Chapter 8 Main Memory

LI Wensheng, SCS, BUPT

Strong point:

Paging

Segmentation

* About Main Memory Management

- **Goals of Memory Management**
  - **Subdividing memory to accommodate multiple processes.**
  - **Memory needs to be allocated efficiently to pack as many processes into memory as possible.**
- **Memory Management Requirements**
  - **Relocation**
    - **Programmer does not know where the program will be placed in memory when it is executed.**
    - **While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated).**
    - **Memory references must be translated in the code to actual physical memory address.**

* About Main Memory Management(Cont.)

- **Memory Management Requirements(Cont.)**
  - **Protection**
    - **Processes should not be able to reference memory locations in another process without permission.**
    - **Must be checked during execution**
      - **Impossible to check absolute addresses in programs since the program could be relocated.**
      - **Operating system cannot anticipate(预测) all of the memory references a program will make.**
  - **Sharing**
    - **Allow several processes to access the same portion of memory.**
    - **Better to allow each process (person) access to the same copy of the program rather than have their own separate copy.**

* About Main Memory Management(Cont.)

- **Memory Management Requirements(Cont.)**
  - **Logical Organization**
    - **Programs are written in modules.**
    - **Modules can be written and compiled independently.**
    - **Different degrees of protection given to modules (read-only, execute-only).**
    - **Share modules**
  - **Physical Organization**
    - **Memory available for a program plus its data may be insufficient.**
      - **Overlaying allows various modules to be assigned the same region of memory**
    - **Programmer does not know how much space will be available.**

Chapter Objectives

- To provides a detailed description of various ways of organizing memory hardware.
- To explore various techniques of allocating memory to processes.
- To discuss in detail how paging works in contemporary computer systems.

Contents

8.1 Background

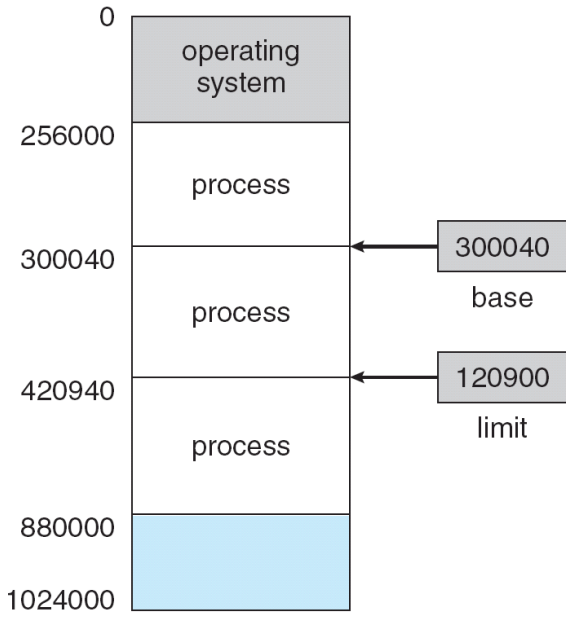
- Memory consists of a large array of words or bytes, each with its own address.
- The program must be brought into memory and placed within a process for it to be run.
- Main memory and registers are only storage CPU can access directly.
  - Register access in one CPU clock (or less).
  - Main memory can take many cycles of the CPU clock, causing a **stall.**
    It does not have the data required to complete the instruction.
- **Cache** sits between main memory and CPU registers.
  - typically on the CPU chip.

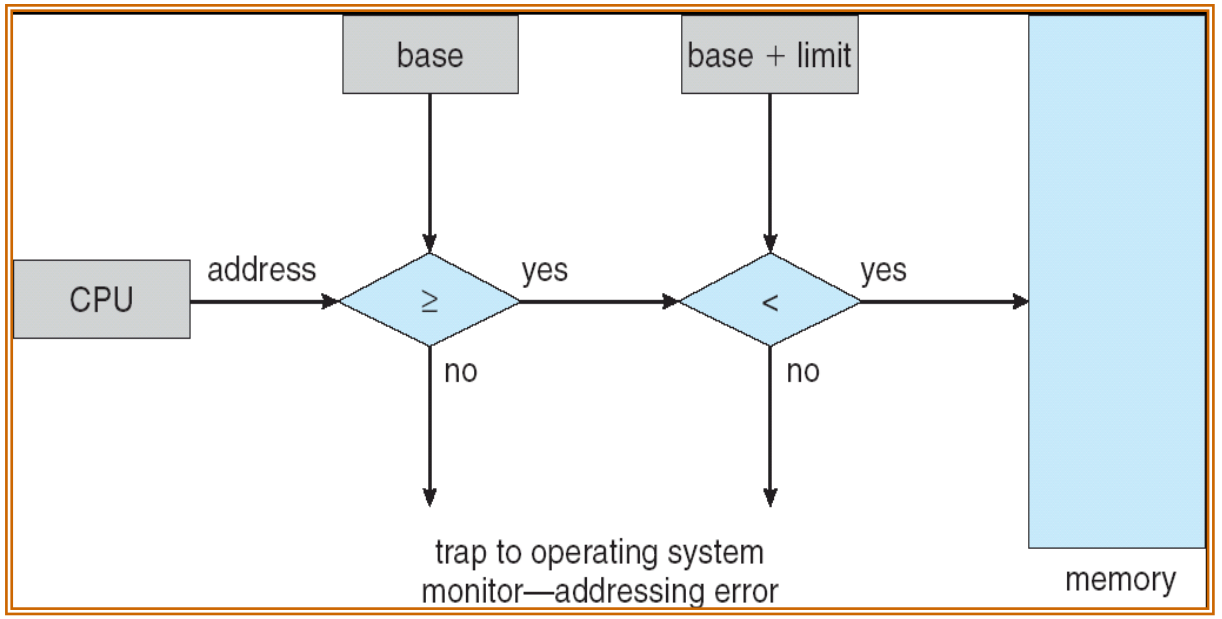
Background(Cont.)

- A typical instruction-execution cycle:
  - Fetch an instruction, decode instruction, fetch operands, execute, store results back into memory.
- Memory Unit only sees:
  - a stream of addresses + read requests, or
  - address + data and write requests.
- Protection of memory required to ensure correct operation.
  - This protection must be provided by the hardware.
  - make sure each process has a separate memory space.
  - ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

Protection: Base and Limit Registers

- **base** and **limit** registers define the physical address space.
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.



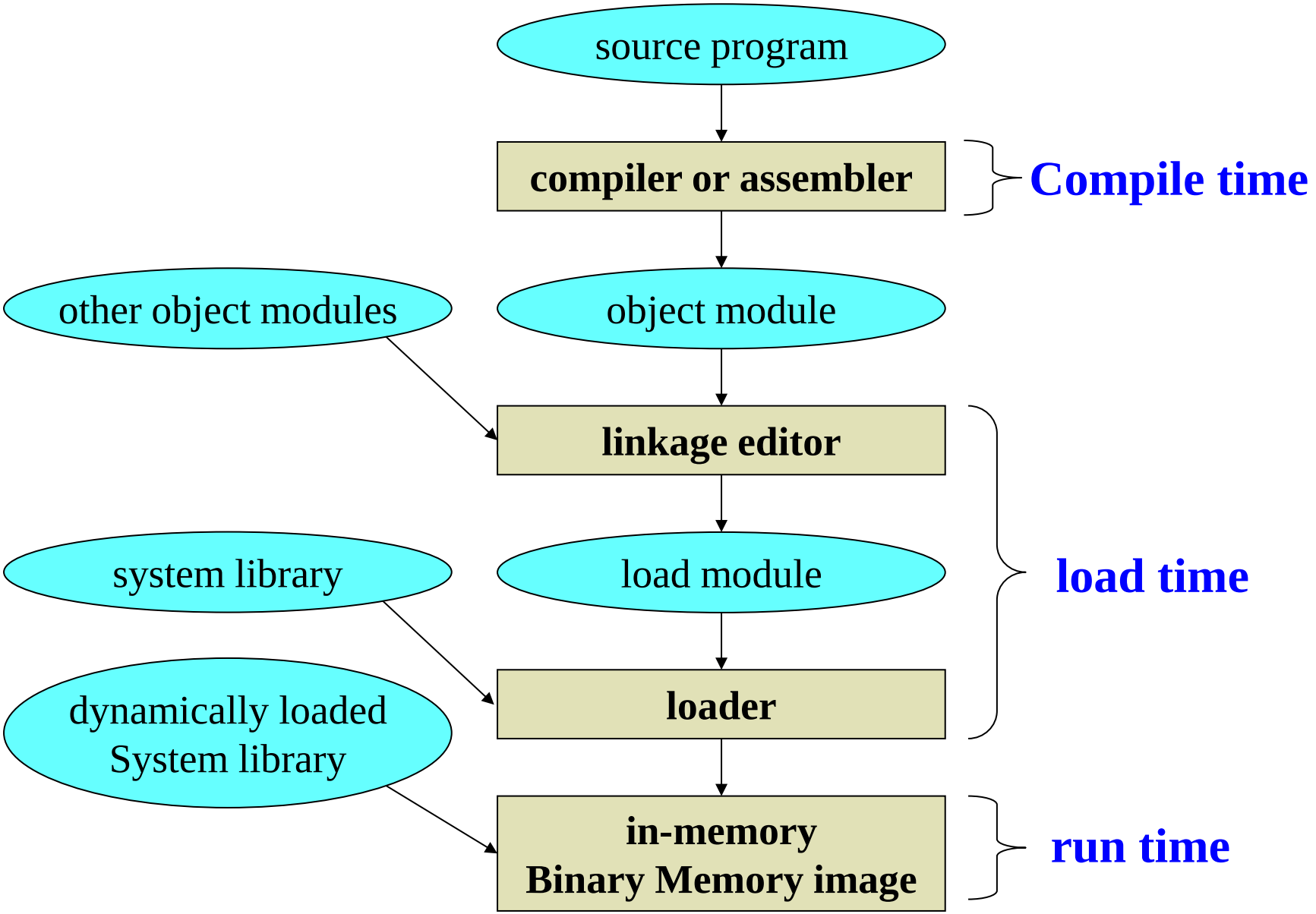


- The base and limit registers can be loaded only by the OS, which uses a special **privileged** instructions.

Address Binding

- Programs on disk, ready to be brought into memory to execute form an input queue.
- Most systems allow a user process to reside in any part of the physical memory.
- A user program goes through several steps before being executed.
  Addresses represented in different ways at different stages.
  - Addresses in source program are usually symbolic.
  - A compiler binds symbolic addresses to relocatable addresses. E.g. “14 bytes from beginning of this module”
  - Linker or loader will bind relocatable addresses to absolute addresses. i.e. 74014.
  - Each binding maps one address space to another.

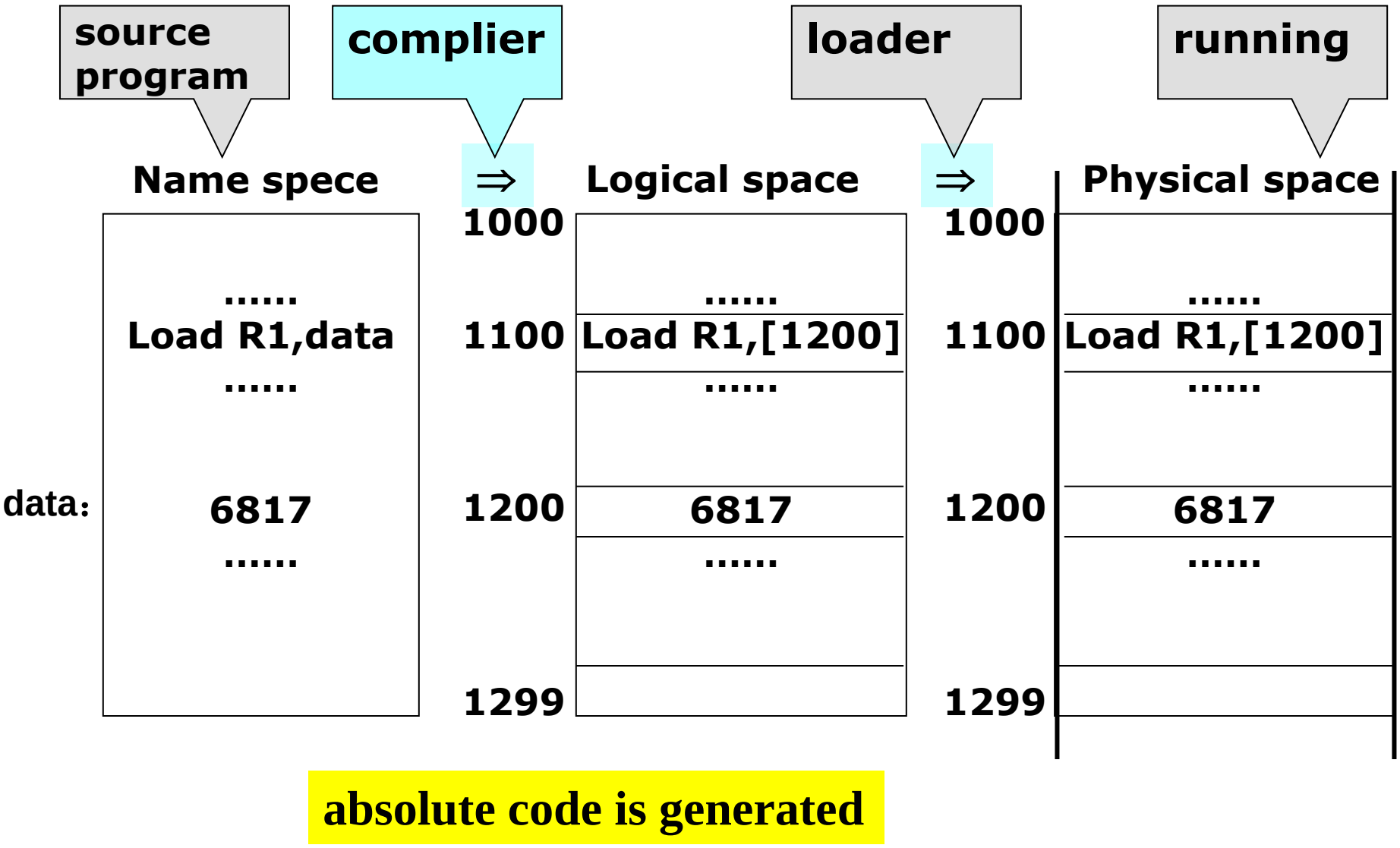
Multistep Processing of a User Program



Binding of Instructions and Data to Memory

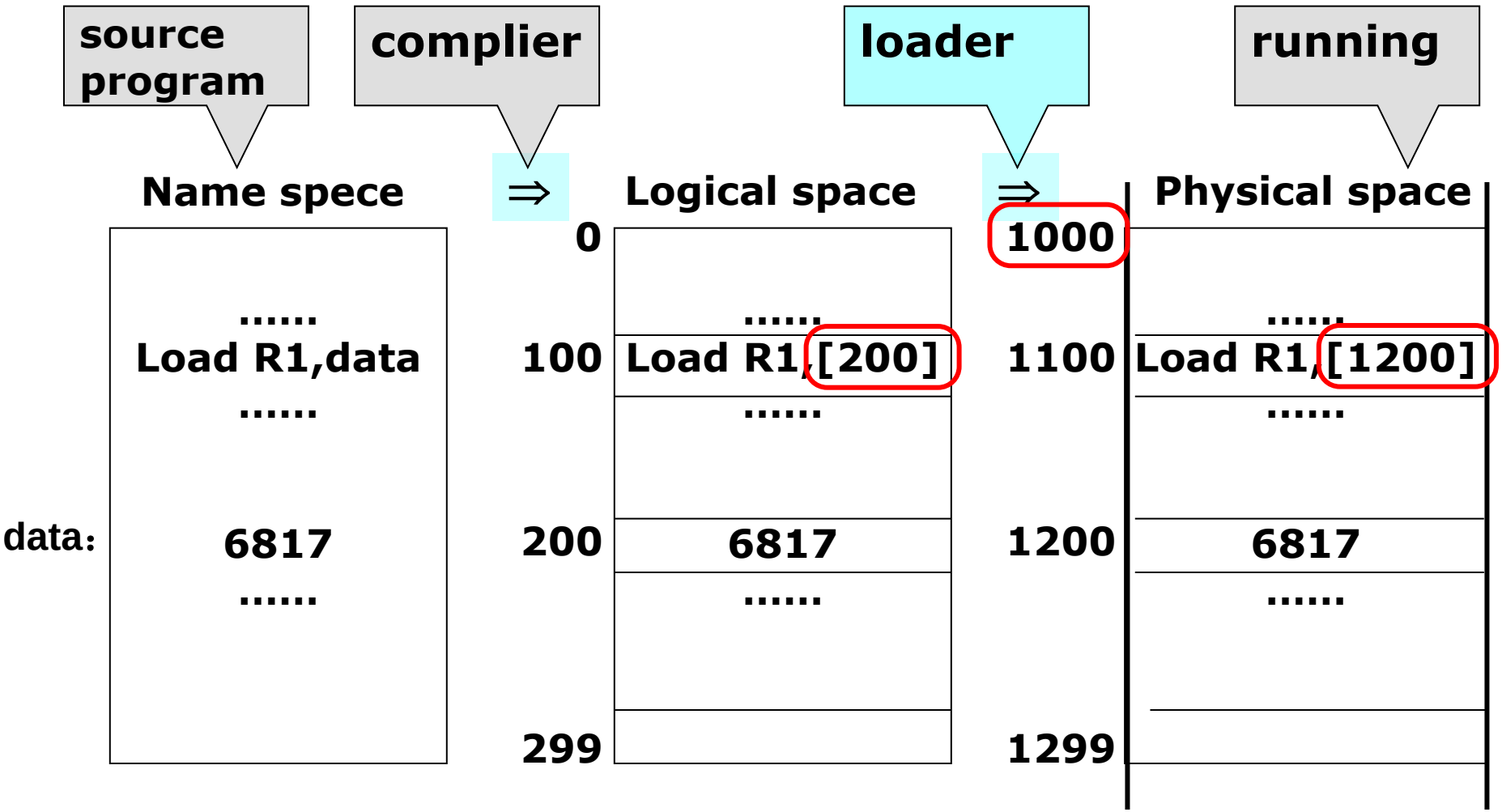
- **The binding of instructions and data to memory addresses can be done at three different stages.**
  - **Compile time: If memory location known a priori, absolute code can be generated.**
    - **Must recompile code if starting location changes.**
  - **Load time: Must generate *relocatable* code if memory location is not known at compile time.**
    - **If the starting address changes, we need only reload the user code to incorporate this changed value.**
  - **Execution time: Binding must be delayed until run time if the process can be moved during its execution from one memory segment to another.**
    - **Need hardware support (e.g., *base* and *limit registers*).**

