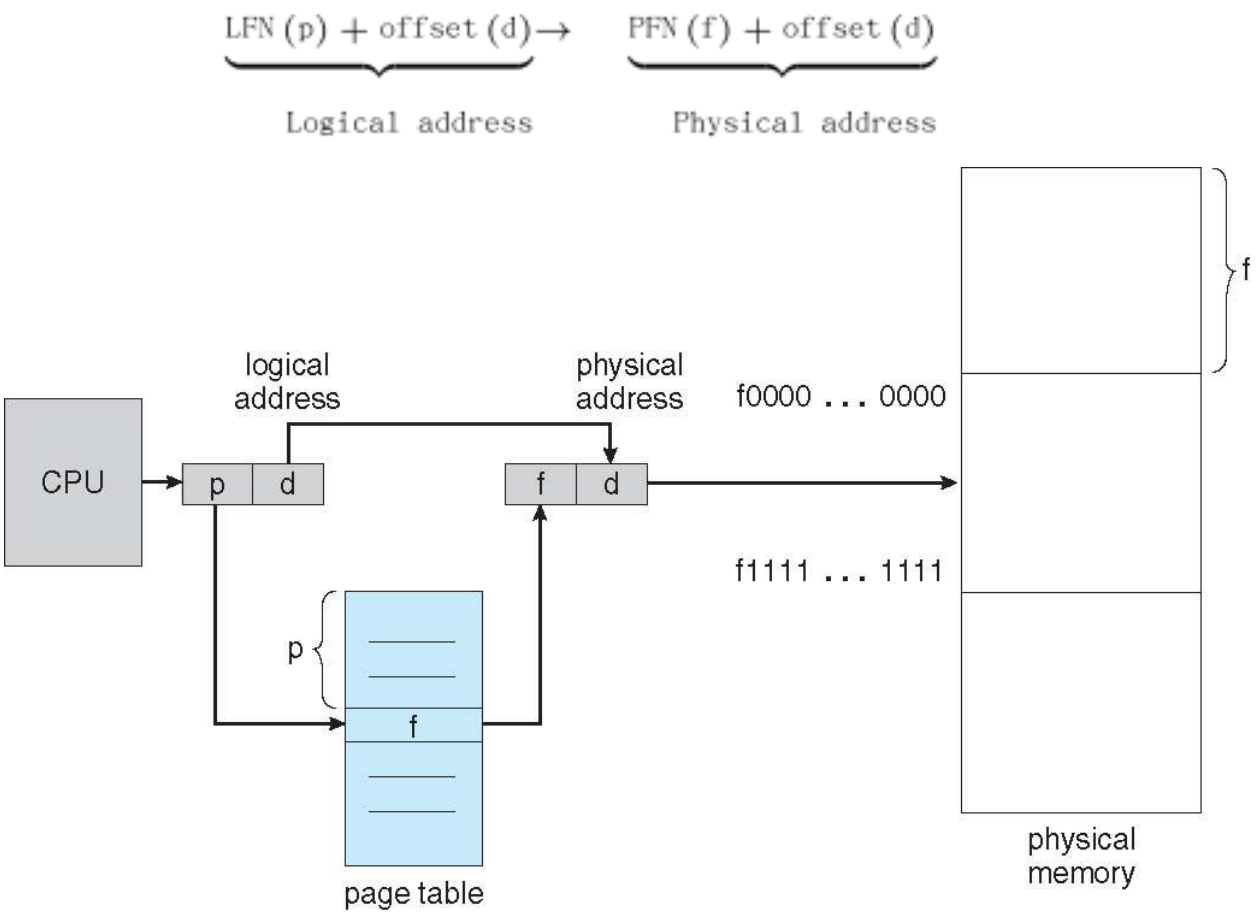
For given logical address space 2^m and page size 2^n

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

Example: A = 0x 1 2 3 4 5 6 7 8
p d

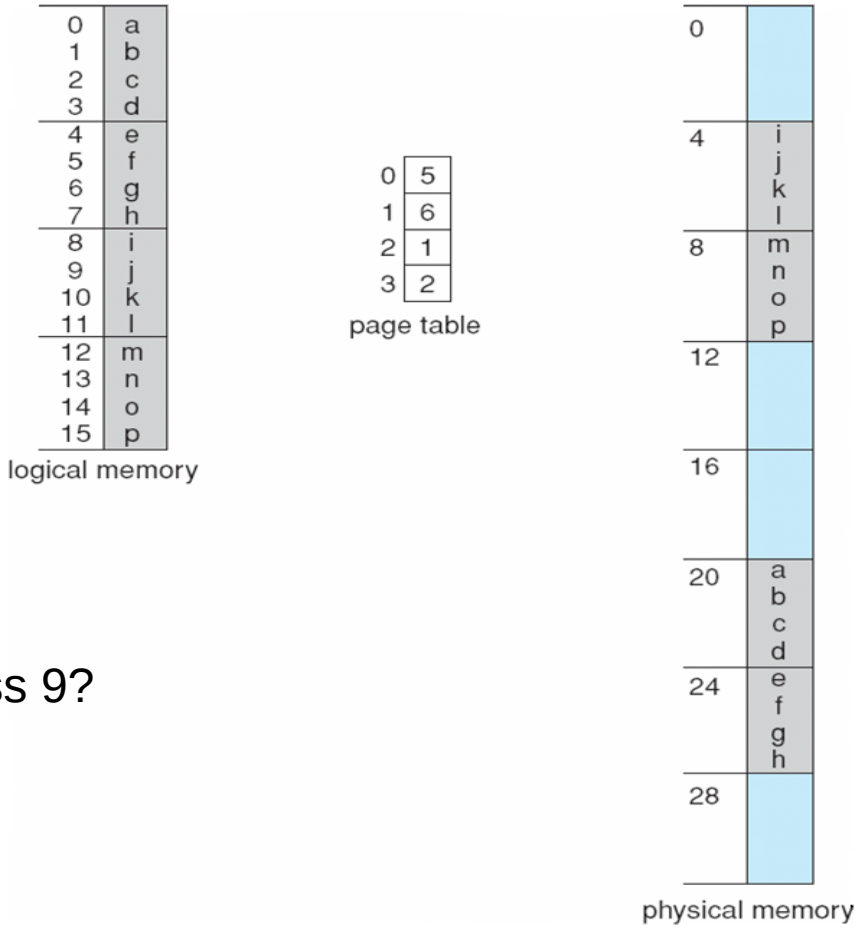
Paging

- Paging Hardware
  - ✓



Paging

- Paging Example



How to read logical address 9?

