

Hệ Điều Hành

(*Nguyên lý các hệ điều hành*)

Đỗ Quốc Huy

huydq@soict.hust.edu.vn

Bộ môn Khoa Học Máy Tính

Viện Công Nghệ Thông Tin và Truyền Thông

Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ Virtual memory
- ④ Memory management in Intel's processors family

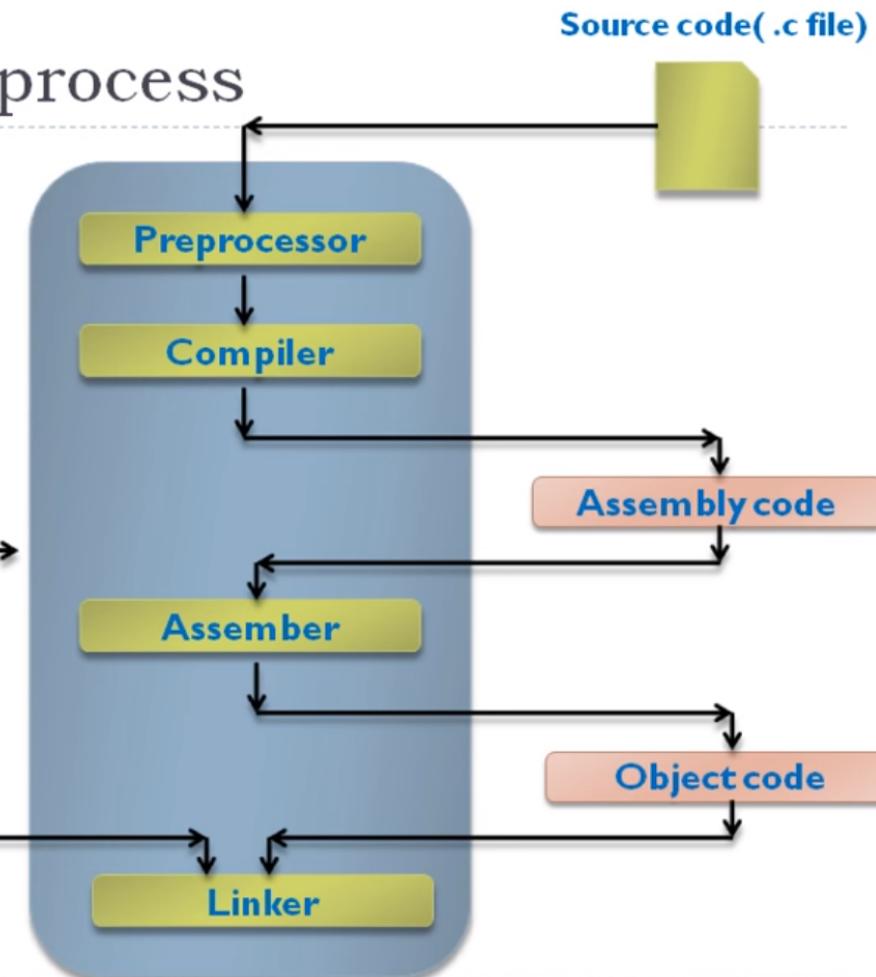
Chapter 3 Memory management

1. Introduction

- Example