* Address Binding-- Compile time



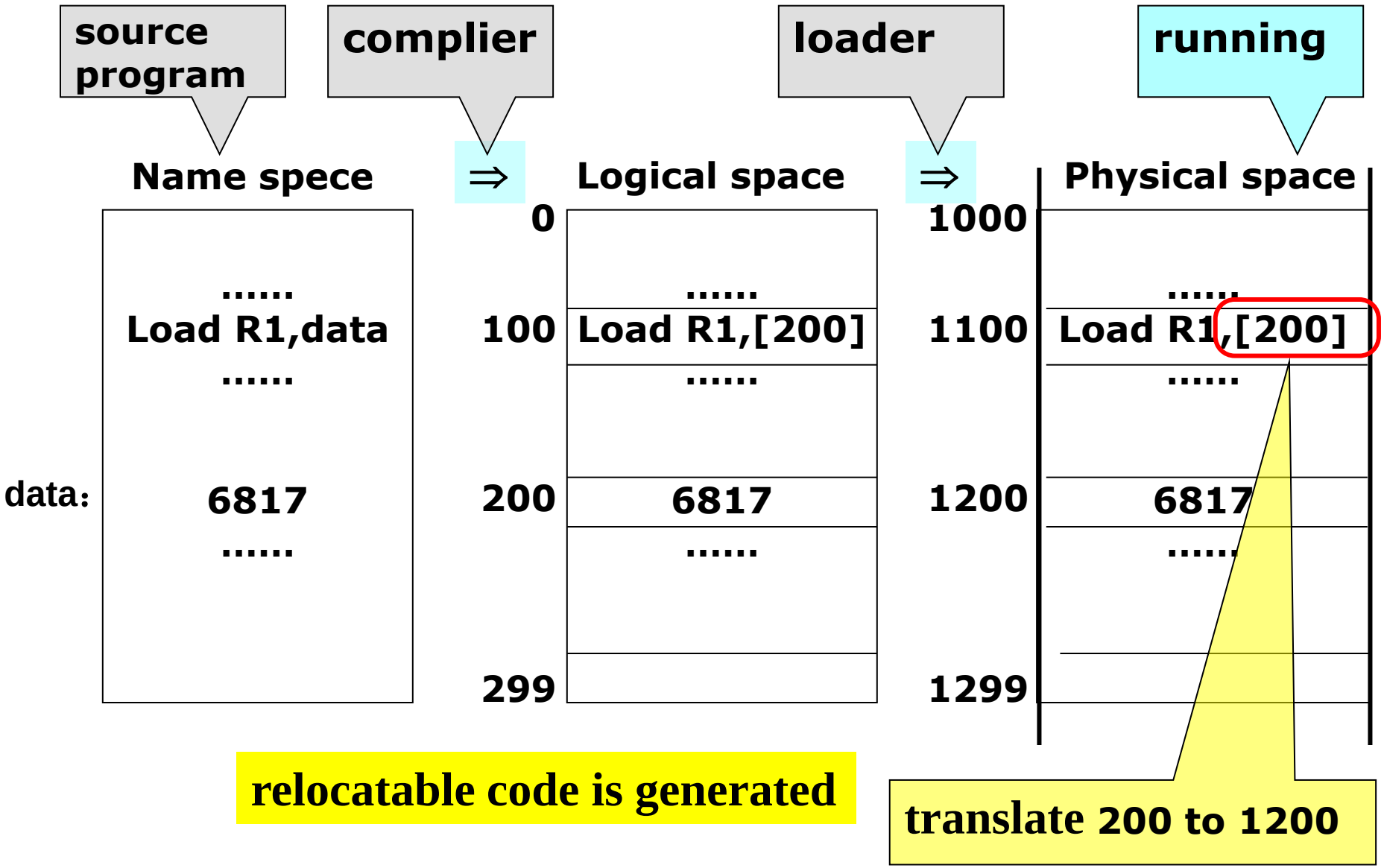
absolute code is generated

* Address Binding-- Load time



relocatable code is generated

* Address Binding-- Execution time



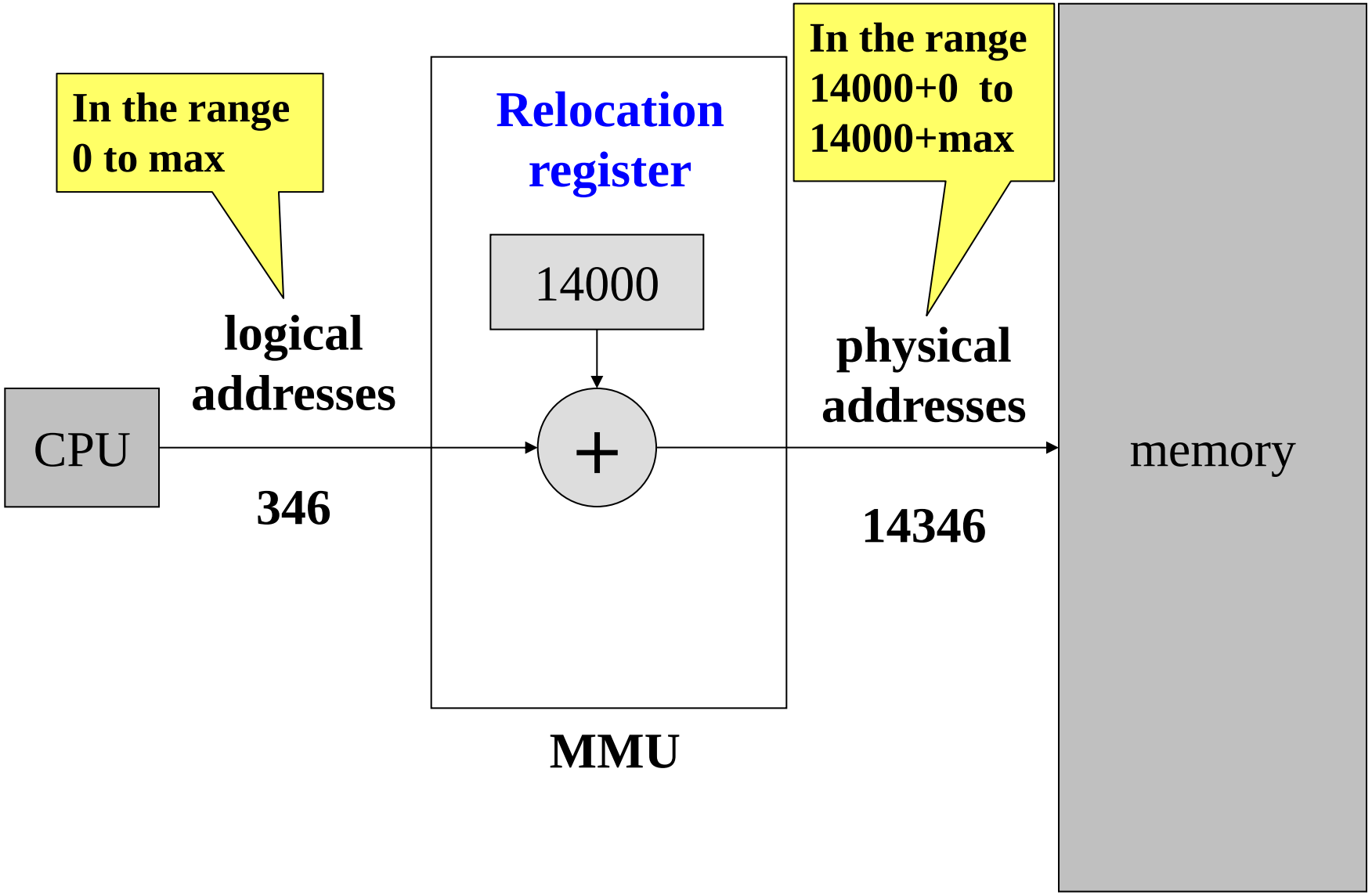
Logical vs. Physical Address Space

- ***Logical address*** **– generated by the CPU; also referred to as *virtual address*.**
- ***Physical address*** **– address seen by the memory unit; loaded into the memory-address register.**
- **In compile-time and load-time address-binding schemes, logical and physical addresses are the same.**
- **In execution-time address-binding scheme, logical (virtual) and physical addresses differ.**
- **Logical address space: The set of all logical addresses generated by a program.**
- **Physical address space: the set of all physical addresses corresponding to these logical addresses.**
- **The memory-mapping hardware converts logical addresses into physical addresses.**

Memory-Management Unit (MMU)

- Hardware device that maps virtual address to physical address in run-time.
  - Many methods possible.
- Consider a simple MMU scheme using base-register.
  - Base register now is called relocation register.
  - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
  - MS-DOS on Intel 80x86 used 4 relocation registers.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
  - Execution-time binding occurs when reference is made to location in memory.

Dynamic Relocation Using a Relocation Register



Dynamic Loading (*)

- **Routine is not loaded until it is called.**
  - **Unused routine is never loaded.**
  - **Useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.**
  - **Better memory-space utilization.**
- **Main program is loaded into memory and is executed.**
  - **all routines are kept on disk in a relocatable load format.**
- **When needed, the relocatable linking loader is called.**
- **No special support from the OS is required**
  - **implemented through program design.**
  - **OS can help by providing libraries to implement dynamic loading.**

Dynamic Linking and Shared Libraries (*)

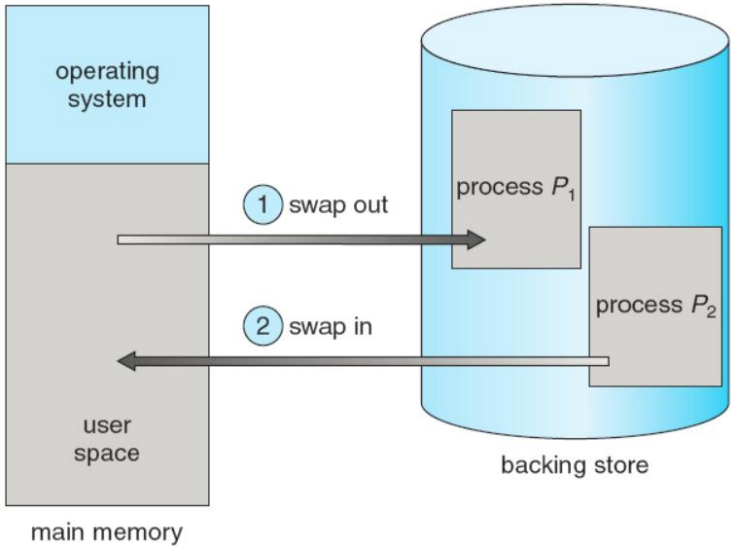
- **Static linking – system libraries and program code combined by the loader into the binary program image.**
- **Dynamic linking – Linking is postponed until execution time.**
- **A *stub* is included in the image for each library-routine reference.**
  - ***stub*, a small piece of code, indicates how to locate the appropriate memory-resident library routine, or how to load the library if the routine is not already present.**
  - **When executed, the *stub* replaces itself with the address of the routine, and executes the routine.**
- **Dynamic linking generally requires help from the OS.**
  - **OS checks if routine is in processes' memory address.**
  - **If not in address space, add to address space.**

Dynamic Linking and Shared Libraries(* Cont.)

- Dynamic linking is particularly useful for libraries.
  - A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
- System also known as shared libraries.
- Consider applicability to patching system libraries
  - Versioning may be needed.
  - Version information is included in both the program and the library, so that programs will not accidentally execute new, incompatible versions of libraries.
  - More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use.

8.2 Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
  - Total physical memory space of processes can exceed physical memory.
- Backing store – fast disk, large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- Will the swapped out process be swapped back to the same memory space it used before?
  - Depends on address binding method.



Swapping (Cont.)

- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms.
  - Lower-priority process is swapped out so higher-priority process can be loaded and executed.
- *ready queue* — Consists of all processes whose memory images are on the backing store or in memory and are ready to run.
- Dispatcher is called whenever the CPU scheduler decides to execute a process.
  - checks to see whether the next process in the queue is in memory.
  - If not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.

Swapping (Cont.)

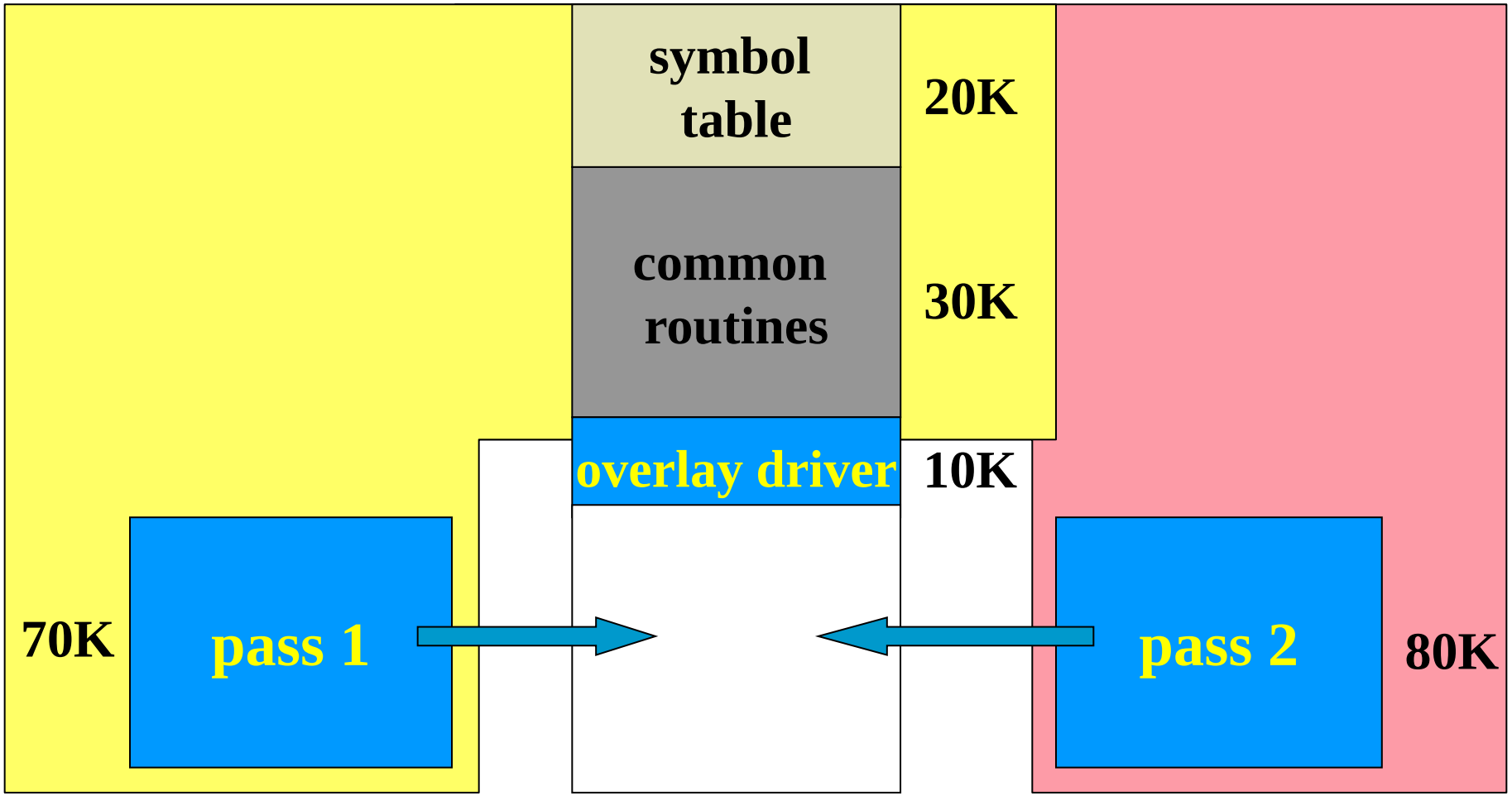
- Context switch time can be very high.
  - Major part of swap time is transfer time.
  - E.g. 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - Swap out time of 2000 ms
    - Plus swap in of same sized process
    - Total swap time is about 4000ms (4 seconds)
- Total transfer time is directly proportional to the *amount* of memory swapped.
  - Be useful to know exactly how much memory a user process is using, not simply how much it might be using.
  - The user must keep the system informed of any changes in memory requirements.
    - System calls: request_memory() and release_memory().

Swapping (Cont.)

- Ensure the process be swapped out is completely idle.
- Solution to pending I/O
  - Never swap a process with pending I/O.
  - Execute I/O operations only into OS buffers.
    Known as double buffering, adds overhead.
- Standard swapping is not used in modern OSs.
  - Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.
  - Swap only when free memory extremely low.
  - Swapping normally disabled, Started if the amount of free memory falls below a threshold amount, Halted when the amount of free memory increases.

* Overlays 覆盖

- Keep in memory only those instructions and data that are needed at any given time.



* Overlays (Cont.)

- Needed when process is larger than amount of memory allocated to it.
- Implemented by user.
  - No special support needed from operating system.
  - Programming design of overlay structure is complex.
  - This task can be a major undertaking, requiring complete knowledge of the structure of the program, its code, and its data structures.
- limited to microcomputer and other systems that have limited amounts of physical memory and that lack hardware support for more advanced techniques.