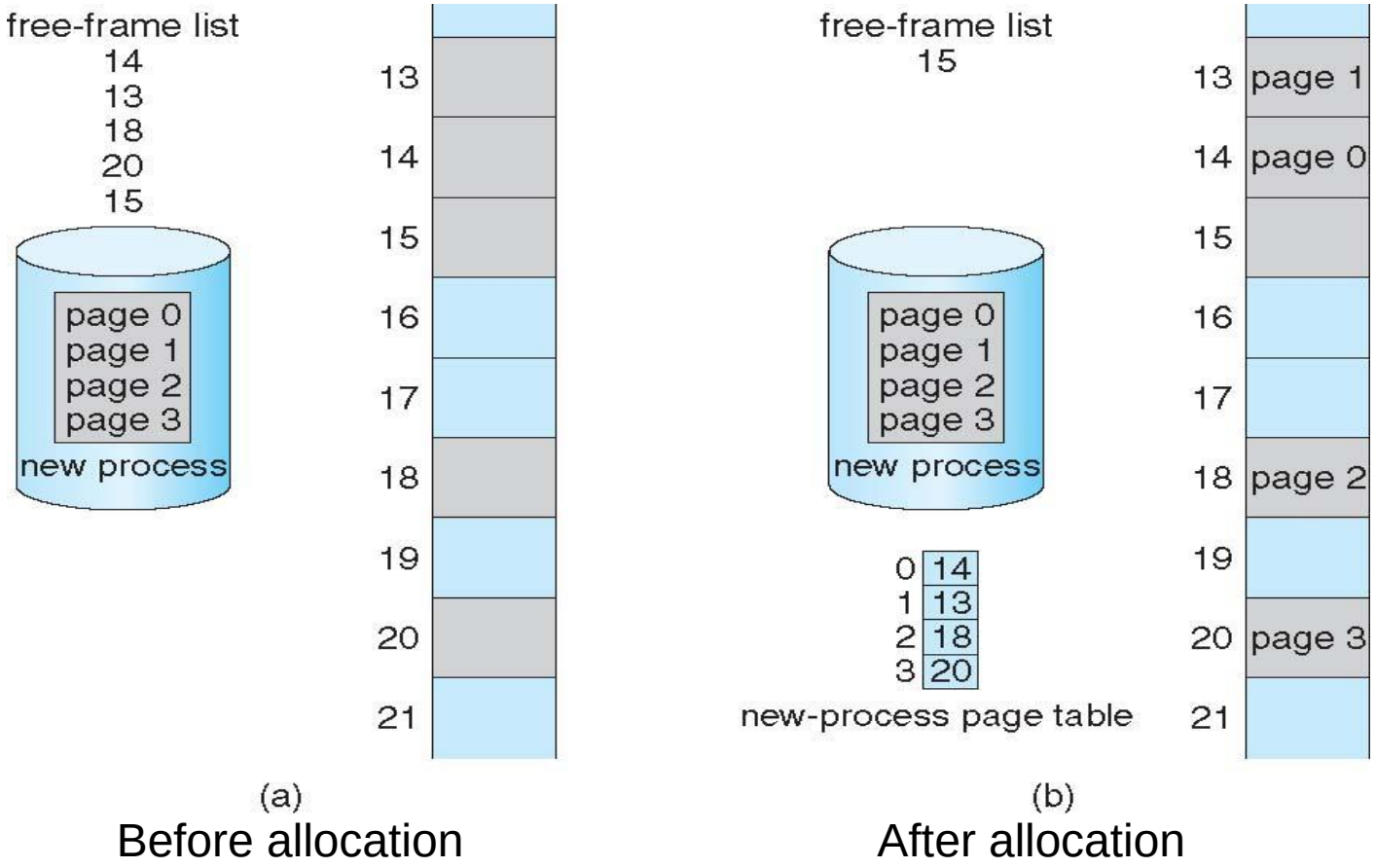
32-byte memory and 4-byte pages

Paging

- 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

Paging

- Free Frames
  - ✓ Frame table: one entry for each physical page frame

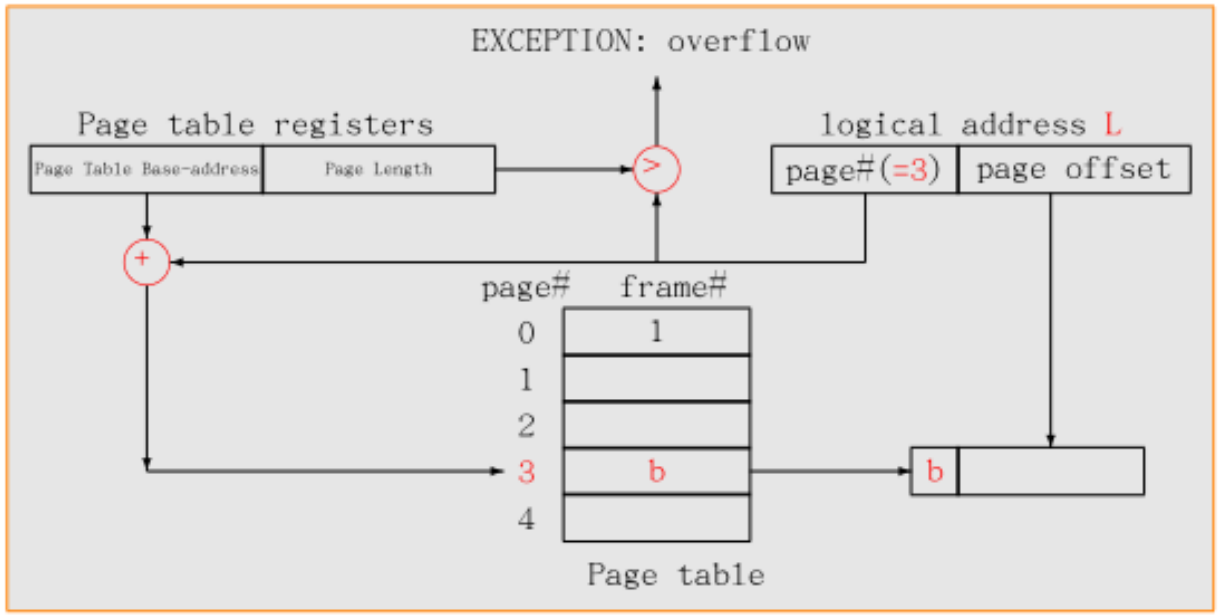


Implementation of Page Table

- Hardware support
  - Special hardware (software) is needed to implement page table
    - ✓ Basic paging hardware
    - ✓ Paging hardware with TLB

Implementation of Page Table

- Hardware support
  - 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

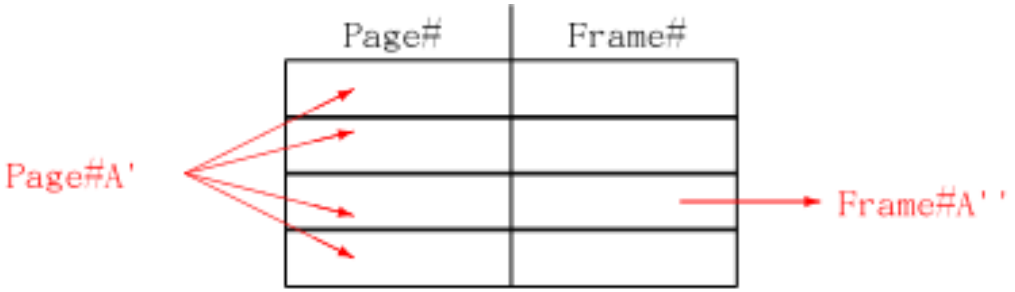


Implementation of Page Table

- Hardware support
  - Basic paging hardware
    - ✓ Context switch?
      - Each process is associated with a page table.
      - Page table must be switched, too.
    - ✓ Effective memory-Access Time (EAT, 有效访问时间)
      - Every data/instruction access requires two memory accesses.
        One for the page table
        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)