- Memory and program
- Address binding
- Program's structures

C compilation process

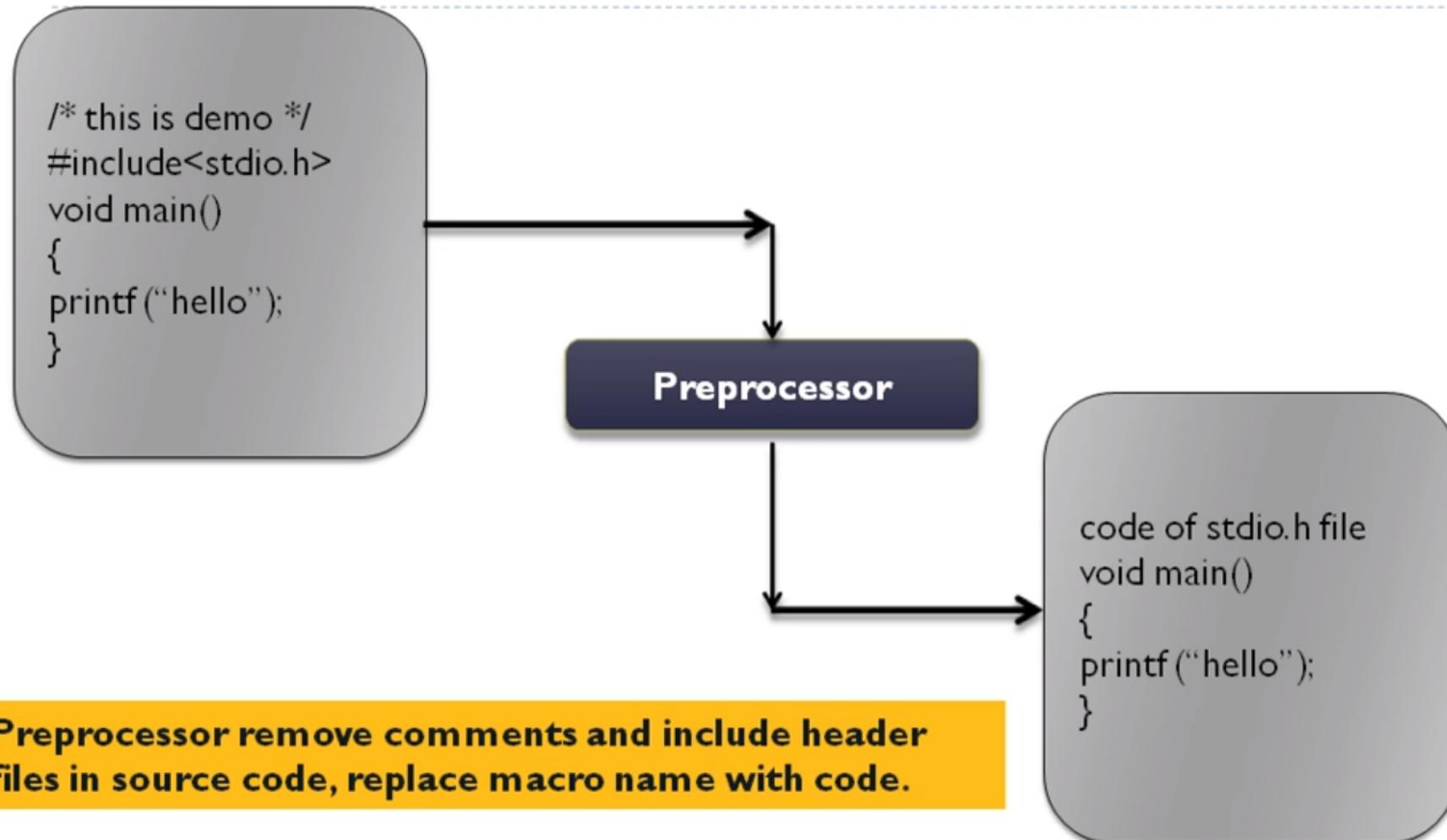
C Compilation process



Components of
Compiler

C compilation process

Preprocessor

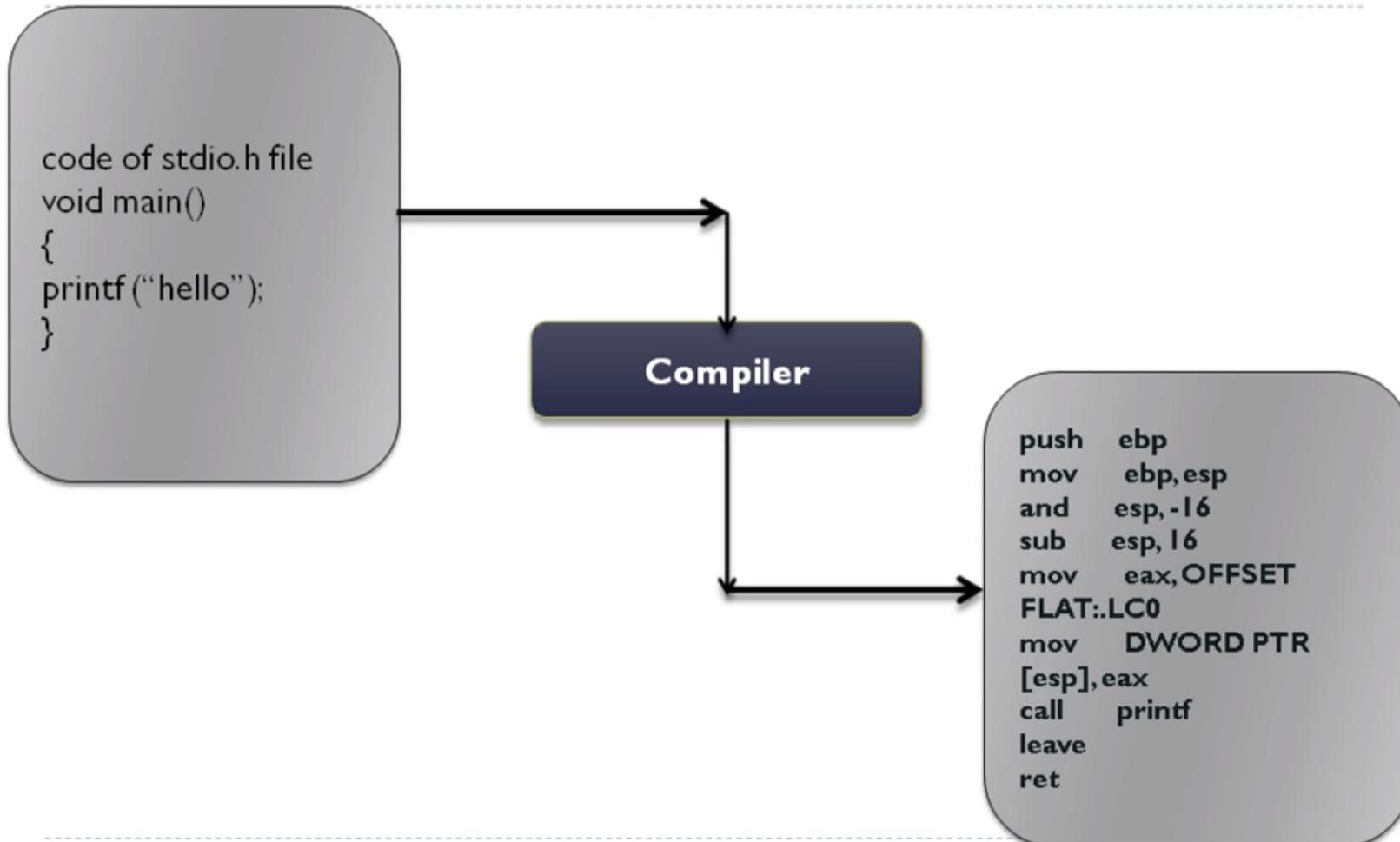


Chapter 3 Memory management

1. Introduction

1.1 Example

C compilation process



Chapter 3 Memory management

1. Introduction

1.1 Example

C compilation process

```
push  ebp  
mov   ebp,esp  
and   esp,-16  
sub   esp,16  
mov   eax,OFFSET  
FLAT:.LC0  
mov   DWORD PTR  
[esp],eax  
call  printf  
leave  
ret
```

Assembler

```
00101001100101001  
10101111010100101  
01010010101011010  
01010101010010101  
0100110011111100  
10001010010101001  
01011110111111111  
11110100000000111  
10010010
```