* 覆盖与交换技术

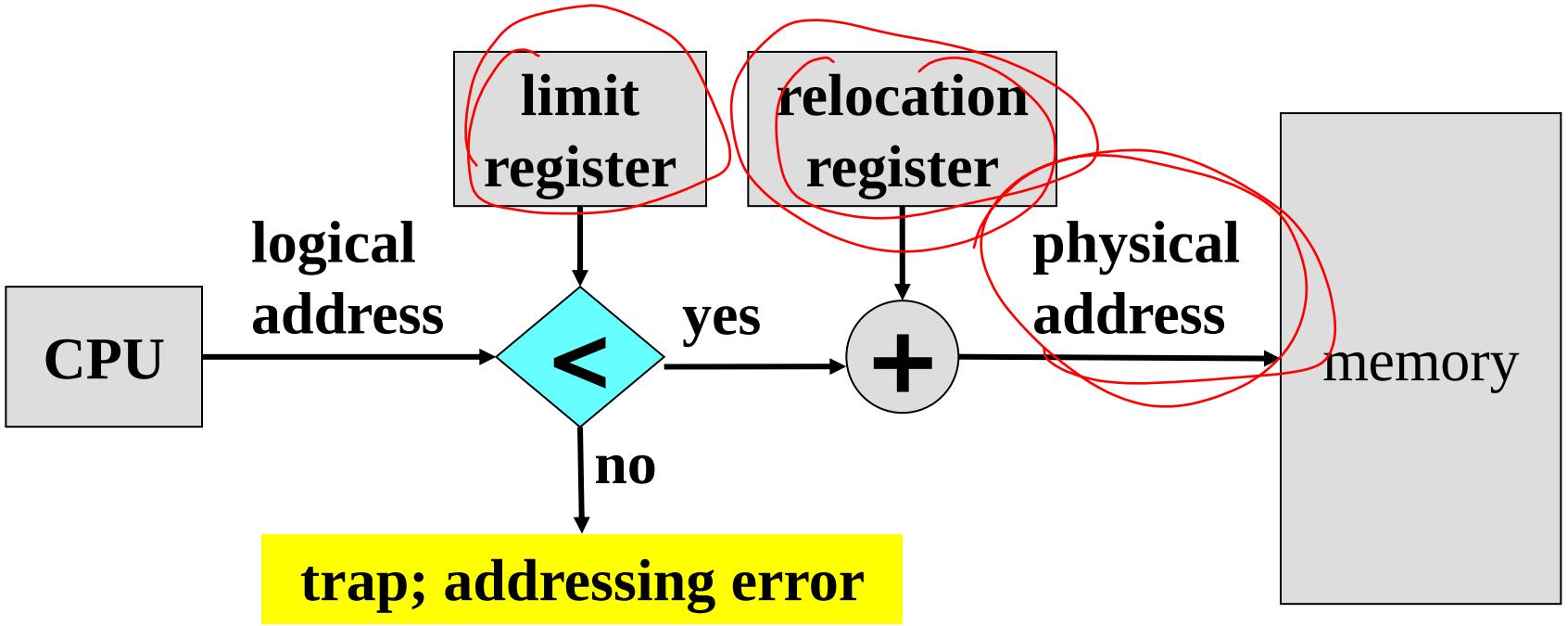
- 多道程序设计环境下用来扩充内存的两种方法。
- 覆盖技术
  - 当用户程序所需要的内存空间比分配给它使用的内存空间大时，可以使用覆盖技术。
  - 由程序员实现，不需要操作系统特别支持
  - 设计思想：把程序划分为若干个功能上相对独立的程序段，按照程序的逻辑结构让那些不会同时执行的程序段共享同一块内存区域。
  - 对程序员要求很高
    - 必须十分清楚程序的逻辑结构
    - 明确规定各程序段的执行和覆盖顺序。
    - 设计实现覆盖驱动模块。
- 交换技术
  - 由操作系统中的交换程序实现，对用户透明。
  - swap out：将内存某区域中的进程调出内存写入外存交换区。
  - swap in：从外存交换区中把指定的进程调入内存，并让它执行。
- 交换与覆盖的区别
  - 覆盖技术要求程序员自己设计覆盖结构，交换对用户是透明的。
  - 交换主要是在进程和进程之间进行，覆盖则是在同一进程内进行。

8.3 Contiguous Memory Allocation

- must accommodate both OS and user processes.
- Limited resource, must allocate efficiently.
- memory is usually divided into two partitions:
  - OS, usually held in low memory with interrupt vector.
  - User processes, held in high memory.
- User space is divided into Multiple partitions.
- Single-partition allocation for each process.
- **Relocation-register scheme** used to protect user processes from each other, and from changing operating-system code and data.
  - **Relocation/base register** contains the value of the smallest physical address;
  - **limit register** contains the range of logical addresses . Each logical address must be less than the limit register.

Memory Protection

领土神圣，不容侵犯
遵守规则，越界制裁



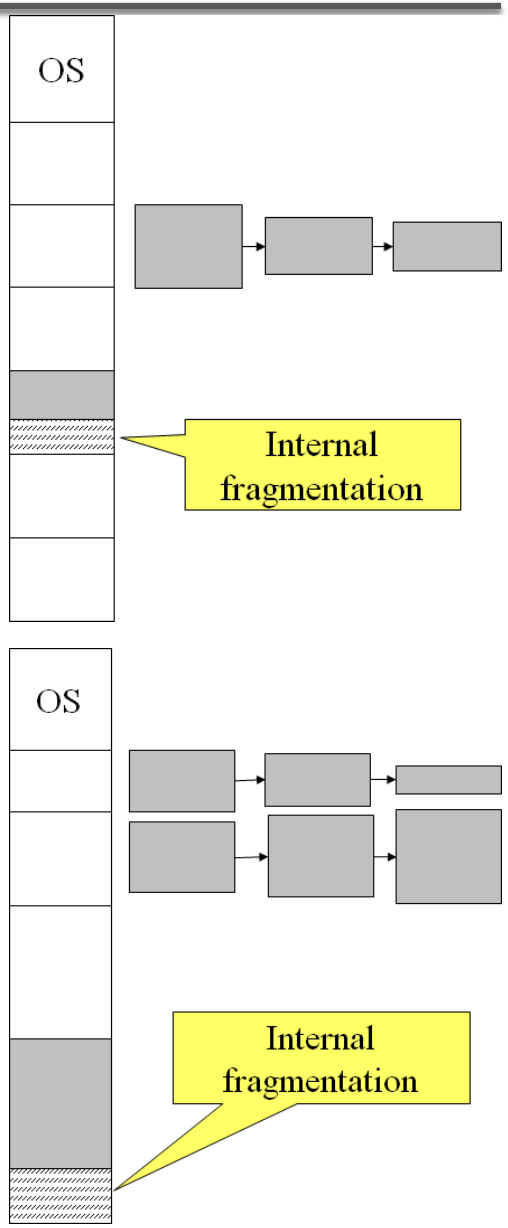
- MMU maps logical address *dynamically*.
- The dispatcher loads the relocation and limit registers.

Memory allocation

- **Fixed Partitioning （called MFT，IBM OS/360）**
  - **The degree of multiprogramming is bound by the number of partitions.**
  - **Equal-size partitions**
    - **Any process whose size is less than or equal to the partition size can be loaded into an available partition.**
    - **All partitions are full, OS can swap a process out of a partition.**
    - **A program may not fit in a partition, the programmer must design the program with overlays.**
  - **Unequal-size partitions**
  - **Memory use is inefficient.**
    - **Any program, no matter how small, occupies an entire partition. This results in internal fragmentation.**

Placement Algorithm with Partitions

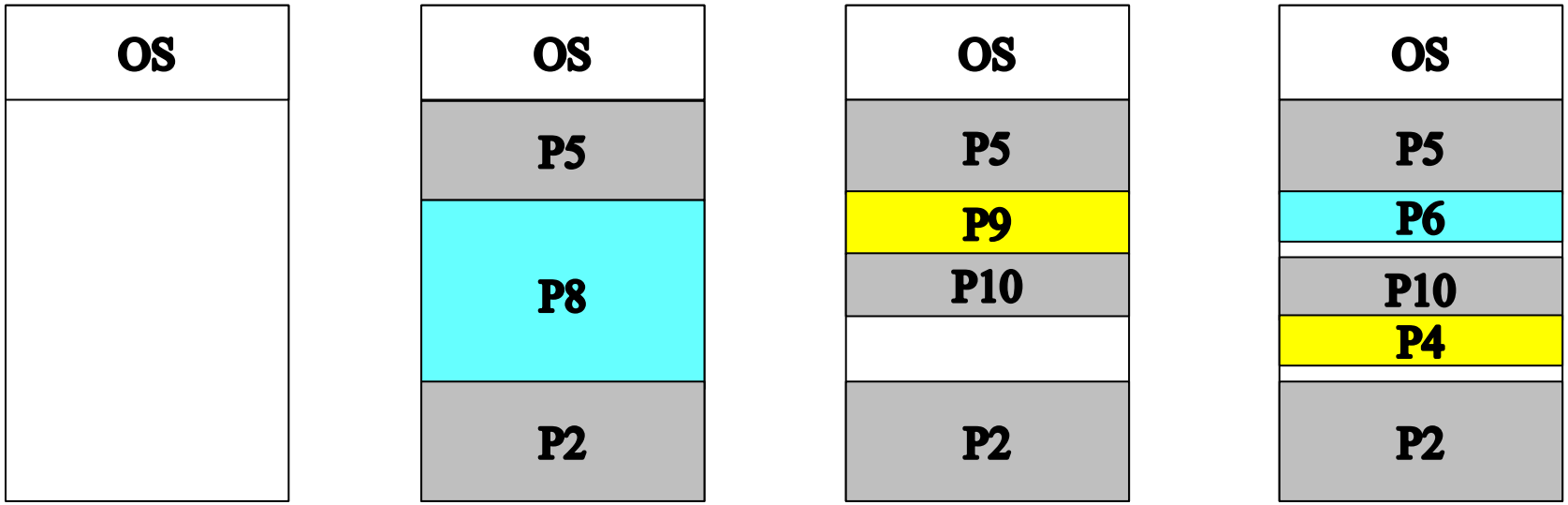
- **Equal-size partitions**
  - All partitions are of equal size, it does not matter which partition is used.
- **Unequal-size partitions**
  - Can assign each process to the smallest partition within which it will fit.
  - Queue for each partition.
  - Processes are assigned in such a way as to minimize wasted memory within a partition.
- **Data structure?**
  - Partition_id, base address, limit, status.



Variable-partition scheme

- Dynamic Partitioning (called MVT)
  - Partitions are of variable length and number.
- *Hole* – a block of available memory.
  - Initially, all memory is available for user processes, a hole.
  - Holes of various size are scattered throughout memory.
- The OS maintains a table containing information:
  - Which partitions are allocated, and to which process.
  - Which partitions are free (hole)
- Memory allocation and reclaim
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
    Process is allocated exactly as much memory as required.
  - Process exiting frees its partition, adjacent free partitions combined.

Variable-partition scheme(Cont.)



- a list of available block sizes, and the input queue.
- A large hole is split into two parts: one is allocated to the process, the other is returned to the set of holes.
- Many small holes in the memory, external fragmentation.
- Can use compaction to shift processes so they are contiguous and all free memory is in one block.
- Data structure: Base address, limit

Dynamic Storage-Allocation Problem

- How to satisfy a request of size *n* from a list of free holes?
  - *First-fit:* Allocate the *first* hole that is big enough.
  - *Best-fit*: Allocate the *smallest* hole that is big enough.
    - must search the entire list, unless ordered by size.
    - Produces the smallest leftover hole.
  - *Worst-fit*: Allocate the *largest* hole.
    - must also search the entire list.
    - Produces the largest leftover hole.
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.
- These algorithms suffer from external fragmentation.

Fragmentation

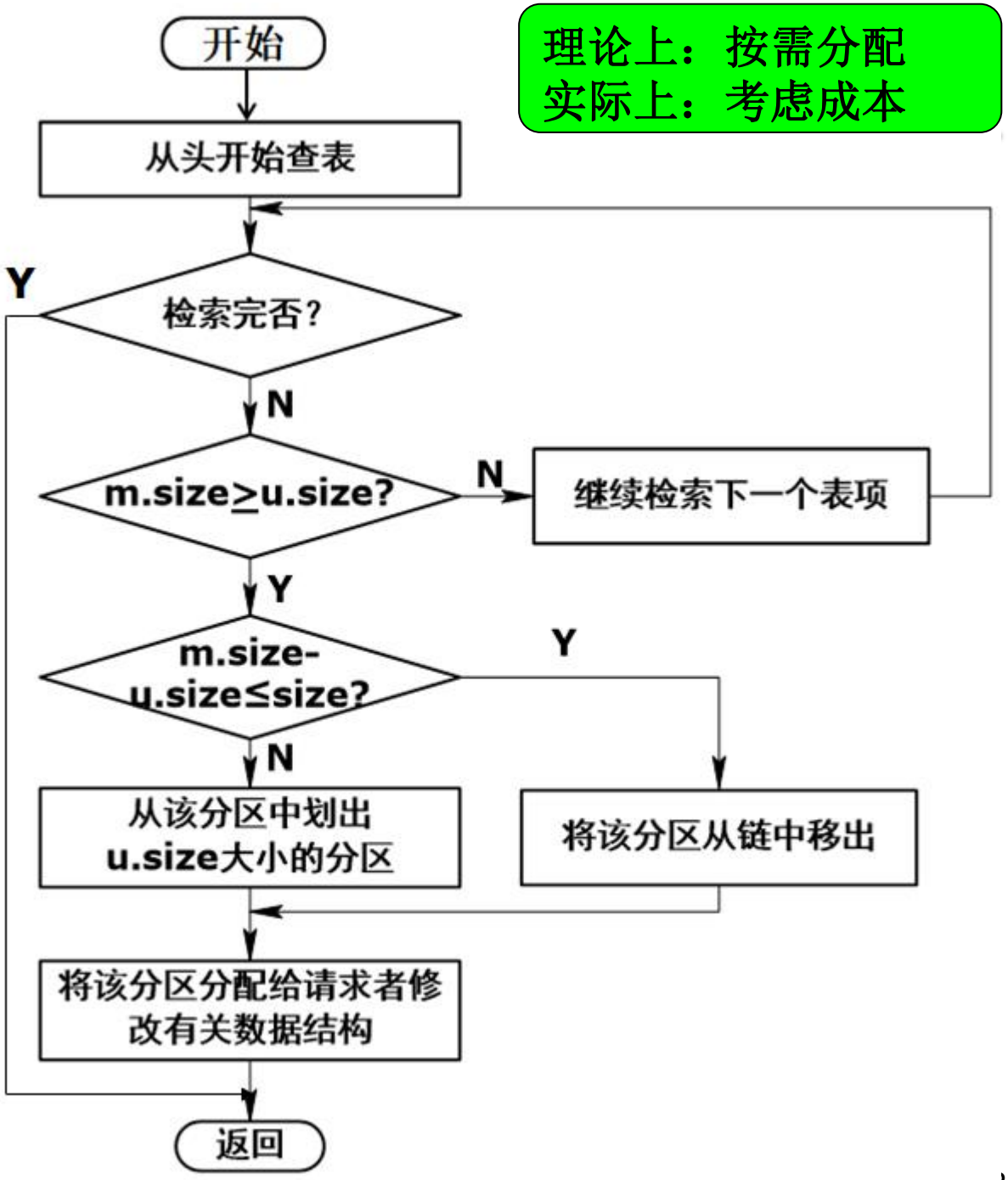
- *External Fragmentation* – total memory space exists to satisfy a request, but it is not contiguous.
- *Internal Fragmentation* – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Statistical analysis of first fit reveals that given N blocks allocated, another 0.5 N blocks will be lost to fragmentation.
  - 1/3 of memory may be unusable -> 50-percent rule.

Fragmentation(Cont.)

- **Reduce external fragmentation by *compaction.***
  - **Shuffle memory contents to place all free memory together in one large block.**
  - **Compaction is possible only if relocation is dynamic, and is done at execution time.**
  - **I/O problem**
    - **Latch job in memory while it is involved in I/O.**
    - **Do I/O only into OS buffers.**
- **Backing store has same fragmentation problems.**

* 分配流程

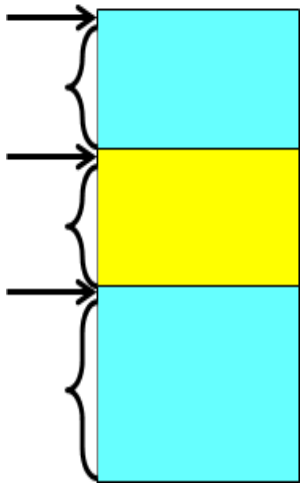
- m.size: hole size
- u.size: process size
- size: threshold

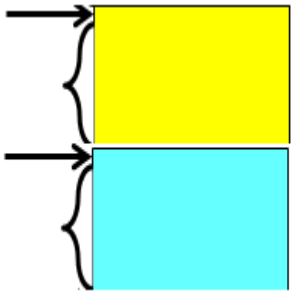


理论上：按需分配
实际上：考虑成本

* 动态分区的回收

- 进程终止，OS 收回其内存空间，并插入空闲分区表/链中。
- 要考虑内存中与其上下相邻的存储空间的使用情况，如果有空闲分区，则要进行合并。
- 上下相邻分区的状态
  - 上下邻分区均空闲
    - 更新上邻空闲区记录的长度=上邻区长度 + 释放区长度+下邻区长度
    - 从表/链中删除下邻区的记录
  - 上邻分区空闲
    - 更新上邻区记录的长度=上邻区长度+释放区长度
  - 下邻分区空闲
    - 更新下邻区记录的长度=释放区长度+下邻区长度
    - 更新下邻区记录的起始地址=释放区的起始地址
  - 上下邻分区均非空闲
    - 为释放区在表/链中适当的位置建立记录





* First fit 首次适应法

- 按空闲区首地址递增的次序组织空闲区表(队列)。
- 分配：
  - 从空闲区表的第一个表项/或上次分配结束时的位置开始查询，遇到满足要求的空闲区，从中分配。
- 回收：
  - 按释放区的首地址查询空闲区表，
    - 若有与释放区相邻的空闲区，则合并到相邻的空闲区中，并修改该区的大小和首地址。
    - 否则，把释放区作为一个空闲区，将其大小和首地址按照首地址大小递增的顺序插入到空闲区表的适当位置。

* Best fit 最佳适应法

- 选中的空闲区是满足要求的最小空闲区。
- 按空闲区大小递增的次序组成空闲区表/队列。
- 分配：
  - 从表头开始查找，找到第一个满足要求的空闲区，从中分配。
- 回收：
  - 按释放区的首地址查询空闲区表/队列，
    - 若有与释放区相邻的空闲区，则合并到相邻的空闲区中，并修改该区的大小和首址。
    - 否则，把释放区作为一个空闲区插入空闲区表/队列。
- 分配和回收后要对空闲区表/队列重新排序。

* Worst fit 最坏适应法

- 按空闲区大小递减的顺序组织空闲区表/队列。
- 分配：
  - 检查空闲区表的第一个空闲区的大小是否满足要求。
    若不满足，则分配失败；满足的话，则从中分配。
- 回收：
  - 按释放区的首地址查询空闲区表/队列
    - 若有与释放区相邻的空闲区，则合并到相邻的空闲区中，并修改该区的大小和首址。
    - 否则，把释放区作为一个空闲区插入空闲区表/队列。
- 分配和回收后要对空闲区表/队列重新排序。

* 三种策略比较

- 上述三种放置策略各有利弊，到底哪种最好不能一概而论，而应针对具体作业序列来分析。
- 对某一作业序列来说，某种算法能将该作业序列中所有作业安置完毕，那么我们说该算法对这一作业序列是合适的。
- 对某一算法而言，如它不能立即满足某一要求，而其它算法却可以满足此要求，则这一算法对该作业序列是不合适的。