Implementation of Page Table

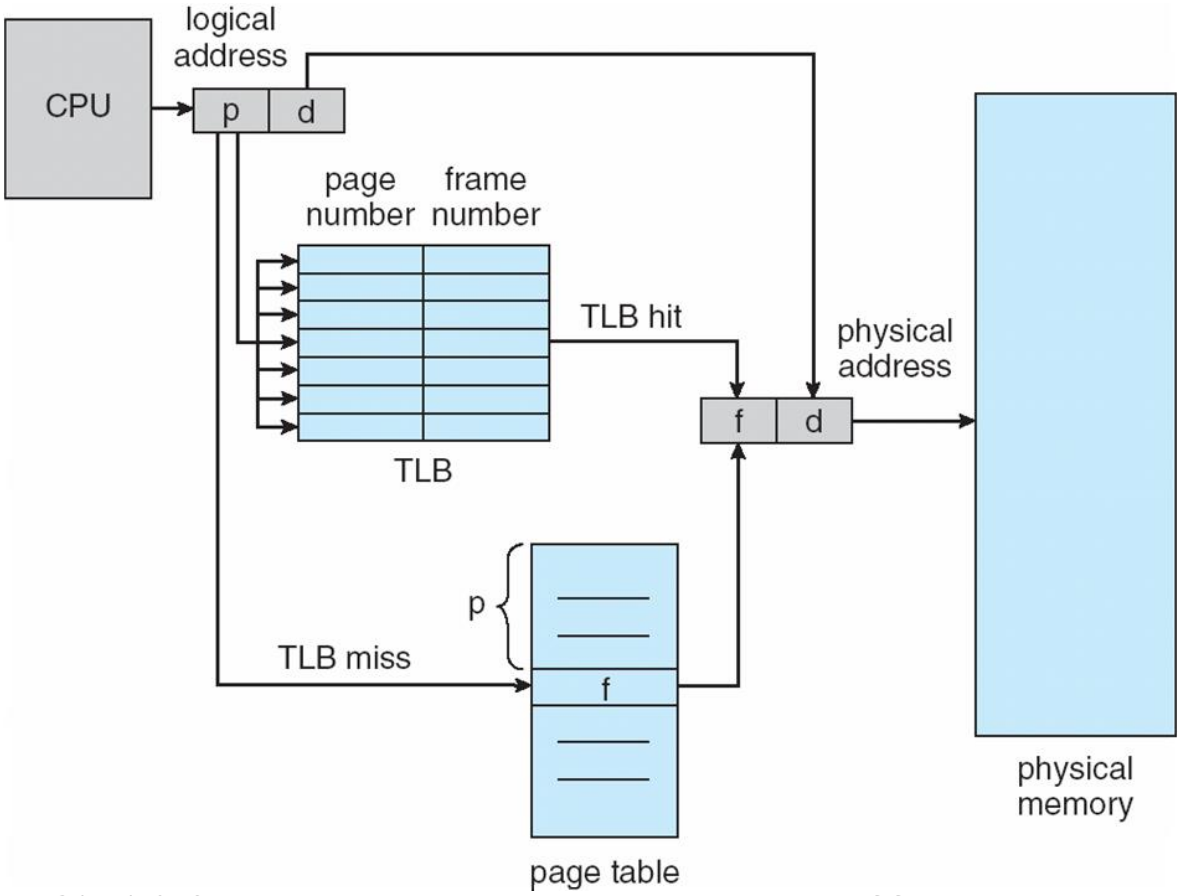
- Hardware support
  - Basic Paging Hardware With TLB
    - ✓ Associative Memory
      - Each register: a key & a value
      - Parallel search (high speed)
      - Expensive, typically 8~2048 entries



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

Implementation of Page Table

- Hardware support
  - Basic Paging Hardware With TLB



✓ Context Switch?

- TLB must be flushed after context is switched!

Implementation of Page Table

- Basic Paging Hardware With TLB
  - 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
  - What will be happened after context is switched?
  - TLB replacement algorithm

Implementation of Page Table

- Basic Paging Hardware With TLB
  - 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)

      EAT = $(t + \epsilon)\alpha + (2t + \epsilon)(1-\alpha) = 2t + \epsilon - t\alpha$
  - If ϵ = 20ns, t = 100ns, $\alpha 1$ = 80%, $\alpha 2$ = 98%:

    If TLB hit: 20 + 100 = 120ns

    If TLB miss: 20 + 100 + 100 = 220ns

    EAT1 = 120 * 0.8+ 220 * 0.2 = 140ns

    EAT2 = 120 * 0.98 + 220 * 0.02 = 122ns

Implementation of Page Table

- Memory Protection
  - 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

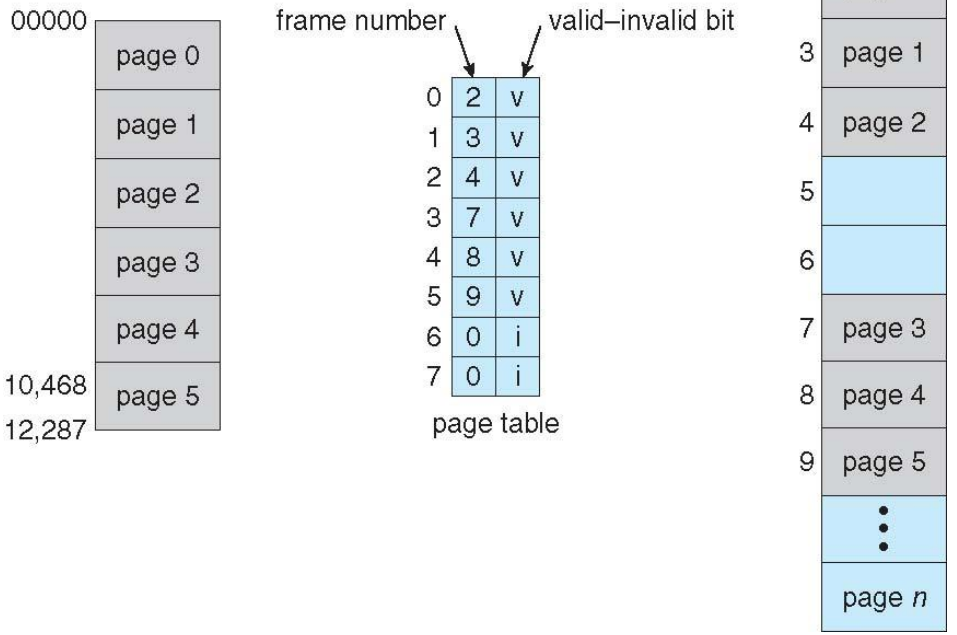
Implementation of Page Table

- Memory Protection
  - 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

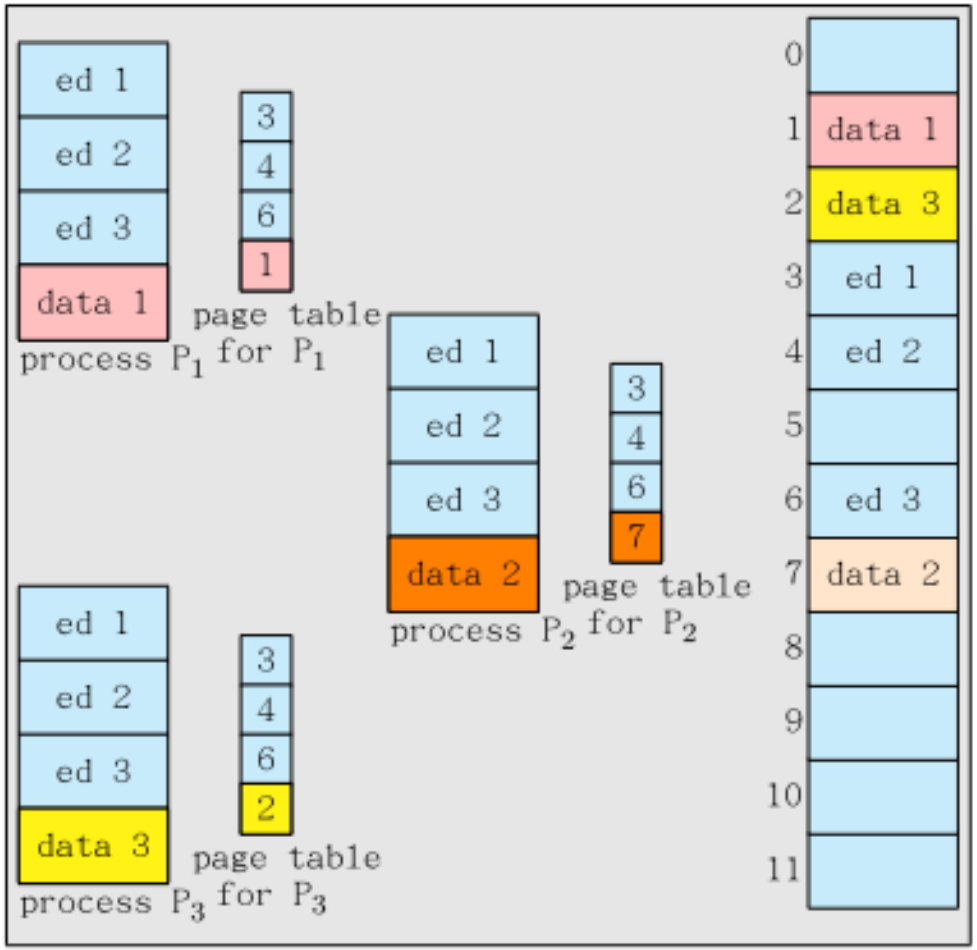


Implementation of Page Table

- 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.
  - 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.