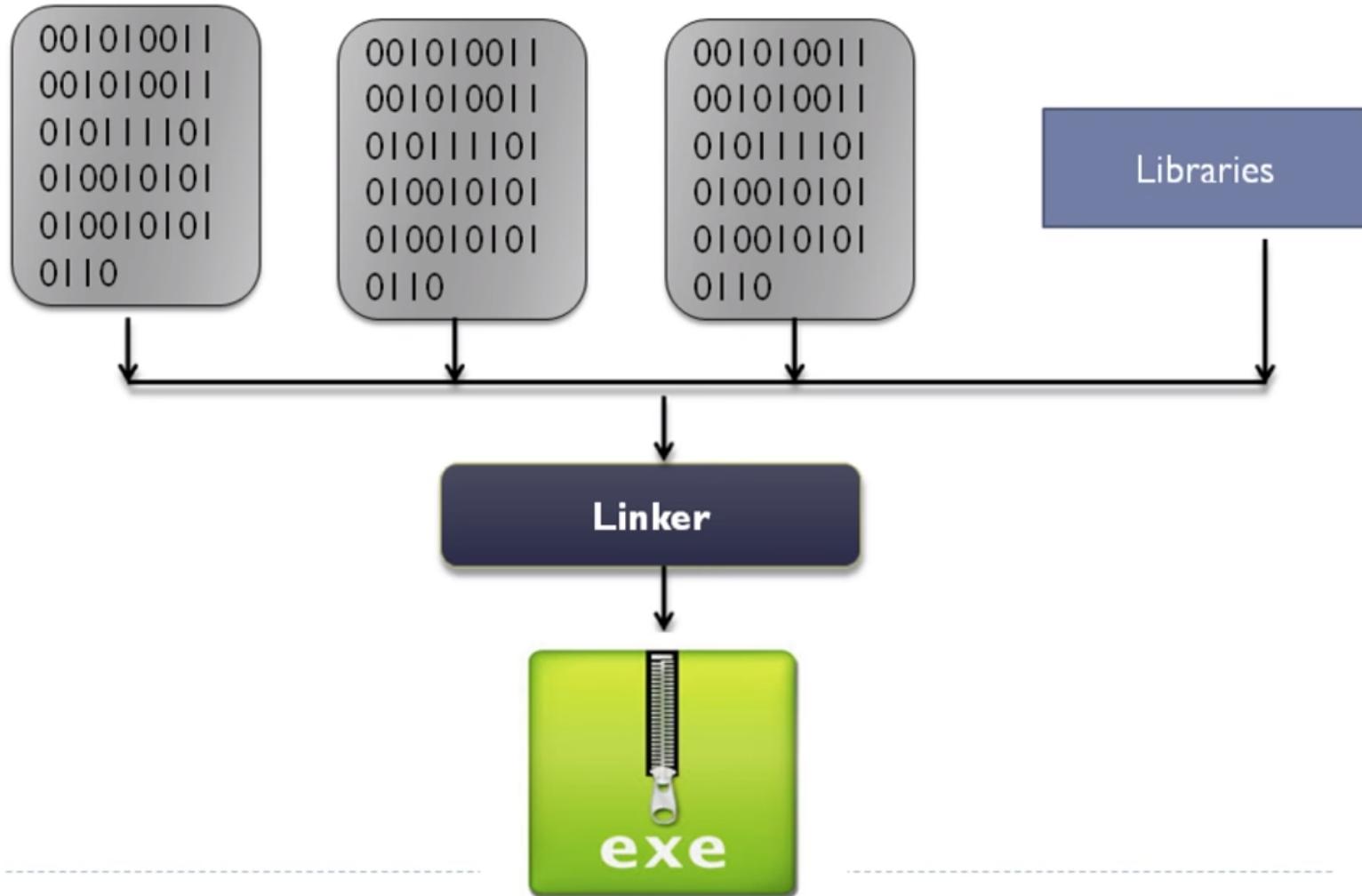
Assembler convert assemble code into object code.

Chapter 3 Memory management

1. Introduction

1.1 Example

C compilation process



Chapter 3 Memory management

1. Introduction

1.1 Example

Example: Generate programs from multi modules

Toto project

file main.c

```
#include <stdio.h>
extern int x, y;
extern void toto();
int main(int argc, char *argv[]) {
    toto();
    printf("KQ: %d \n", x * y);
    return 0;
}
```