* Exercise 1

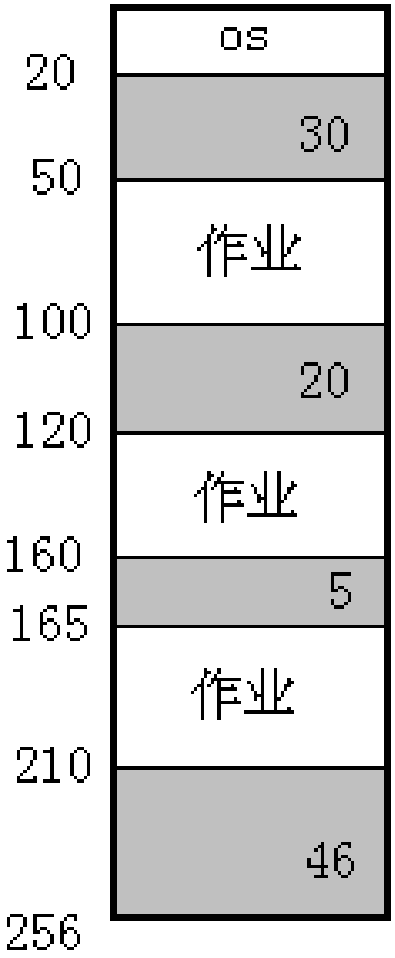
某系统，内存状态如右所示。

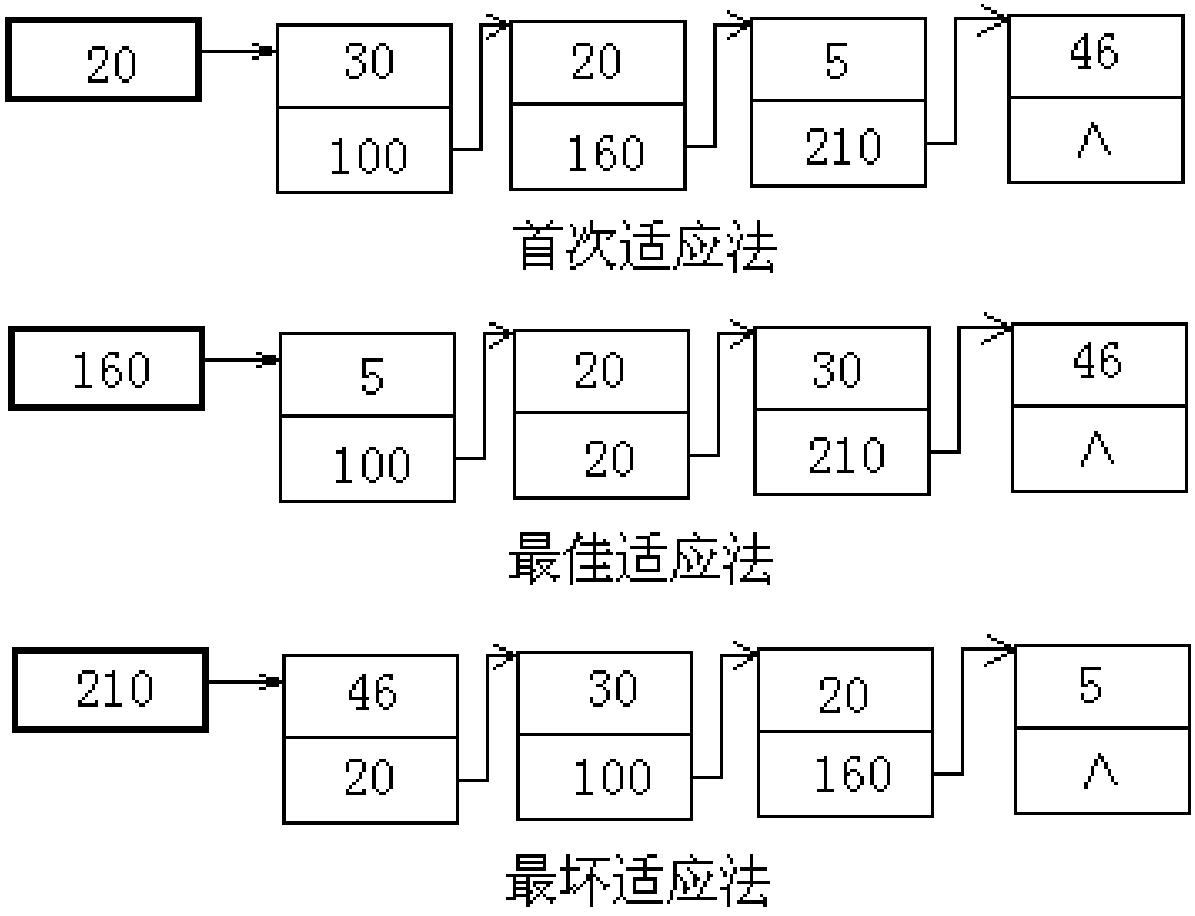
有如下作业序列：

- A需要内存18M，B需要25M，C需要30M。
- 根据分析结果回答：哪种分配算法对此作业序列是合适的？

* Answer to Exercise 1

- A需要18M，B需要25M，C需要30M。





* Exercise 2

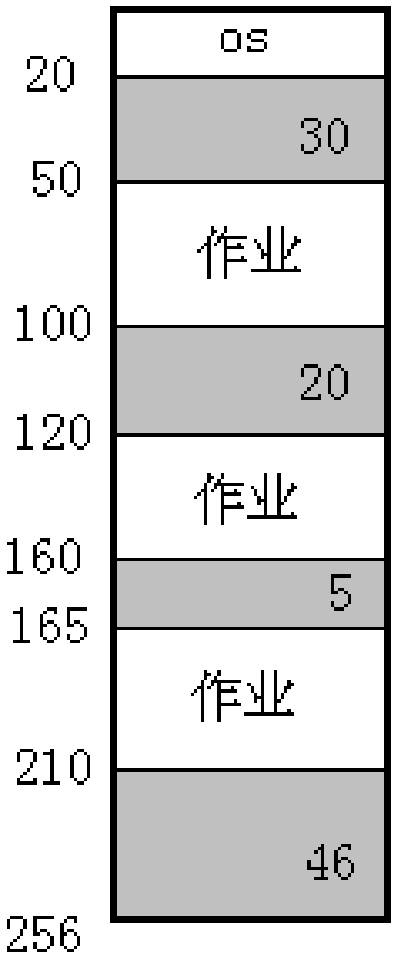
某系统，内存状态如右所示。

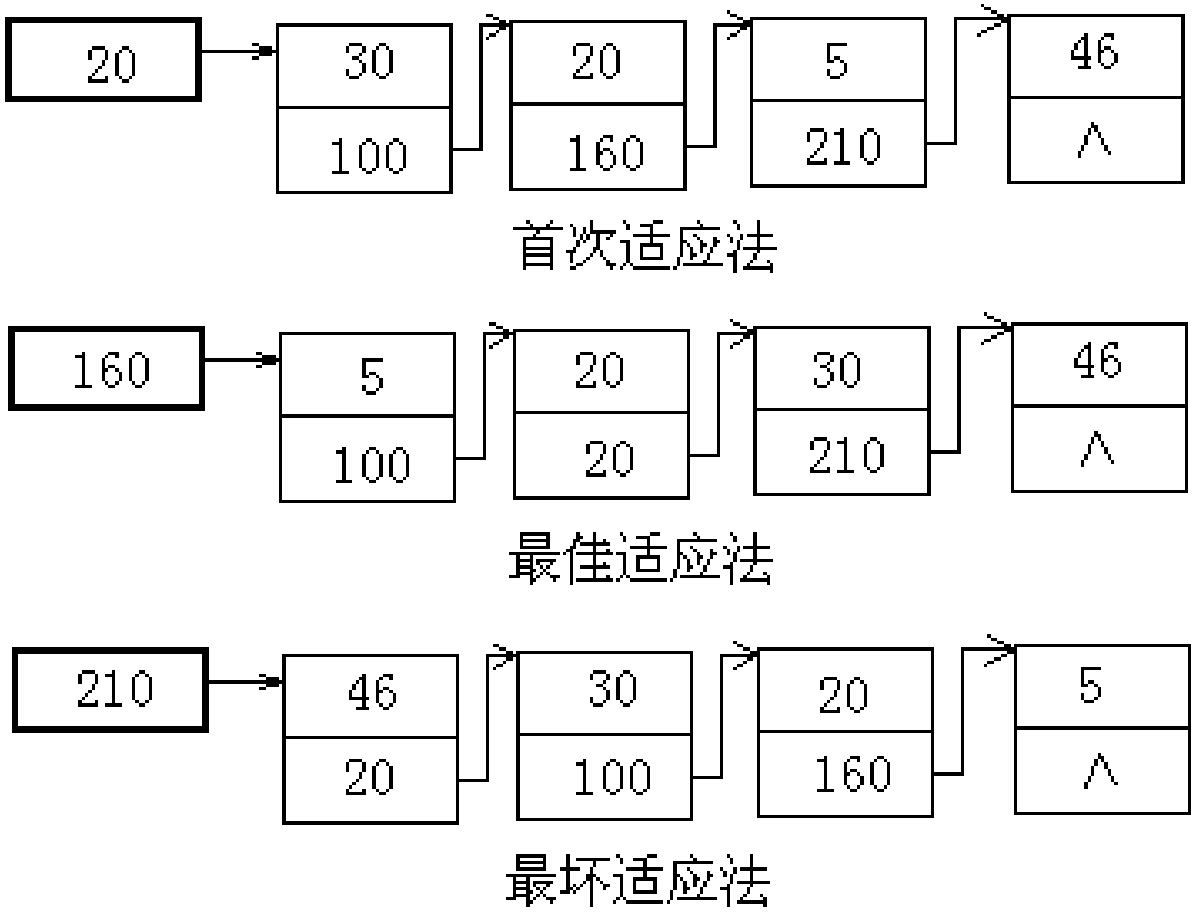
有如下作业序列：

- A需要内存21M，B需要30M，C需要25M。
- 根据分析结果回答：哪种分配算法对此作业序列是合适的？

* Answer to Exercise 2

- A需要21M，B需要30M，C需要25M。





* Exercise 3

- 用可变分区(动态重定位)方式管理内存时，假定内存中按地址顺序依次有5个空闲区，空闲区的大小依次为：32M、10M、5M、228M和100M。

  现在有5个作业A、B、C、D、E。它们各需主存大小为：1M、10M、108M、28M和115M。

  问：

  采用哪种算法能把这5个作业按顺序全部装入内存?

* Answer to Exercise 3

A: 1M
B: 10M
C: 108M
D: 28M
E: 115M

- **First fit:**

- **Best fit:**

* Answer to Exercise 3 (Cont.)

A: 1M
B: 10M
C: 108M
D: 28M
E: 115M

- **Worst fit:**

End 3

8.4 Paging

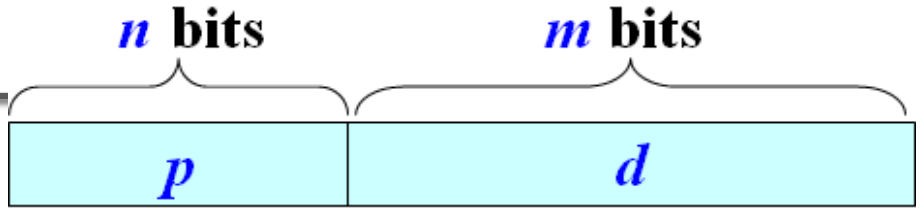
- Process's physical address space can be noncontiguous.
- The basic method for implementing paging:
  - Physical memory is divided into fixed-sized blocks, **frames**.
  - Logical memory is divided into blocks of same size, **pages**.
  - Backing store is divided into fixed-sized **blocks** that are of the same size as the memory frames.
  - size is power of 2, between 512B and 16MB.
- When a process is to be executed, its **pages** are loaded into any available **frames** from the **blocks**.
- Process is allocated memory **frames** whenever the latter is available.
  - Avoids external fragmentation.
  - Avoids problem of varying sized memory chunks.

Paging (Cont.)

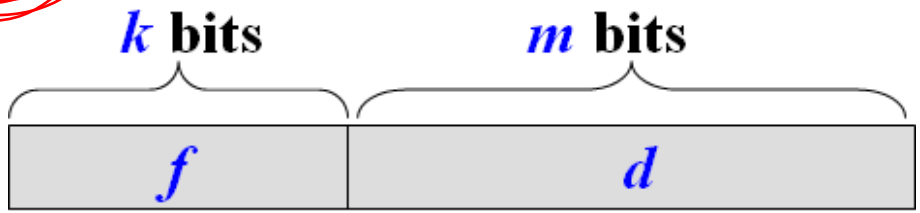
data structure?

- Keep track of all free frames.
- To run a program of size *n* pages, need to find *n* free frames and load program.
- OS maintains a page table for each process.
  - contains the frame location for each page in the process.
  - used to translate logical to physical addresses.
- Internal fragmentation. (-- the last page)
- Every logical address is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains the base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.

Address structure

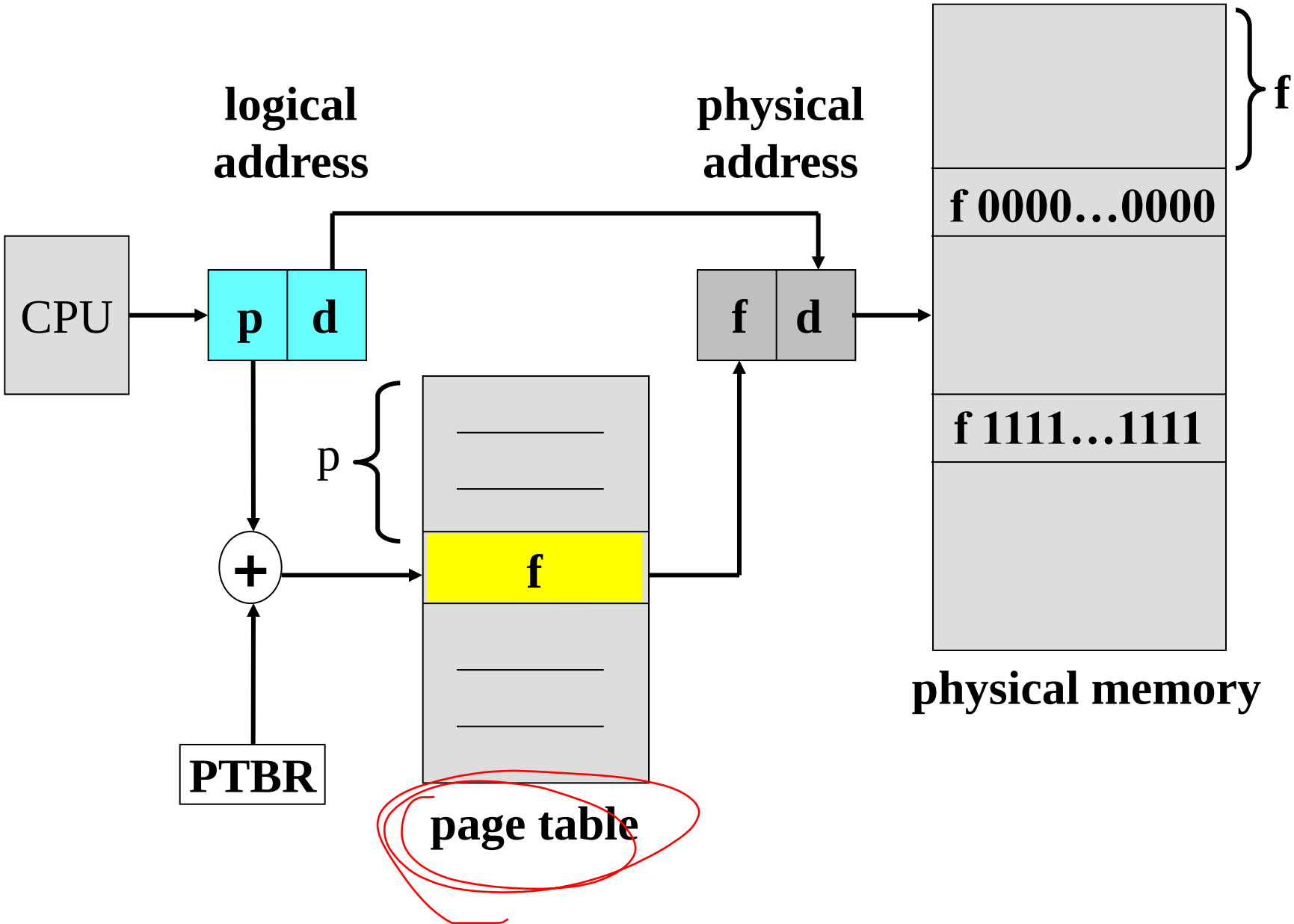


- **Logical address**
  - two part: *p* -- page number, *d* -- offset
  - bits of *p*: decided by the number of pages
  - bits of *d*: decided by the size of page
  - bits of logical address: *n*+*m*

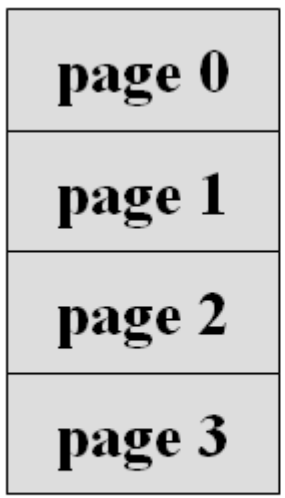


- **Physical address**
  - two part: *f* -- frame number, *d* -- offset
  - bits of *f*: decided by the number of memory frames
  - bits of *d*: decided by the size of frame/page
  - bits of physical address: *k*+*m*