Implementation of Page Table

- Shared Pages Example

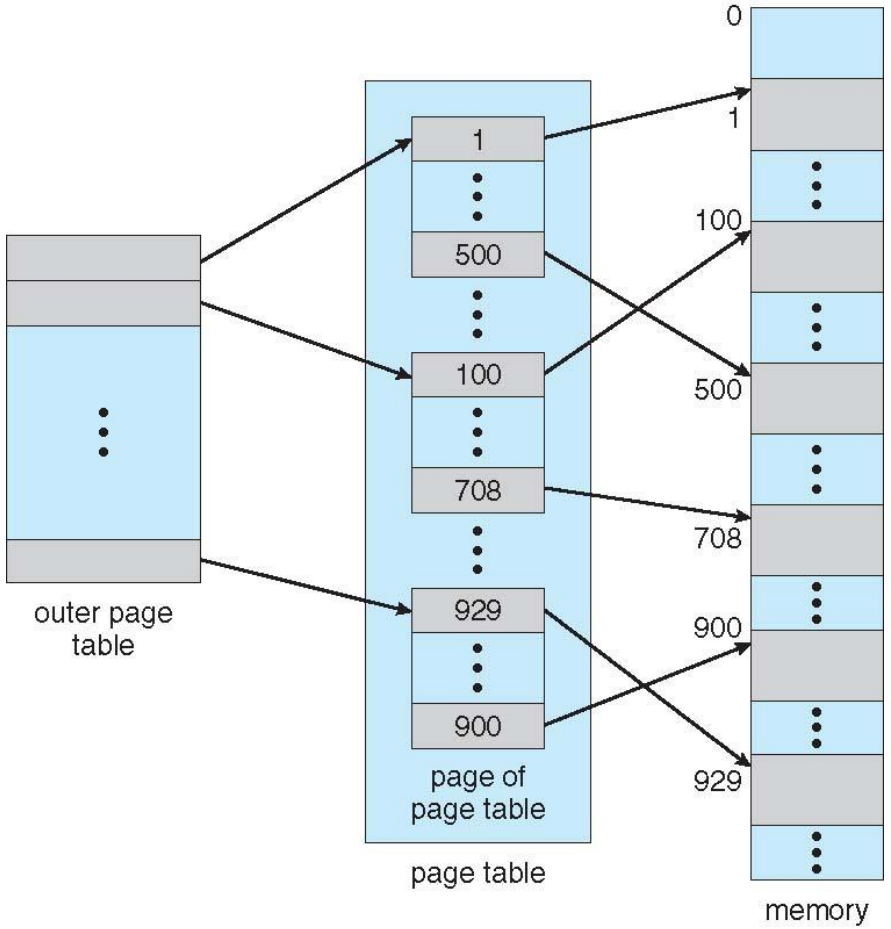


Catalog Description

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

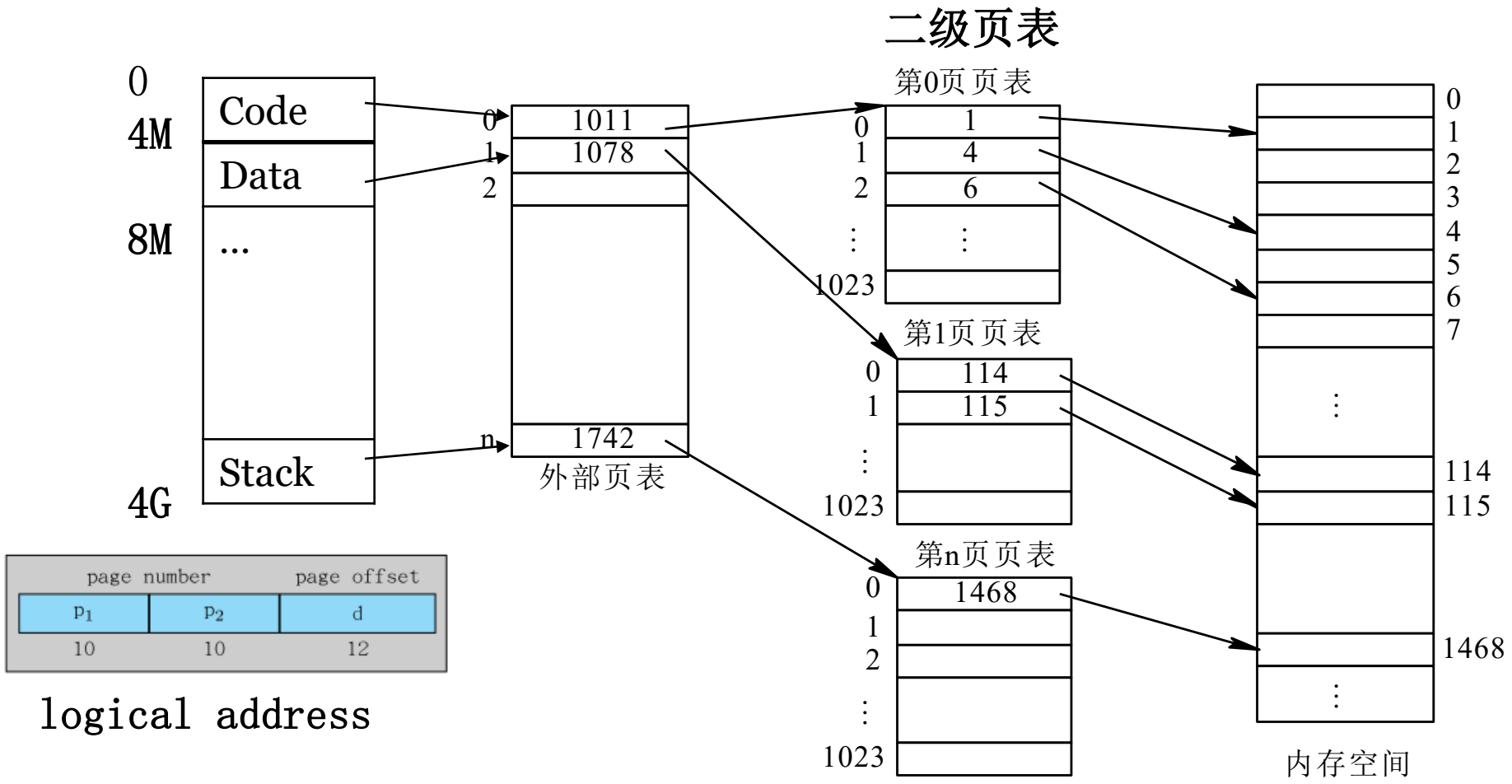
Structure of the Page Table

- Hierarchical Paging
  - Break up the logical address space(LAS) into multiple page tables
    - ✓ Need directories
    - ✓ A simple technique is a two-level page table



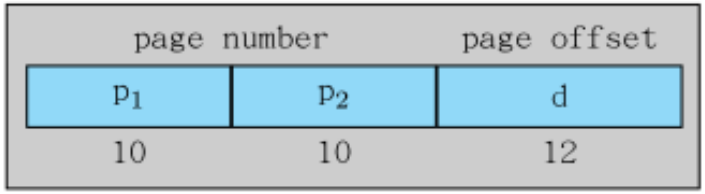
Structure of the Page Table

- Two-Level Paging Scheme



Structure of the Page Table

- 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:



Where p1 is an index into the outer page table, and p2 is the displacement within the page of the outer page table