file M1.c

```
int y = 10;
```

file M2.c

```
int x;
extern int y;
void toto() {
    x = 10 * y;
}
```

Result

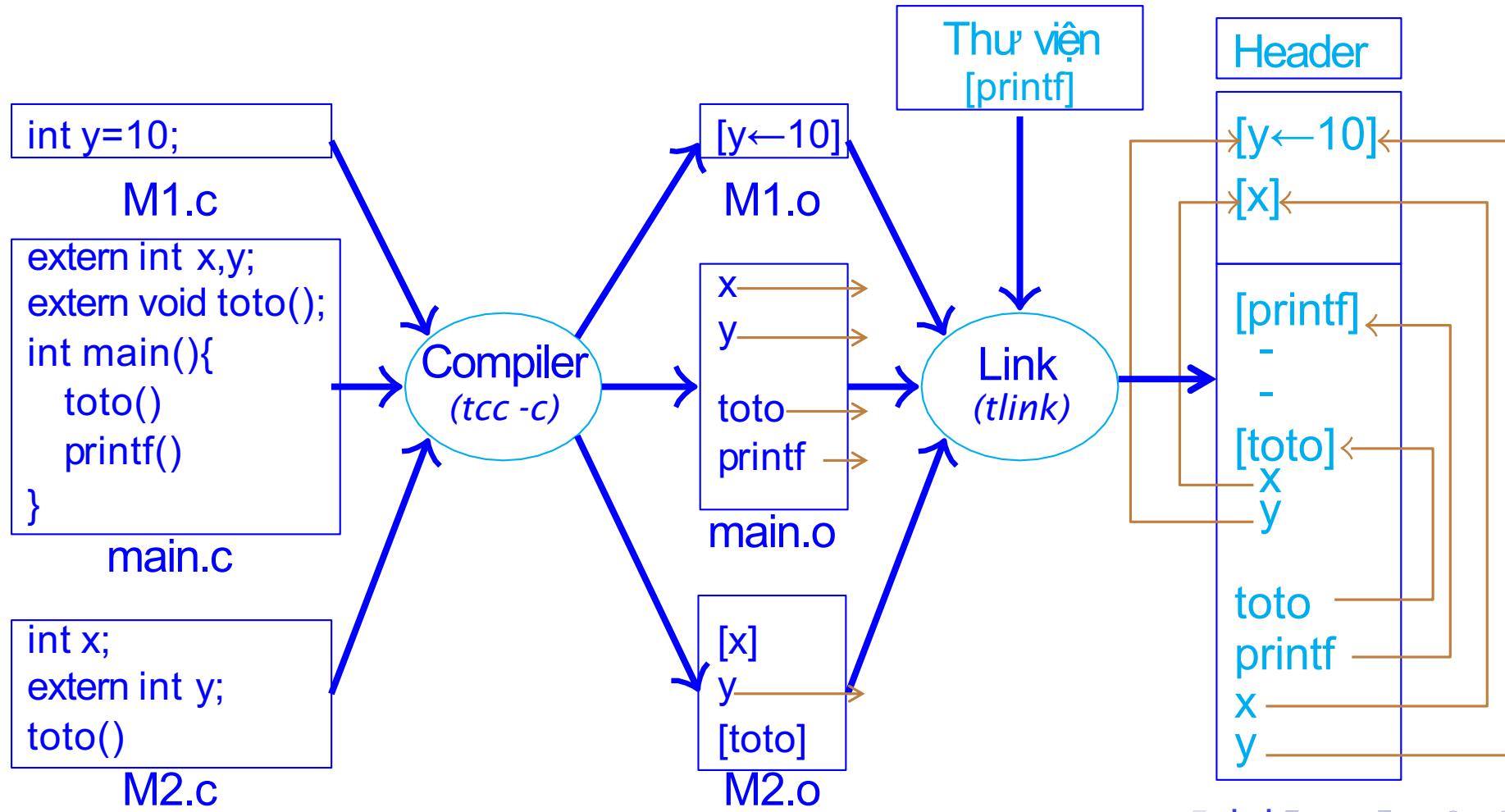
```
KQ: 1000
```

Chapter 3 Memory management

1. Introduction

1.1 Example

toto project compilation process



Chapter 3 Memory management

1. Introduction

1.2. Memory and program

- Example
- **Memory and program**
- Address binding
- Program's structures

Chapter 3 Memory management

1. Introduction

1.2. Memory and program

Memory leveling

- Memory is an important system's resource
 - Program must be kept inside memory to run
- Memory is characterized by size and access speed
- Memory leveling by access speed