Address Translation Architecture



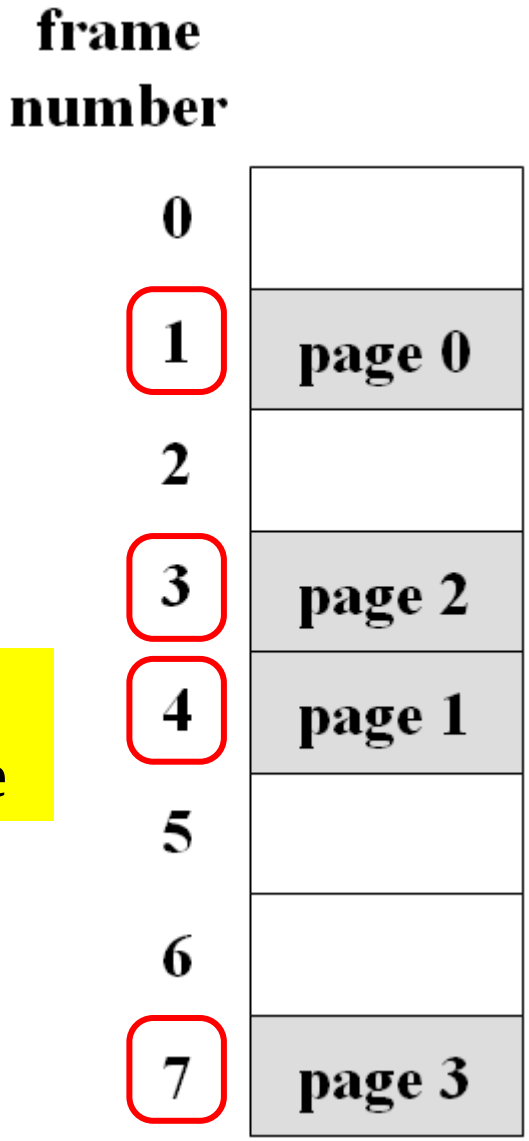
Paging model



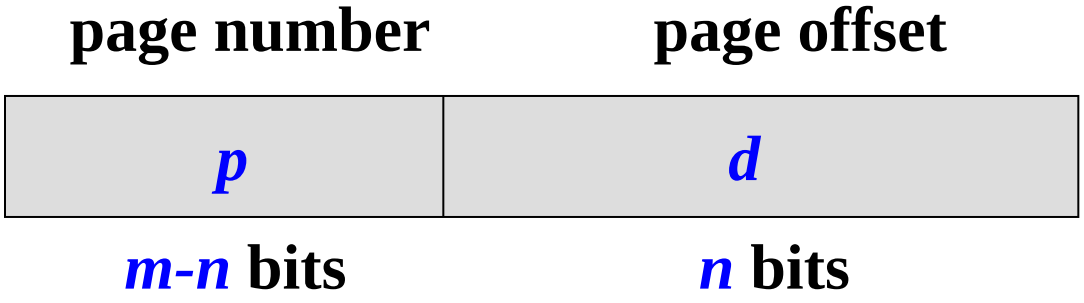
logical memory

0	1
1	4
2	3
3	7

Page table

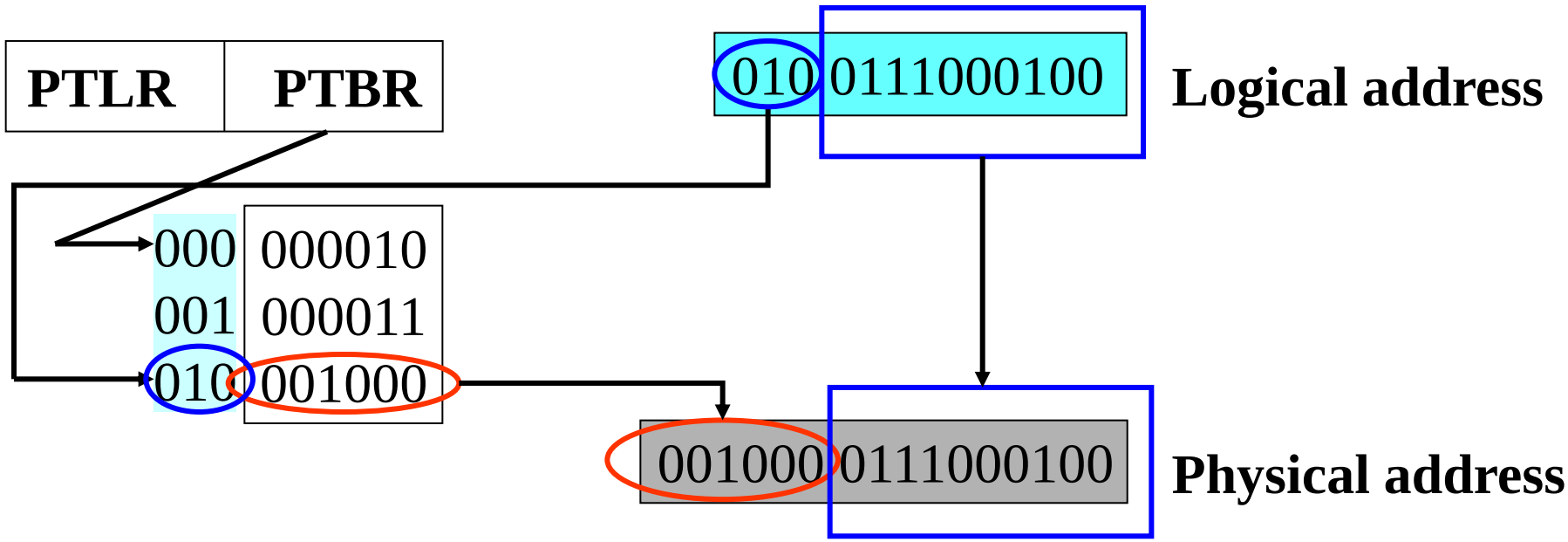


The size of a page is typically a power of 2, varying between 512 bytes and 16MB per page



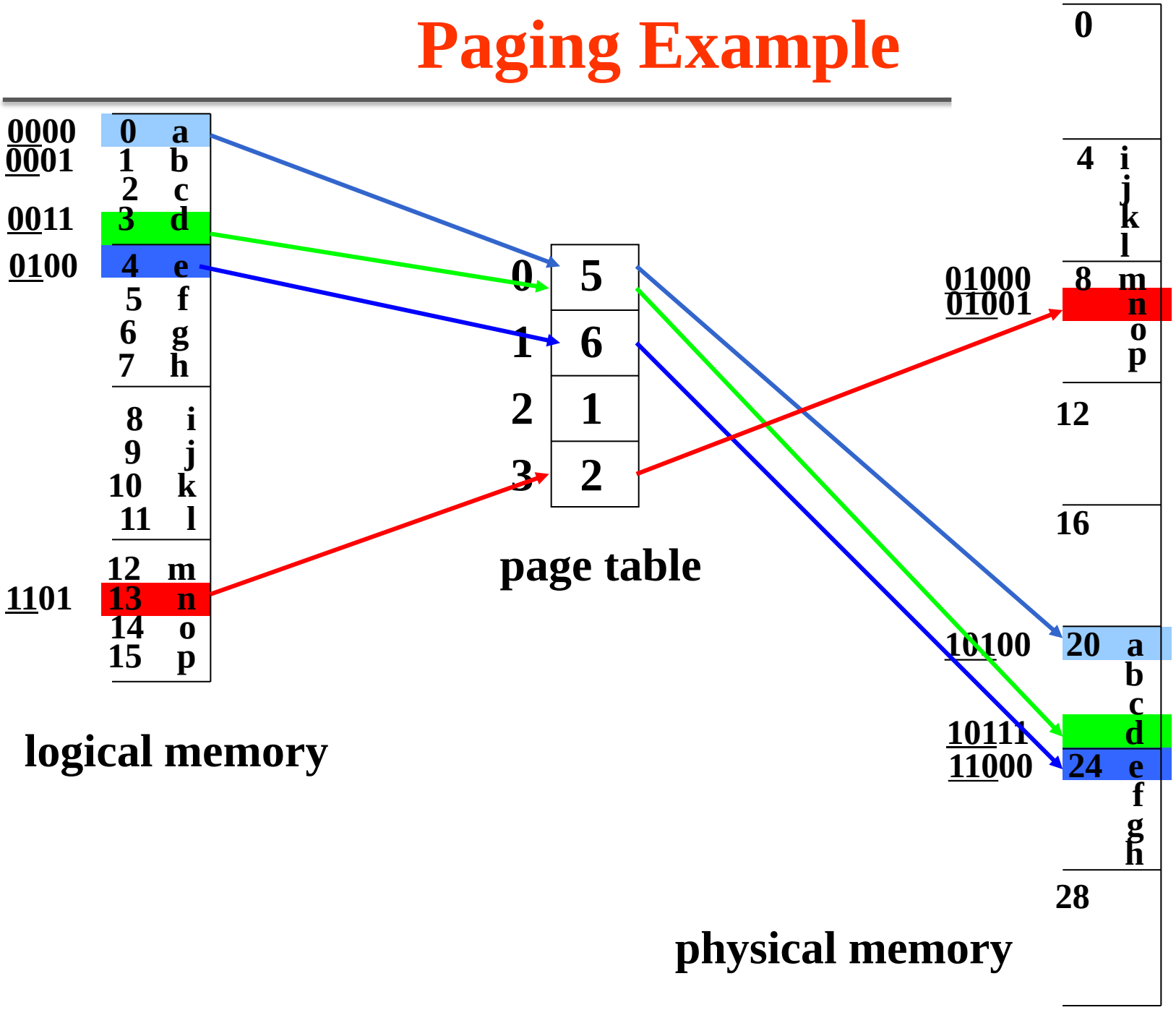
Example of Address Mapping

- **Page size: 1KB, a process can have 8 pages, there are 64 frames.**
- **Length of logical address (LA) is 13bits（3+10）**
- **Length of physical address (PA) is 16bits（6+10）**
- **LA: 2500 = 2*1024+452 ➔ 010 0111000100**



- **PA: 001000 0111000100 ➔ 8*1024+452=8644**

Paging Example



Exercise 4:

- **Consider a system with physical memory of 2048 frames, the logical address space of a process is up to 32 pages, the page size is 2048 bytes.**

  (1) How many bits are there in the logical address?

  (2) How many bits in the logical address refer to virtual page number?

  (3) How many bits are there in the physical address?

  (4) How many bits in the physical address refer to frame number?

  (5) How many bits in the physical address refer to offset in a frame?

Exercise 5:

- **A certain computer provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes.**
  (1) How many bits are there in the logical address?
  (2) How many bits in the logical address refer to virtual page number?
  (3) How many bits are there in the physical address?
  (4) How many bits in the physical address refer to frame number?
  (5) How many bits in the physical address refer to offset in a frame?
  (6) A user process generates the virtual address 11123456, figure out its page number and page offset.
  Explain how the system establishes the corresponding physical location.

Exercise 6:

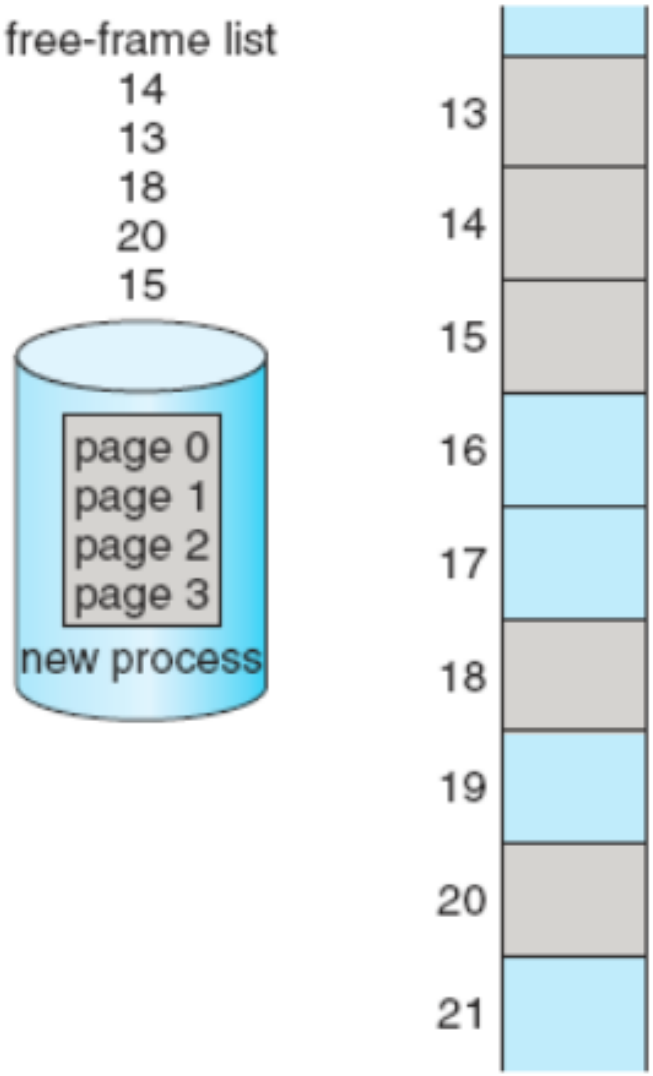
- Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
  a. 3085
  b. 42095
  c. 215201
  d. 650000
  e. 2000001

Logical address / page size → **Quotient ⇒ page number**; **Remainder ⇒ offset**

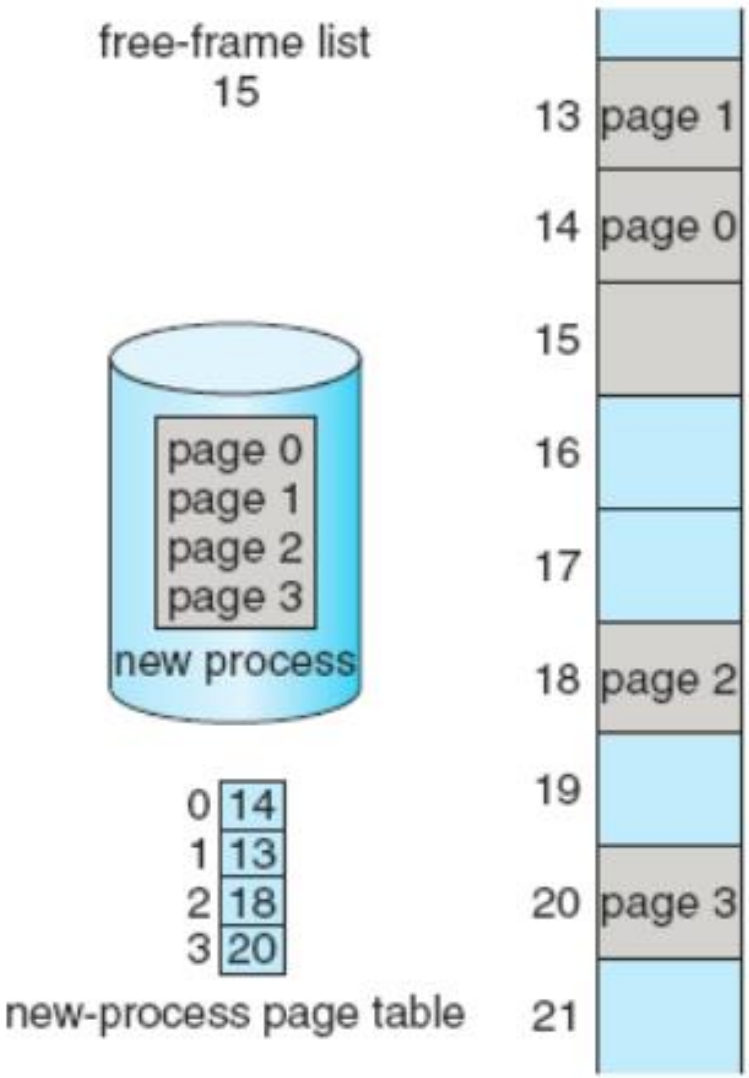
Fragmentation

- Has no external fragmentation
  - Any free frame can be allocated to a process.
- Has internal fragmentation, e.g.
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
  - But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



before allocation



after allocation

Hardware support

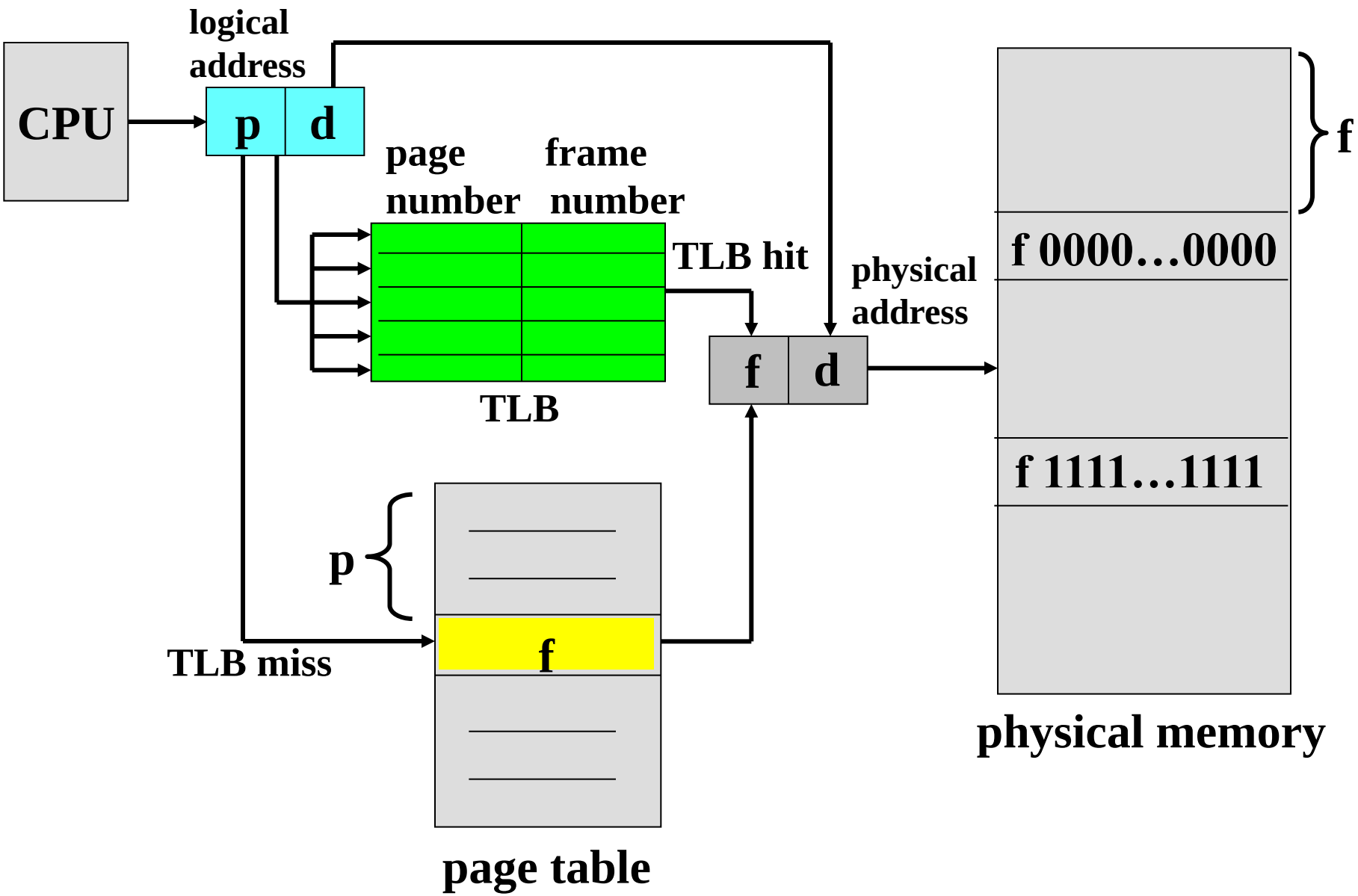
- A set of registers used for page-table (<256 entries)
- For large page-table (e.g. 1 million entries), Page table is kept in main memory.
- *Page-table base register* (PTBR) points to the page table.
- *Page-table length register* (PTLR) indicates size of the page table.
- In this scheme every data/instruction access requires twice memory access. One for the page table entry and one for the data/instruction.
- The twice memory access problem can be solved by the use of a special, small, fast-lookup hardware cache, called *translation look-aside buffers* (TLBs), or *associative memory*.
  - Functions same way as a memory cache.

TLB --Translation Look-aside Buffer

Page #	Frame #

- TLB is associative, high-speed memory.
- Contains page table entries that have been most recently used.
- Each entry in the TLB consists of two parts: a key and a value. (page #, frame #)
- Given a logical address, processor examines the TLB.
  - The given page number is compared with all keys simultaneously.
  - If page table entry is found (a hit), the frame number is retrieved and the physical address is formed.
  - If page table entry is not found in the TLB (a miss), the page number is used to index the process page table.

Paging Hardware With TLB



TLB(Cont.)

- On a TLB miss, value is loaded into the TLB for faster access next time.
  - TLBs typically small (64 to 1,024 entries).
  - Replacement policies must be considered (e.g. LRU).
  - Some entries can be wired down for permanent fast access e.g. entries for key kernel code.
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry.
  - ASID uniquely identifies each process, and is used to provide address-space protection for that process.

Pid	Page#	Frame#

  - ASID allows the TLB to contain entries for several different processes simultaneously.