Structure of the Page Table

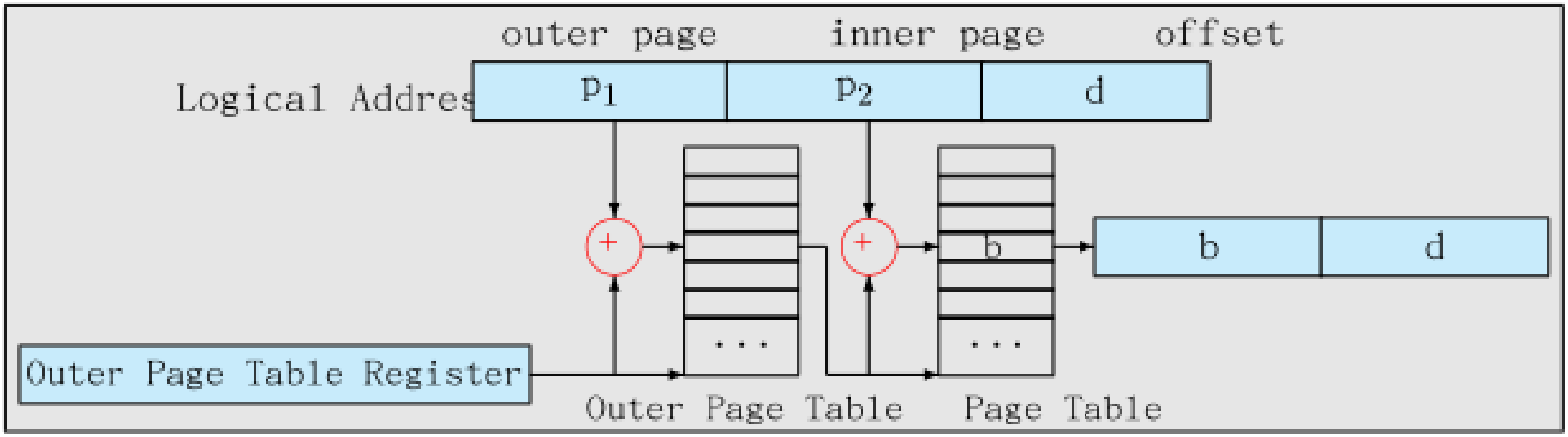
- Two-Level Paging Scheme
  - Example

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

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

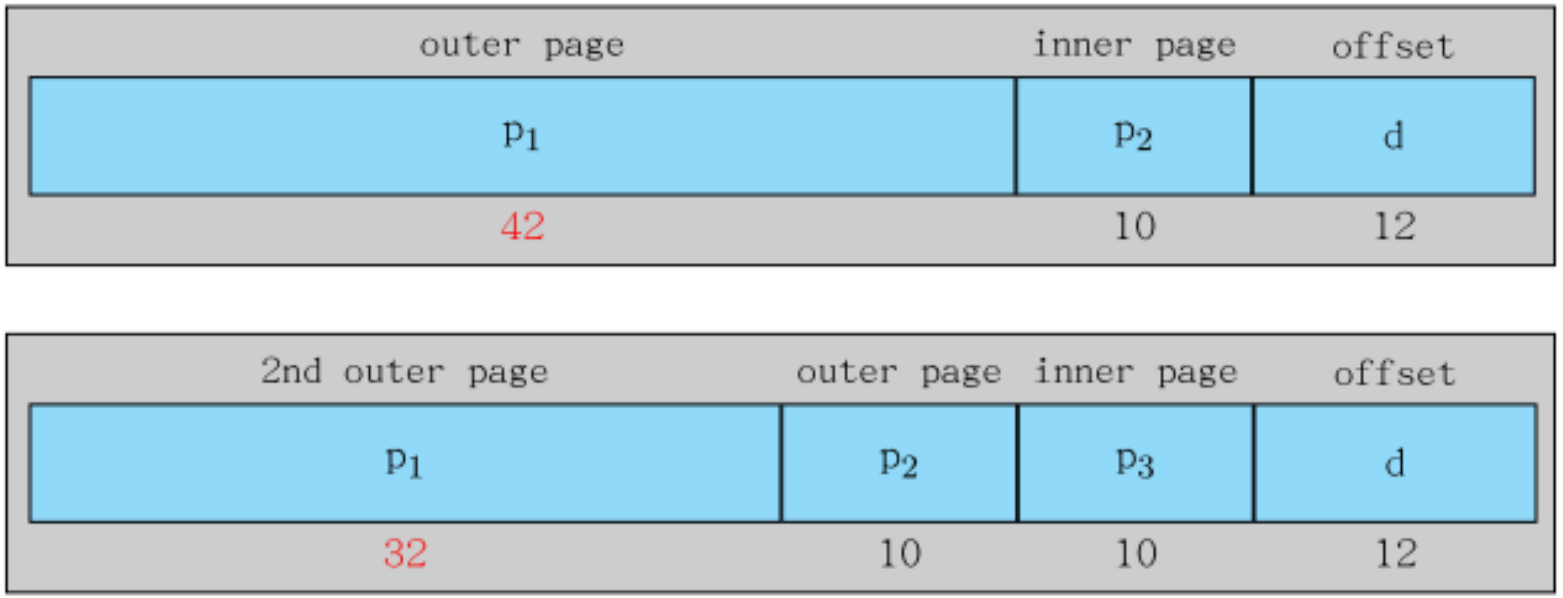
Structure of the Page Table

- Two-Level Paging Scheme
  - Address-Translation Scheme



Structure of the Page Table

- Three-level Paging Scheme

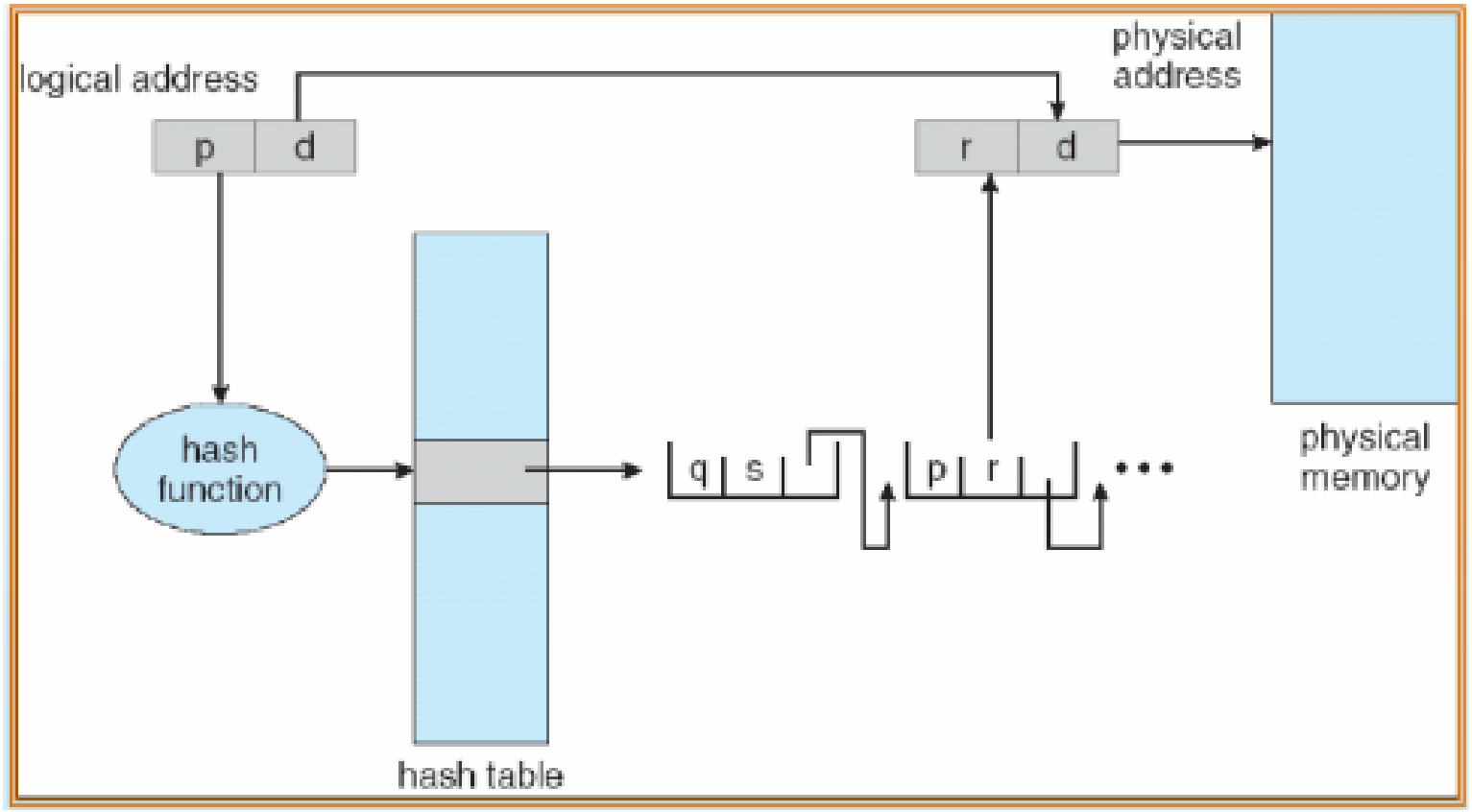


Structure of the Page Table

- Hashed Page Tables (哈希页表)
  - Common in address spaces > 32 bits
  - Hash table contains a linked list of element
  - Each element consists:
    - ✓ The virtual page No.
    - ✓ The value of the mapped page frame
    - ✓ A pointer to the next element in the linked list
  - Algorithm
    - ✓ 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.

Structure of the Page Table

- Hashed Page Tables (哈希页表)

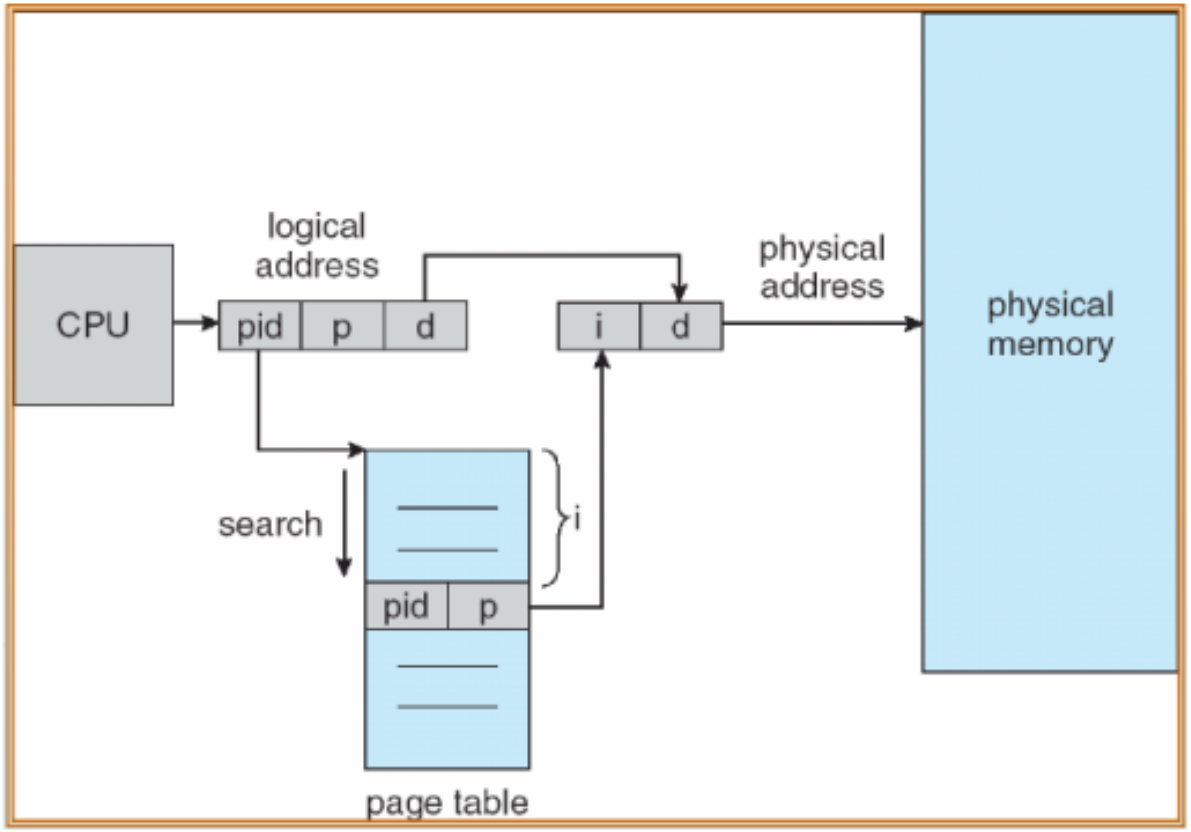


Structure of the Page Table

- Inverted Page Tables (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.
  - Benefits
    - ✓ Decreases memory needed to store each page table
    - ✓ Only one page table in the system
  - Drawbacks
    - ✓ increases time needed to search the table when a page reference occurs.
    - ✓ Use hash table to limit the search to one