Effective Access Time (EAT)

- **Hit ratio** – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Associative Lookup = ε nanosecond
- Assume memory cycle time is τ nanosecond
- Hit ratio = α
- **Effective Access Time (EAT)**

$$EAT = (\varepsilon + \tau)\alpha + (\varepsilon + 2\tau)(1 - \alpha)$$

$$= 2\tau + \varepsilon - \tau\alpha$$

- ε=20ns, τ=100ns, α=0.8, EAT=140ns
- ε=20ns, τ=100ns, α=0.98, EAT=122ns

Exercise 7

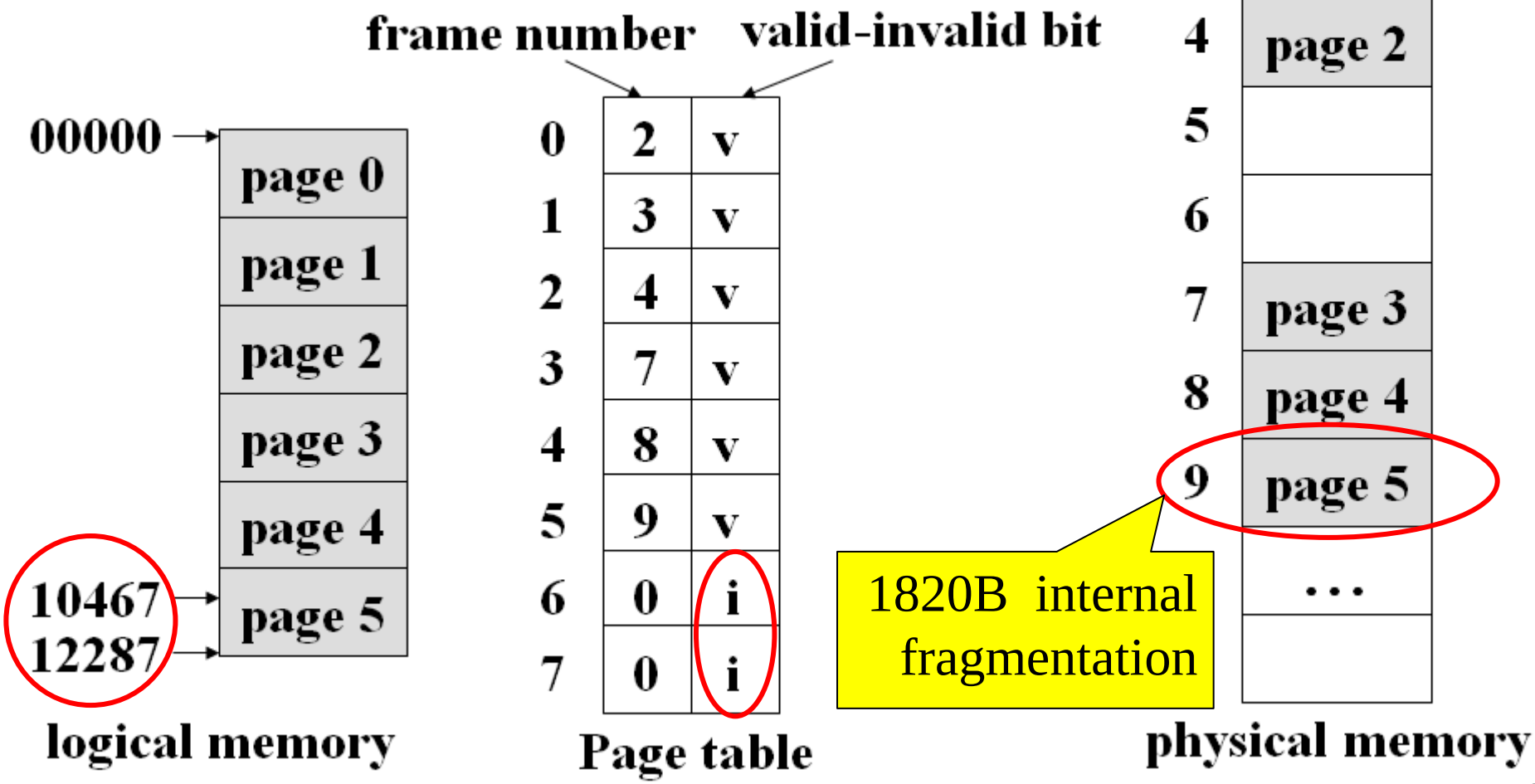
- **Consider a paging system with the page table stored in memory.**
  - **if a memory reference takes 200 nanoseconds, how long does a paged memory reference take?**
  - **If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 4 time, if the entry is there.)**

Memory Protection

- **Memory protection implemented by associating protection bit with each frame.**
  - **Normally these bits are kept in the page table.**
- ***RW* bit, can define a page to be read-write or read-only.**
  - **Can also add more bits to indicate page execute-only, and so on.**
- ***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.**
  - **Or use page-table length register (PTLR).**
- **Any violations result in a trap to the kernel.**

Valid(v)-Invalid(i) Bit

In a system with a 14-bit address space (0 to 16383), page size of 2 KB.
A program with size 10468B.

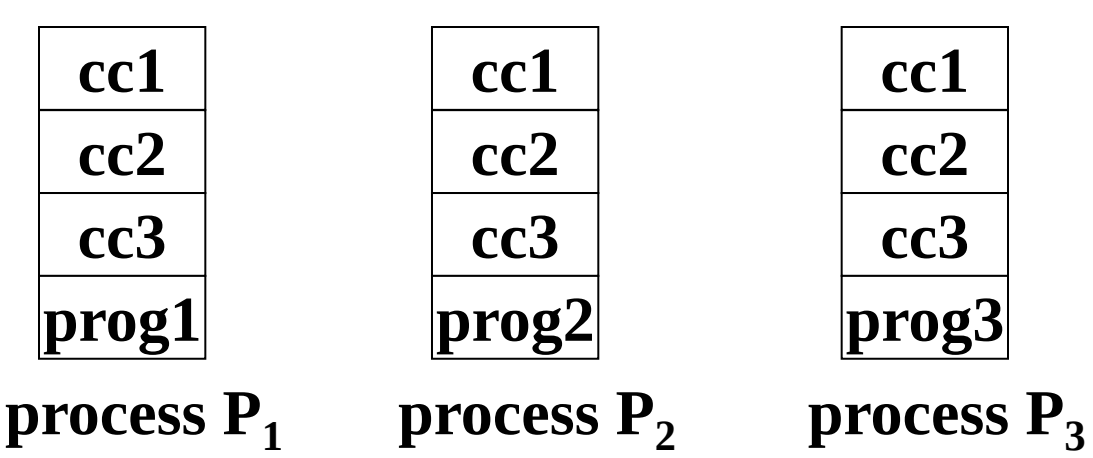


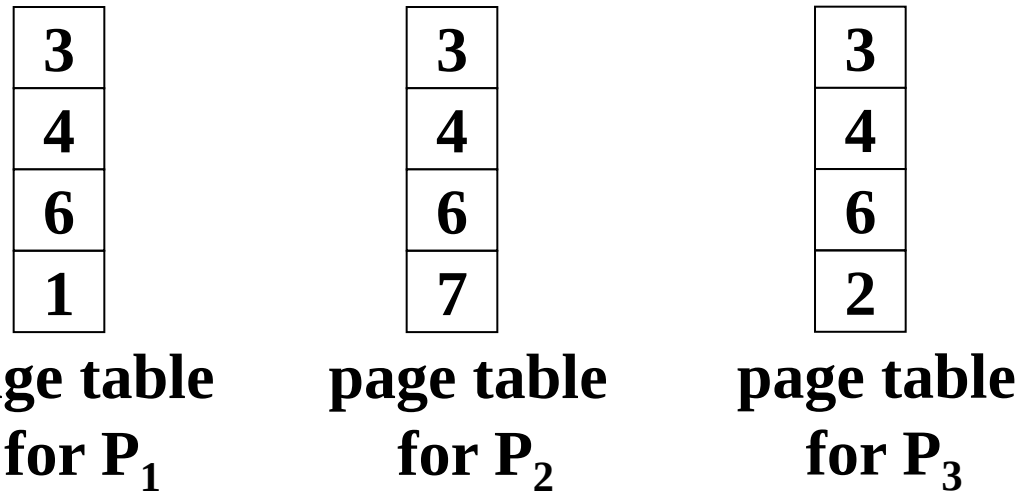
Shared Pages

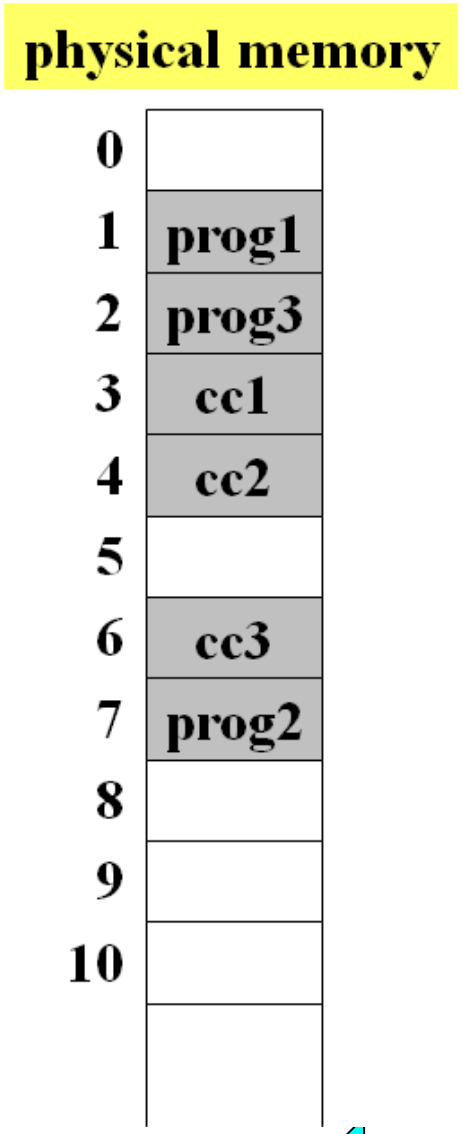
- **Shared code**
  - **One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).**
  - **Shared code must appear in same location in the logical address space of all processes.**
  - **Similar to multiple threads sharing the same process space.**
  - **Also useful for interprocess communication if sharing of read-write pages is allowed.**
- **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 Example

共享 ⇒ 经济、提高效率







End 4

8.5 Structure of Page Table

- Consider a 32-bit logical address space as on modern computers， with 4 KB Page size.
  - How many page table entries?

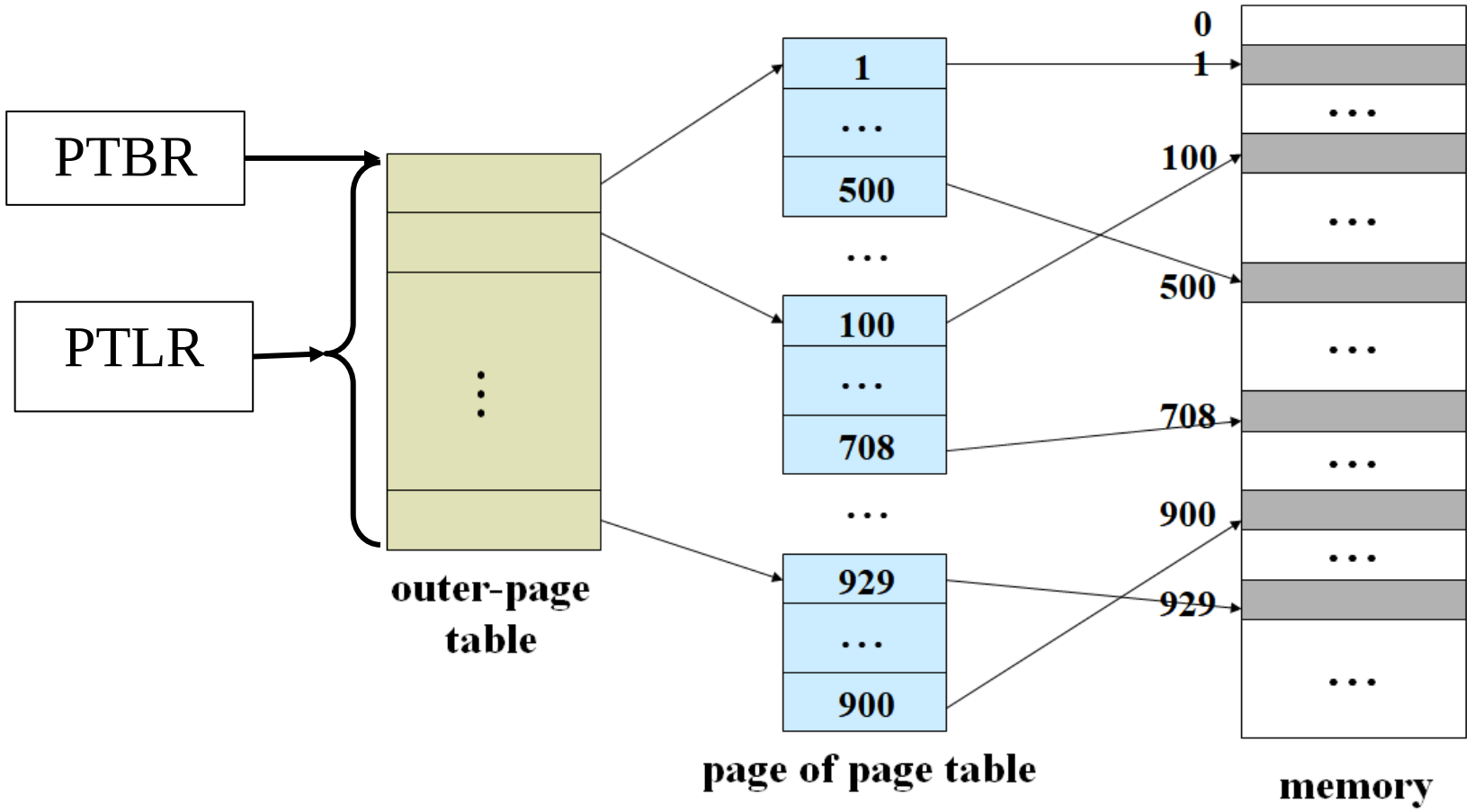
    1 million entries ($2^{32} / 2^{12} = 2^{20}$)
  - If each entry consists of 4 bytes, the size of the page table?

    $2^{20} * 4B = 4$ MB, needs 2^{10} frames
  - Don't want to allocate that contiguously in main memory.
  - Divides the page table into small pieces.

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

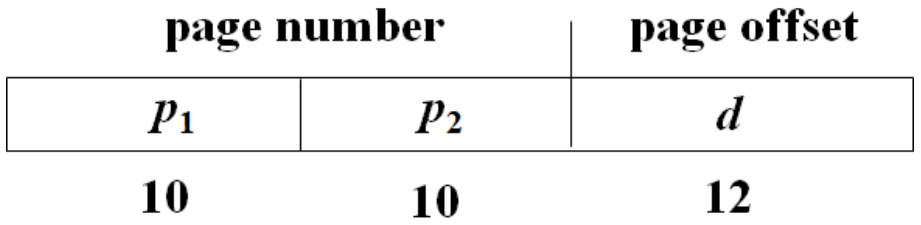
Hierarchical Paging

- A simple technique is a two-level paging algorithm.
  - the page table itself is also paged.



Two-Level Paging Example

- A logical address (on 32-bit machine with 4KB page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 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.
- A logical address is as:

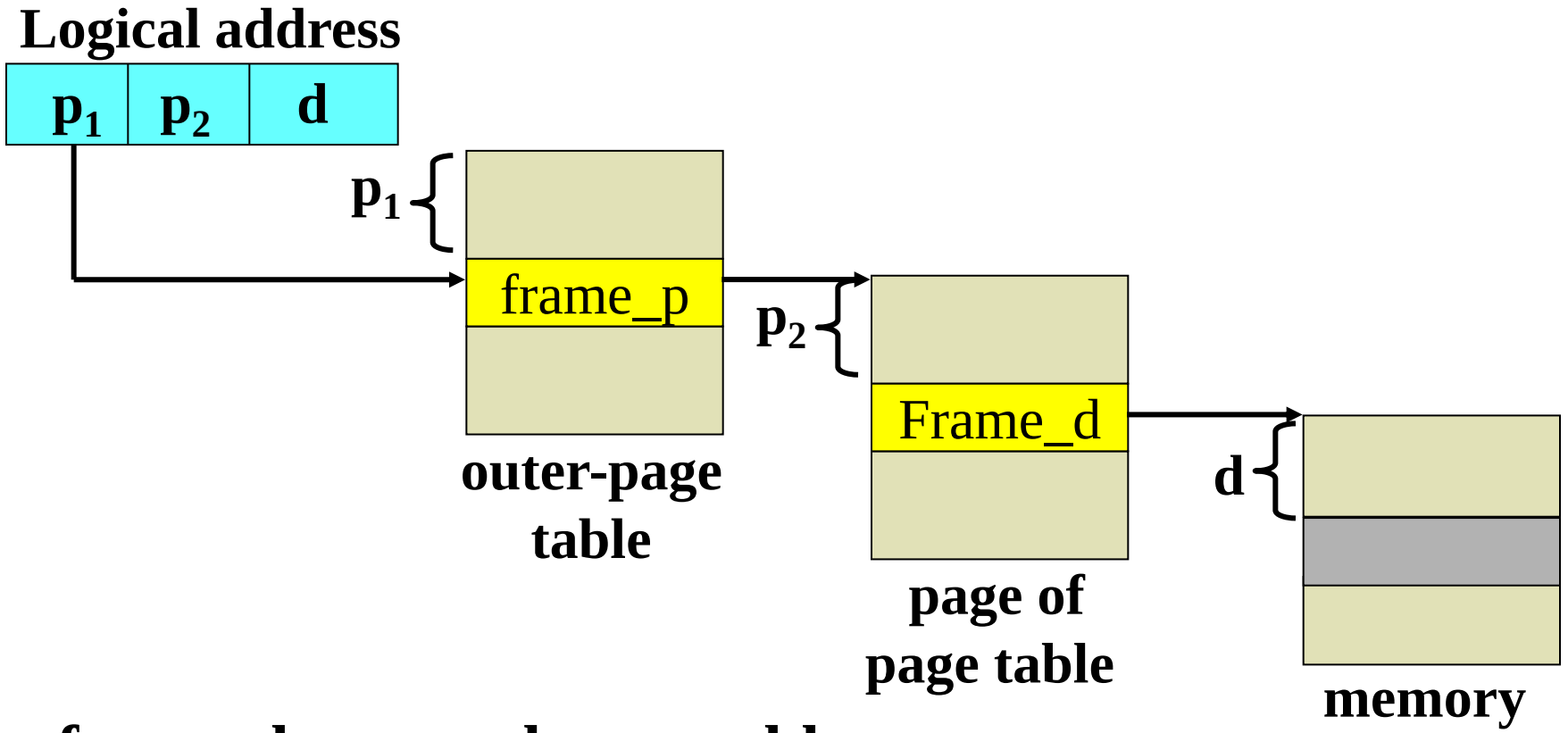


p_1 is an index into the outer page table

p_2 is the displacement within the page of the inner page table.