Structure of the Page Table

- Inverted Page Tables (反置页表)
  - Inverted Page Table Architecture



Catalog Description

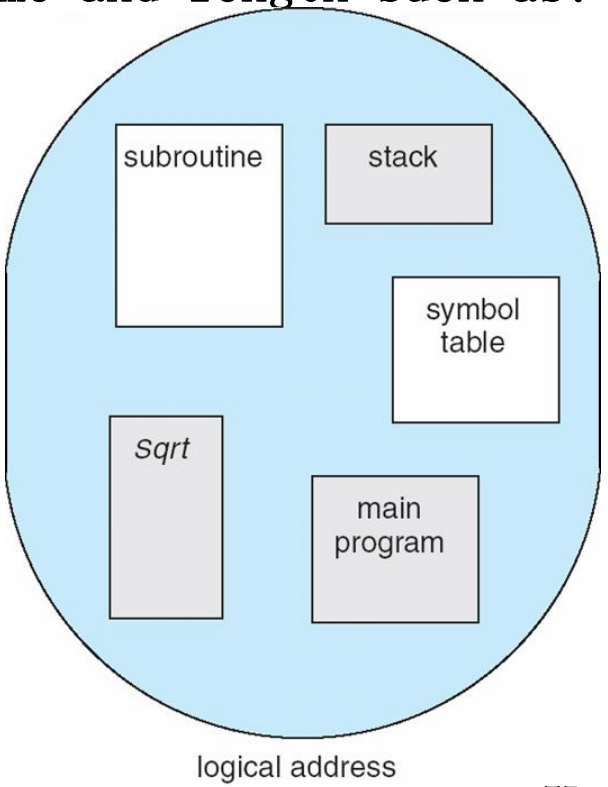
- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation

Segmentation

- Segmentation(分段)
  - Memory-management scheme that supports user view of memory
  - A program is a collection of segments
  - A segment is a logical unit, with name and length such as:

main program
procedure
function
method
object
local variables
global variables
common block
stack
symbol table
arrays

User's View of a Program

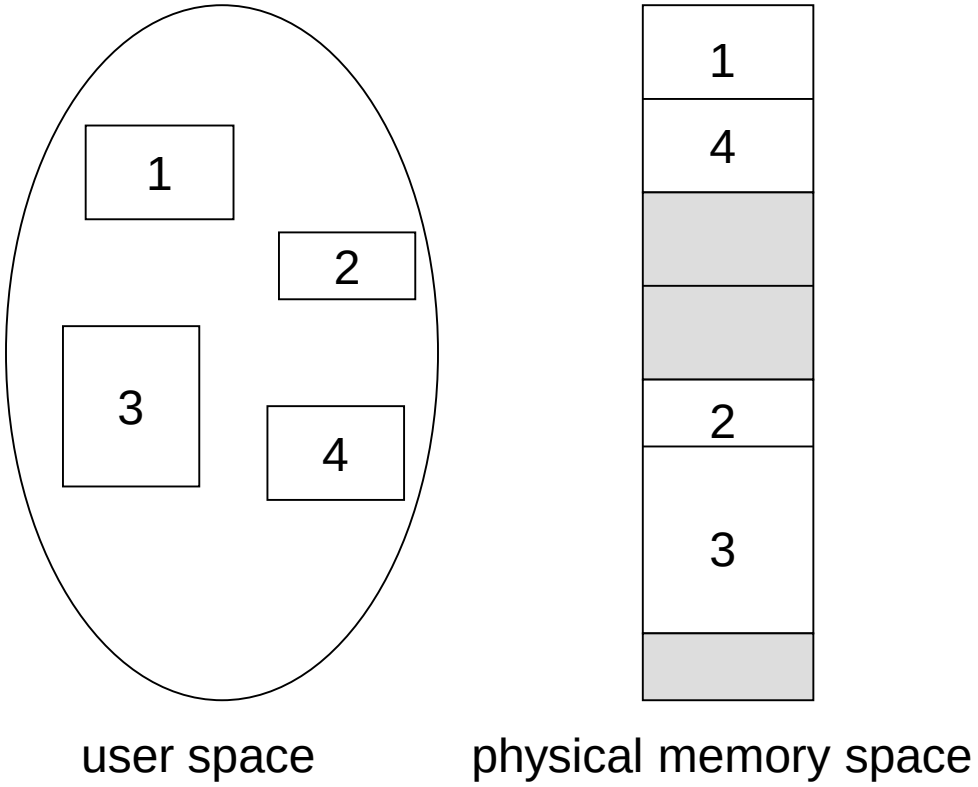


Segmentation

- Segmentation(分段)
  - 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 >

Segmentation

- Segmentation(分段)
  - Logical View of Segmentation



user space　　physical memory space

✓ 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.