Address-Translation Scheme

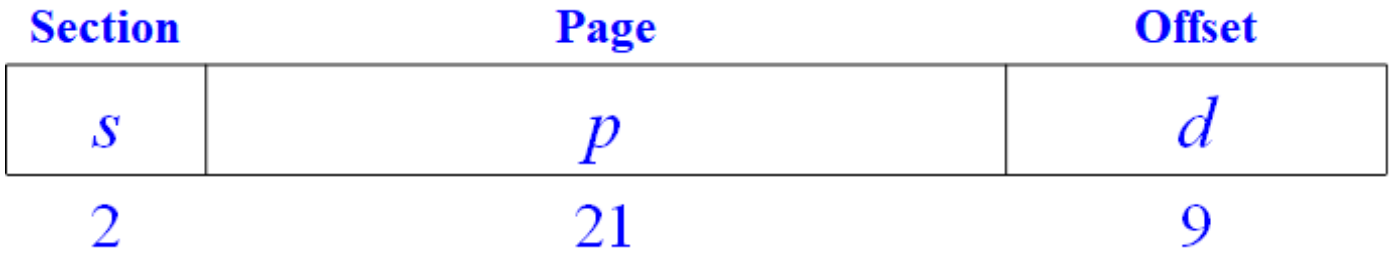
- Address-translation scheme for a two-level 32-bit paging architecture.



- forward-mapped page table.

e.g. VAX Architecture

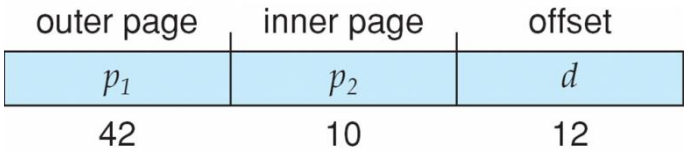
- 32-bit machine with a page size of 512 bytes.
- The logical address space of a process is divided into 4 equal sections, each of which consists of 2^{30} bytes.
- An address on the VAX architecture:

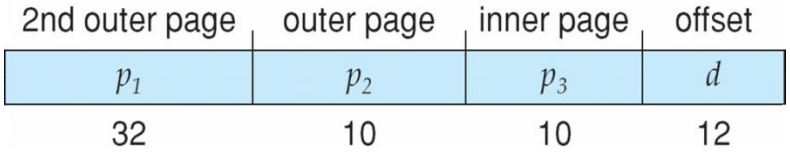


- The size of a one-level page table for a VAX process using one section is:
  - 2^{21} * 4 bytes per entry =8 MB, 2^{14} frames
- 3-level paging architecture
  - 2^{nd} outer: 1 frame, outer: 128 frames, inner: 128*128 frames

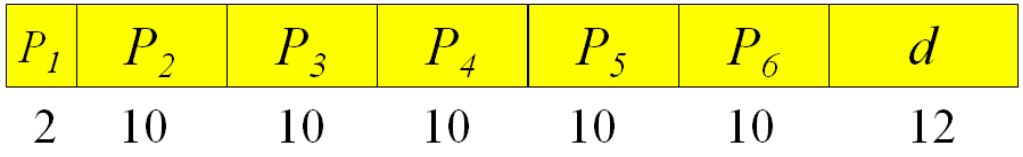
64-bit Logical Address Space

- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries.
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries.
  - Outer page table has 2^{42} entries.
  - Solution： add a 2nd outer page table.
  - The 2nd outer page table is still 2^{32} bytes in size.





  - The 2nd outer page table is also paged.
  - …
  - 7 levels of paging is required to translate each logical address.

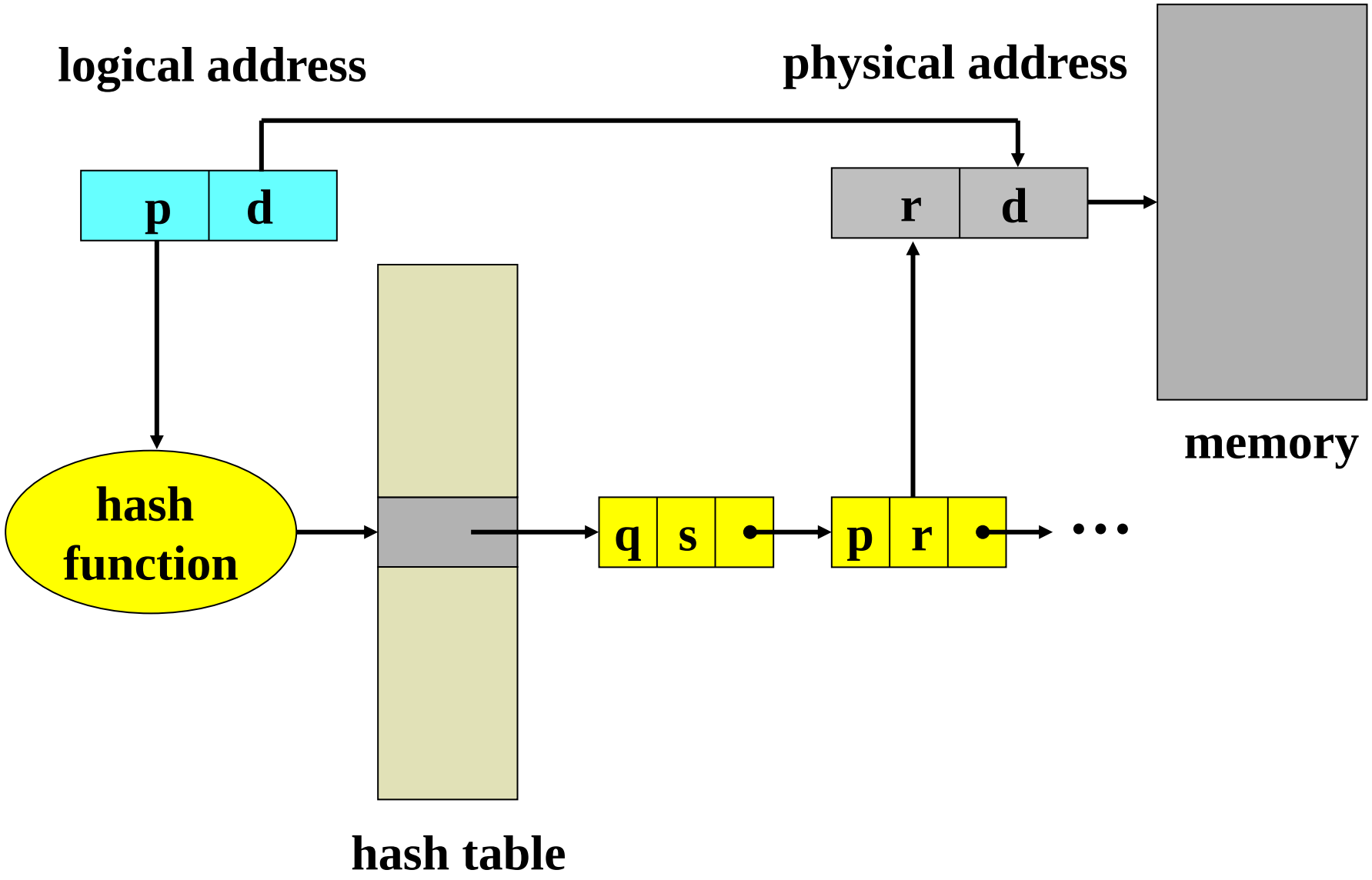


  - A prohibitive number of memory accesses.

Hashed Page Tables

- Common in address spaces > 32 bits.
- Each entry in the hash table contains a linked list of elements that hashed to the same location.
- Each element consists of three fields:
  (1) The virtual page number
  (2) The value of the mapped page frame (frame number)
  (3) A pointer to the next element in the linked list
- The virtual page number is hashed into the hash table, and is compared to field (1) in this chain searching for a match.
  - If a match is found, the corresponding physical frame number (field 2) is extracted.
  - If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

Hashed Page Tables(Cont.)

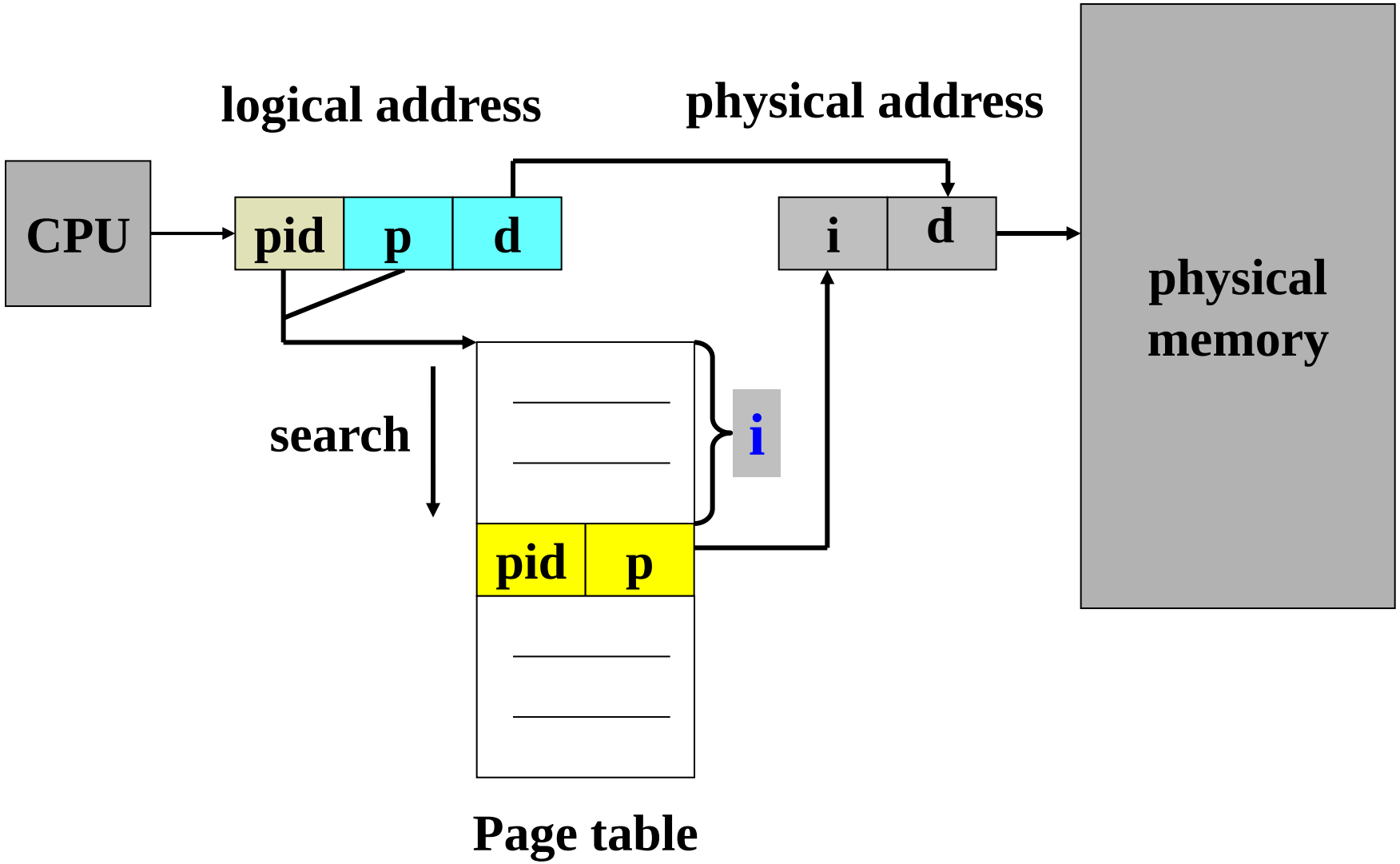


Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages.
- One entry for each real frame of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
  - an address-space identifier (ASID) stored in each entry of the page table.
- A simplified version of the implementation used in the IBM RT, each virtual address consists of a triple:

  < process-id, page-number, offset >

Inverted Page Table Architecture



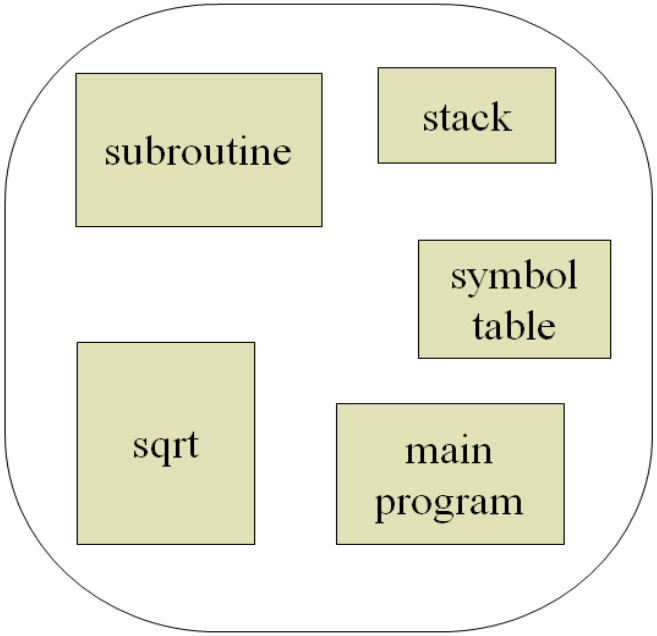
Inverted Page Table (Cont.)

- Decreases the amount of memory needed to store each page table, but increases the amount of 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.
  - TLB can accelerate access
- How to implement shared memory?
  - One mapping of a virtual address to the shared physical address

end 5

8.6 Segmentation

- A program is a collection of segments. A segment is a logical unit such as:
  - main program
  - Procedure / function / method
  - common block
  - Object
  - Stack / symbol table / arrays
  - local variables / global variables
- Segments vary in length.
- Elements within a segment are identified by their offset from the beginning of the segment.



User's View of a Program

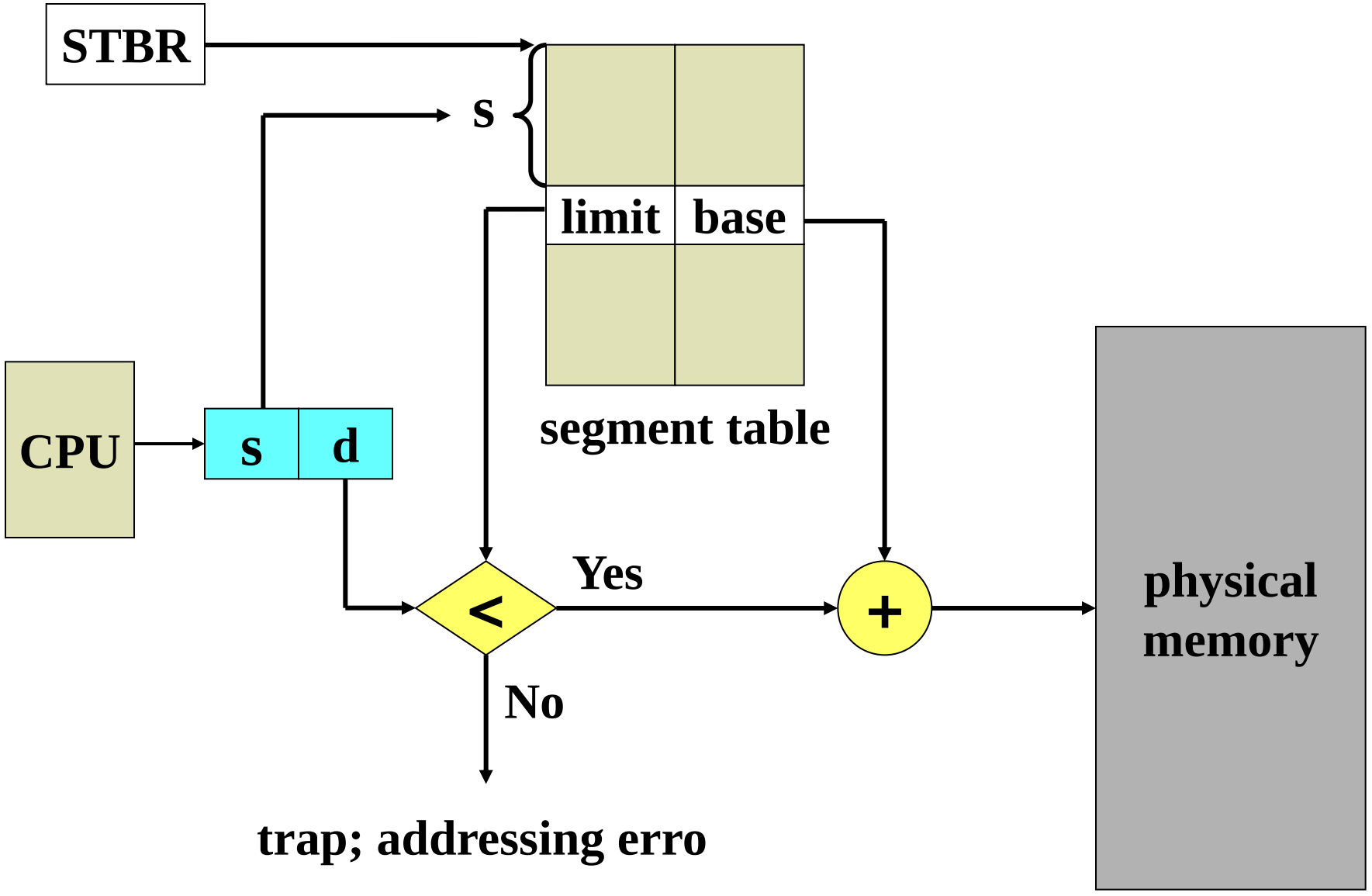
- Segmentation is a memory-management scheme that supports this user view of memory.
- Each segment has a name and a length.