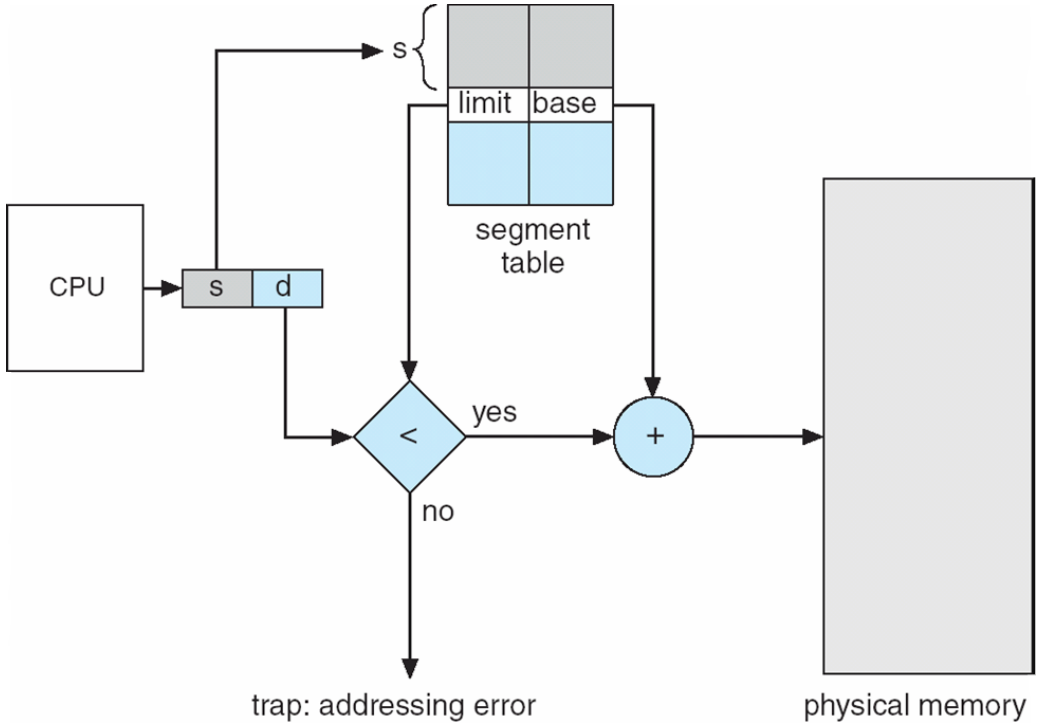
Logical address = ⟨segment#; offset⟩

Segmentation

- Segmentation(分段)
  - Logical address consists of a two tuple:
    <segment-number, offset>,
  - Segment table – maps two-dimensional physical addresses; each table entry has:
    - ✓ base – contains the starting physical address where the segments reside in memory
    - ✓ 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;
    segment number s is legal if s < STLR

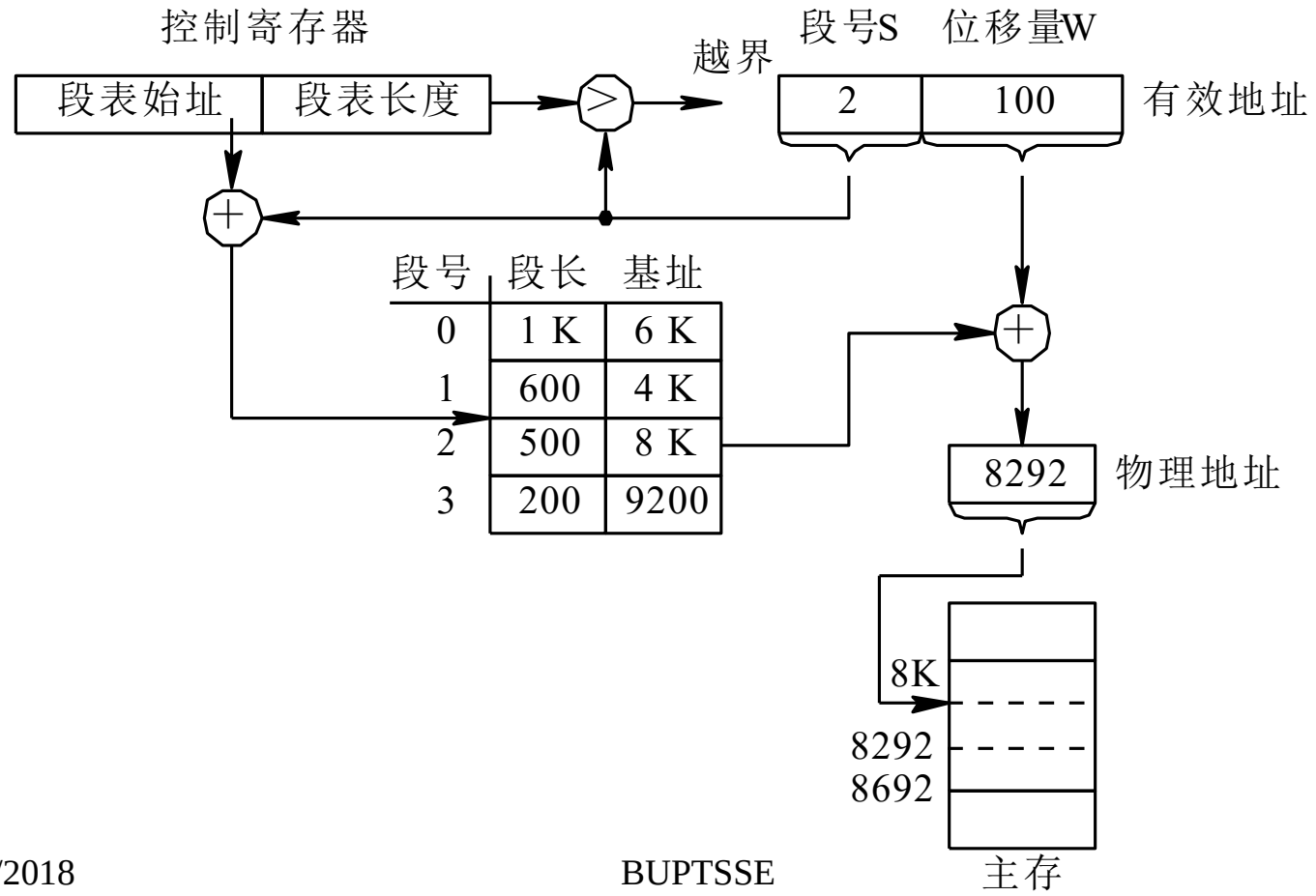
Segmentation

- Segmentation(分段)
  - Segmentation Architecture



Segmentation

- Segmentation(分段)
  - Segmentation Architecture



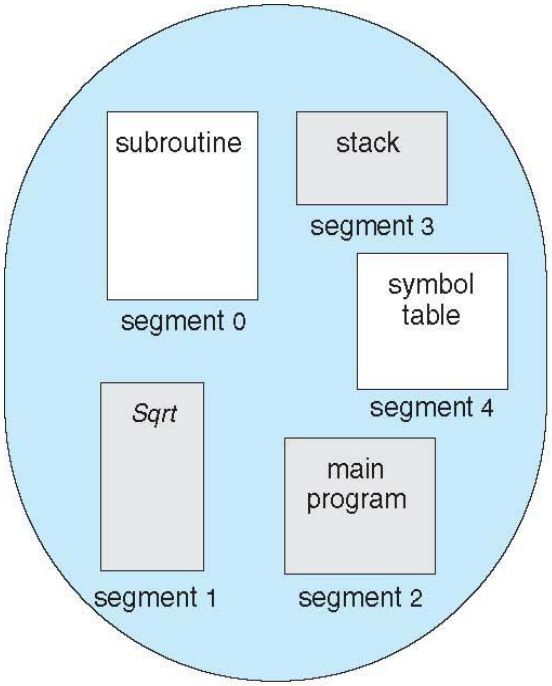
Segmentation

- Segmentation(分段)
  - Example of Segmentation

✓ < segment2, 53 > → 4300 + 53 = 4353

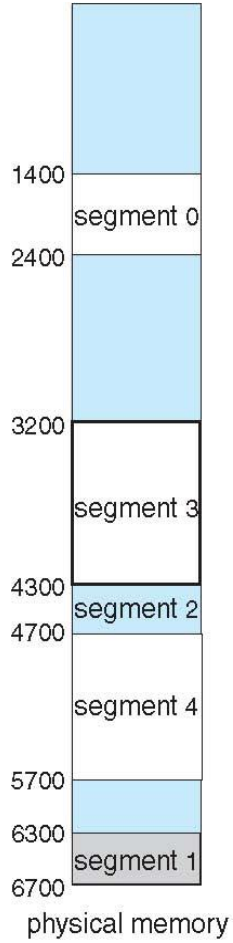
✓ < setment3, 852 > → 3200 + 852 = 4052

✓ What about < segment1, 536 >?



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

segment table



Segmentation

- 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 information
    - ✓ 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