Segmentation Architecture

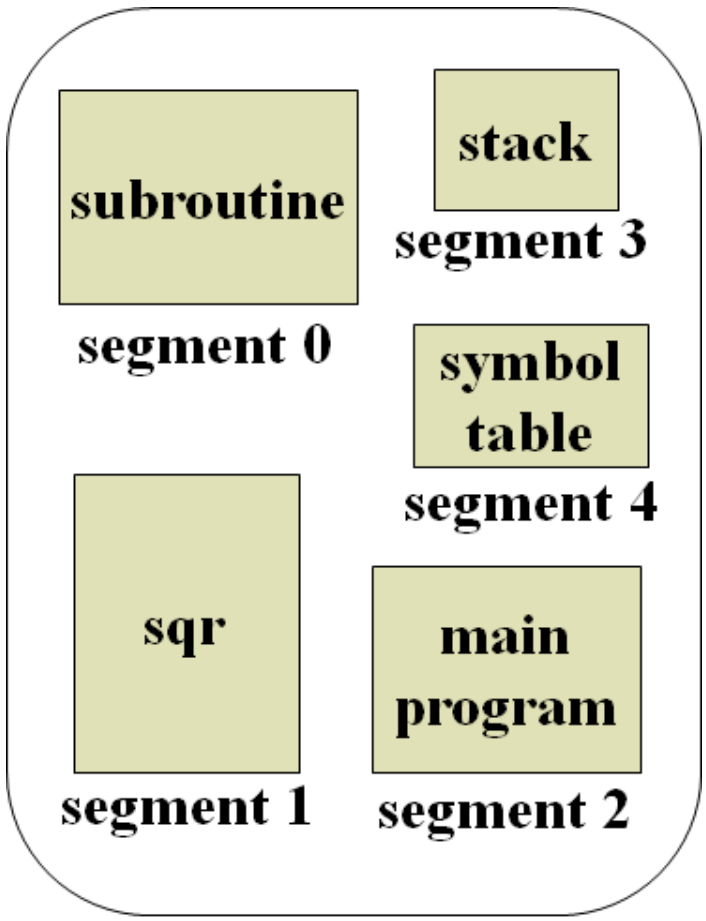
- Logical address consists of two parts: a segment number and an offset: <segment-number, offset>
- *Segment table* -- maps two-dimensional user-defined addresses into one-dimensional physical addresses; each table entry has:
  - Segment base – contains the starting physical address where the segment resides in memory.
  - Segment 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 Hardware



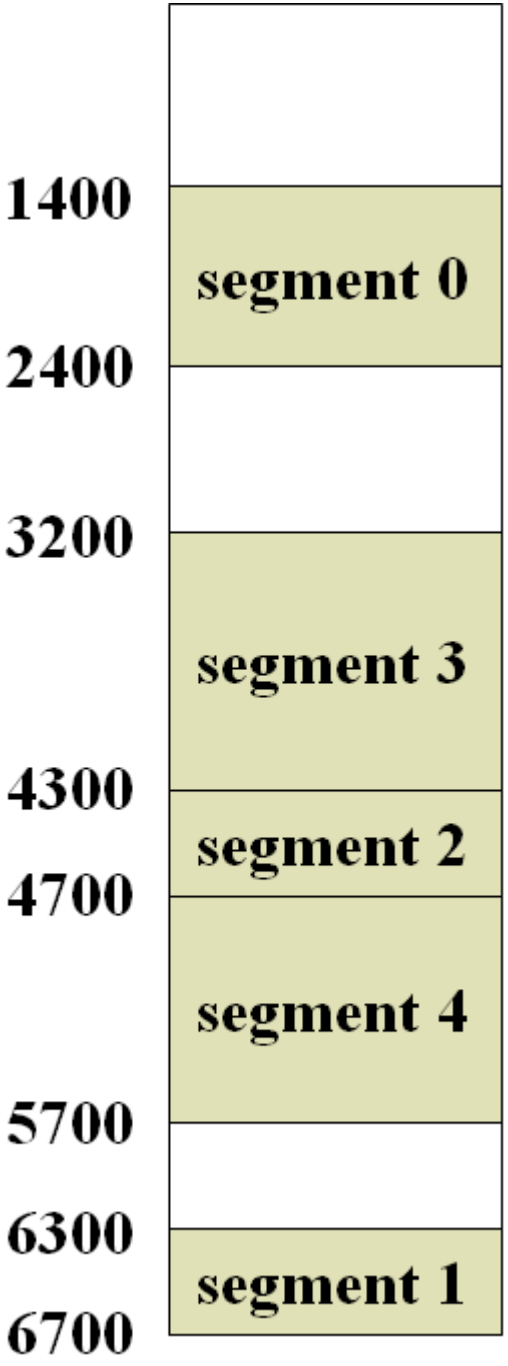
Example of Segmentation



Logical address space

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

segment table



Exercise 8

- **Segment table：**

Segment no.	Limit	base
0	660	2219
1	140	3300
2	100	90
3	580	1237
4	960	1959

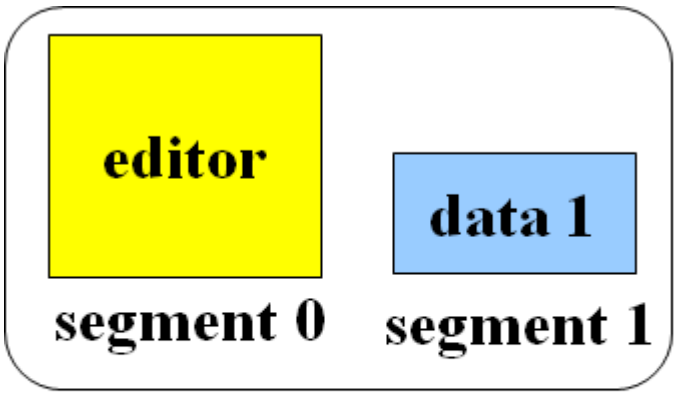
- **mapping logical address to physical address:**
  [0, 432]， [1, 10]， [2, 500]， [3, 400]

- **Summarize the procedure that mapping a logical address to physical address.**

* Segmentation Architecture (Cont.)

- Relocation
  - dynamic
  - by segment table
- Allocation
  - first fit / best fit
  - external fragmentation
- Sharing
  - shared segments
  - same segment number

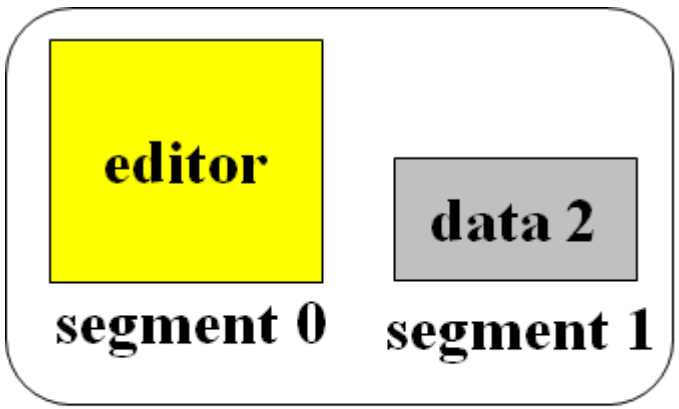
* Sharing of Segments



logical address space
process P_1

	limit	base
0	25286	43062
1	4425	68348

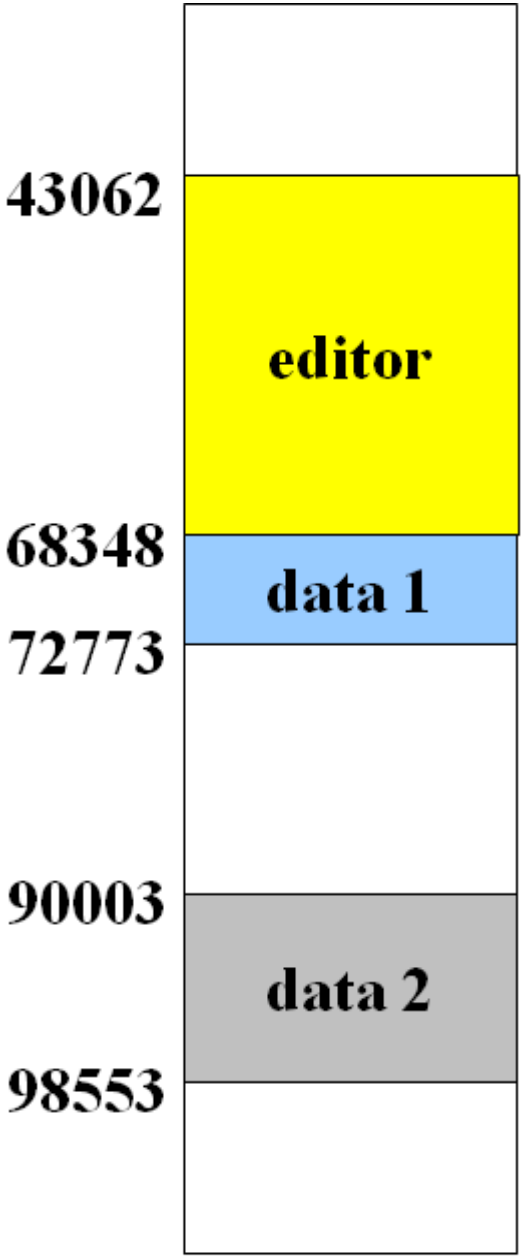
segment table
process P_1



logical address space
process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2



physical memory

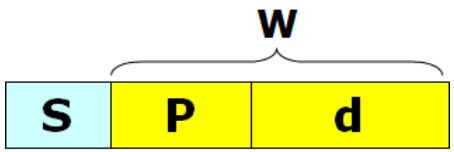
* Segmentation Architecture (Cont.)

- **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.**
- **Problems:**
  - **External fragmentation in memory**
  - **External fragmentation on backing store**
  - **compaction**

* Segmentation With Paging 段页式

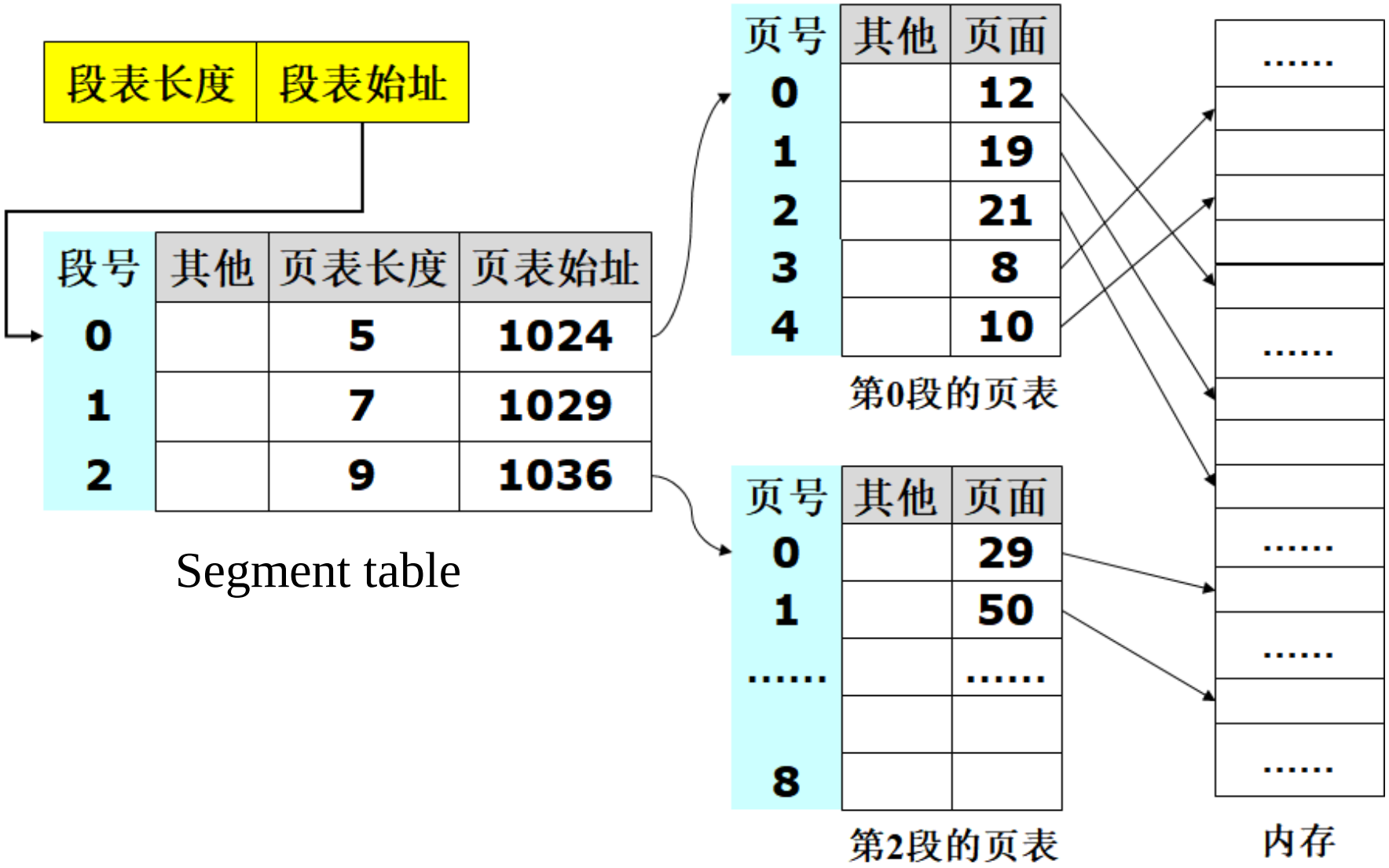
- 页式存储管理的特点
  - 以页面为单位进行内存分配和管理，使得内存中不存在外部碎片、并且产生的内部碎片也很少，有利于提高内存的利用率。
  - 不能兼顾作业本身的逻辑结构，难于实现代码段/数据段的共享。
- 段式存储管理的特点
  - 为用户程序提供一个二维的地址空间，能够反映程序的逻辑结构，便于实现存储保护和共享。
  - 以段为单位进行内存分配和管理，内存中不存在内部碎片，但常常会出现较多的外部碎片，不利于提高内存的利用率。
- 解决方法：段页式存储管理
  - 把段式管理和页式管理结合起来实现对内存的管理。
  - 内存空间被划分成与页大小相同的页面。
  - 对程序既分段又分页
    - 进程中具有独立逻辑功能的程序或数据仍被划分成段。
    - 各段再分页，每个段的各页可以装入内存中不连续的页面中。

* Segmentation With Paging(Cont.)

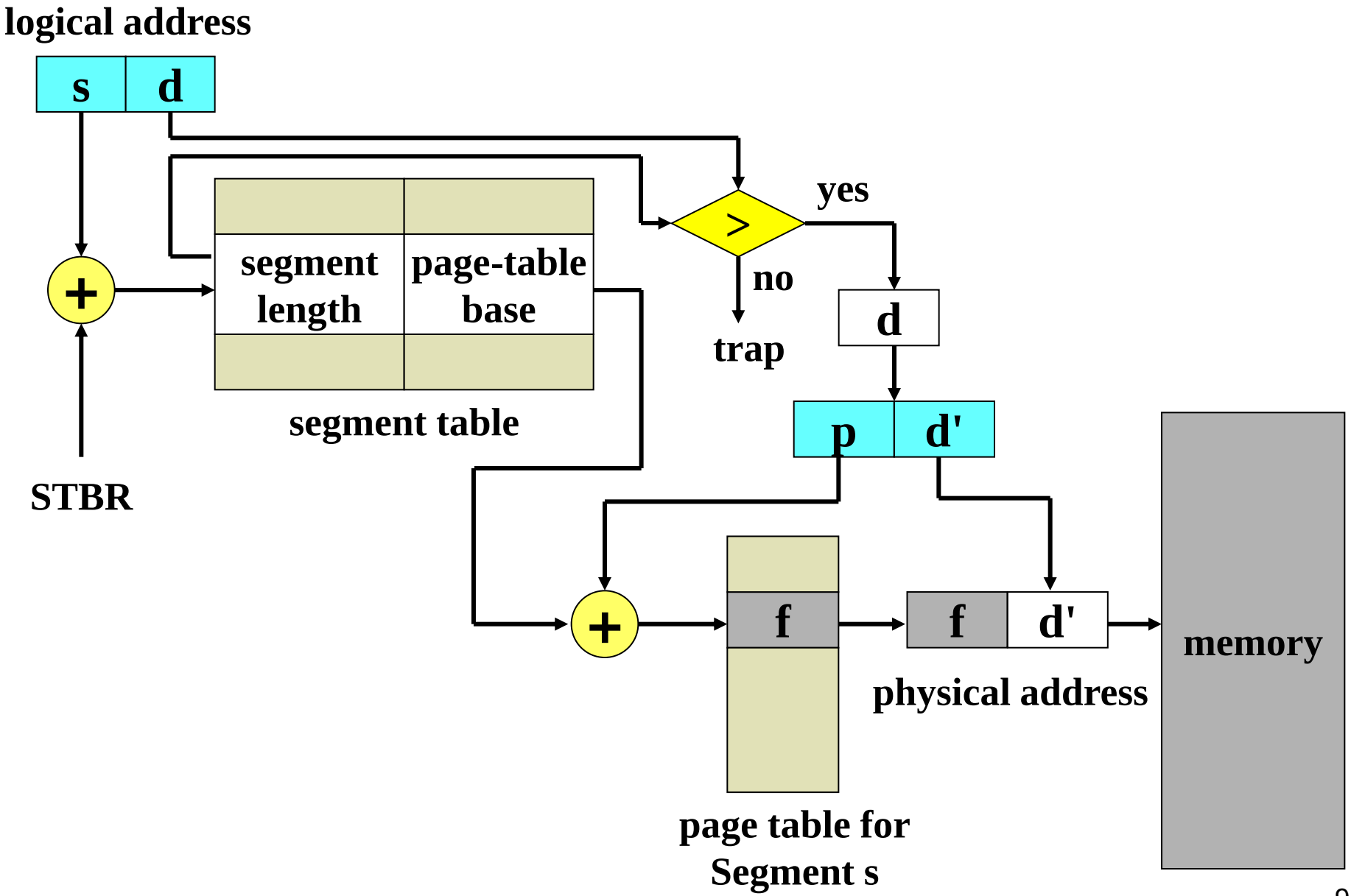


- 虚拟地址的构成
  - 进程的二维逻辑地址空间仍被划分成段，每个段有自己的段号S。
  - 段内的程序或数据，按照内存页面的大小划分为若干页。
  - 进程的虚拟地址包括三部分：段号S、页号P、以及页内地址d。
    - 程序员可见：段号S、段内相对地址W。
    - 页号P和页内地址d是由地址变换机构根据W生成的。
- 主要数据结构：段表和页表
  - 每个进程建立一张段表。
    - 管理内存的分配与释放、存储保护和地址变换等。
    - 每个表项含有对应段的页表的起始地址和页表长度。
  - 每个段建立一张页表。
    - 保存段中的每个逻辑页在物理内存中的内存块号。
  - 段表、页表与内存之间的关系

* 段表、页表与内存之间的关系示例



* Address Translation Scheme

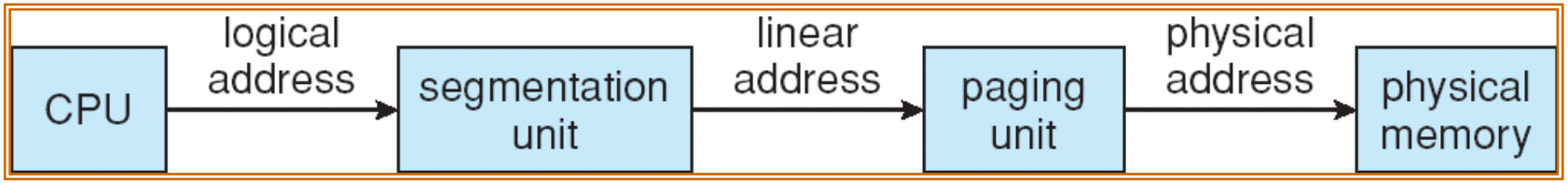


* TLB 快表

- 对内存中指令或数据进行一次存取的话，至少需要访问三次以上的内存。
- 设置TLB表，存放当前最常用的段号S、页号P和对应的内存块号F。
- 当要访问内存时，首先根据段号S和页号P在TLB中查找。
  - TLB命中：取出对应的内存块号F、拼接物理地址、访问内存；
  - TLB失败：查找内存中的段表、页表、取得内存块号F，拼接物理地址，访问内存。
    并将段号S、页号P、内存块号F存入TLB中。

8.7 Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit, Which produces linear addresses
  - Linear address given to paging unit, Which generates physical address in main memory
- The segmentation and Paging units form the equivalent of MMU.



Homework (page 310)

8.3

8.5

8.9

8.12

Thinking about:

8.1

8.10

8.11

8.13