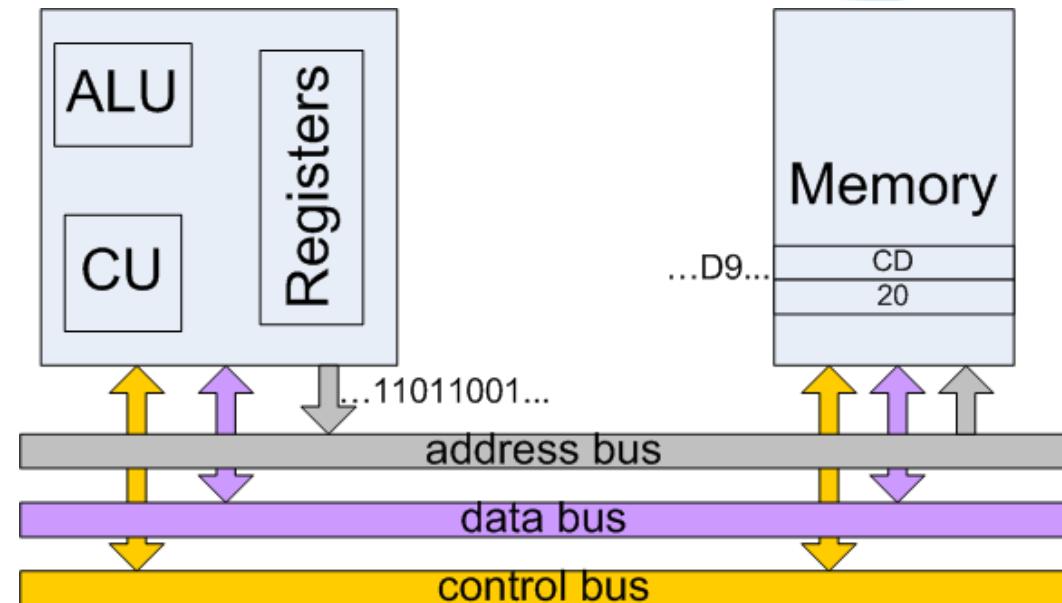
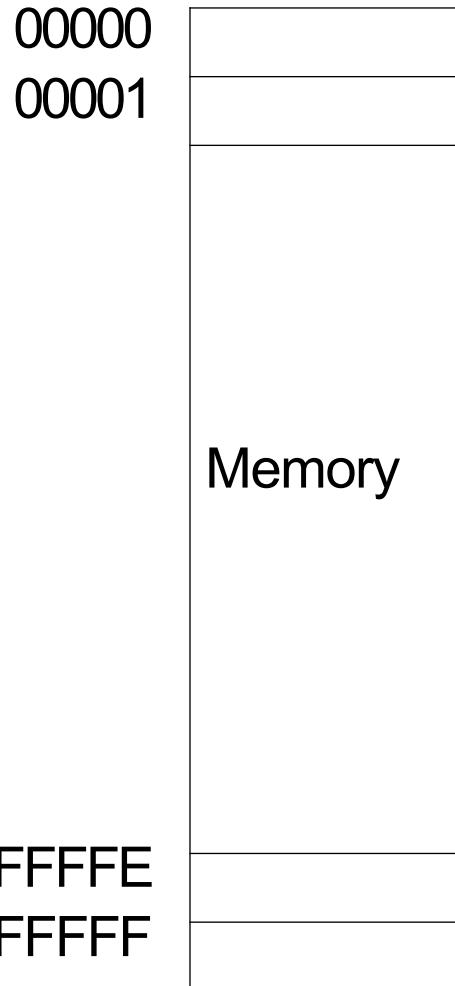
Memory type	Size	Speed
Registers	bytes	CPU speed (η s) 10 nano seconds
Cache on processor	Kilo Bytes	100 nanoseconds e
Cache level 2	KiloByte-MegaByte	Micro-seconds
Main memory	MegaByte-GigaByte	Mili-Seconds 10
Secondary storage(Disk)	GigaByte-Terabytes	Seconds
Tape, optical disk	Unlimit	

Chapter 3 Memory management

1. Introduction

1.2. Memory and program

Main memory



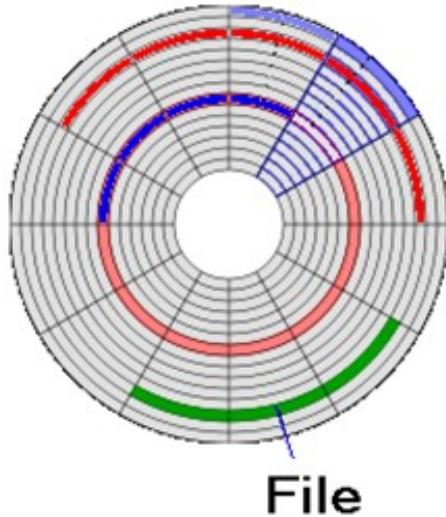
- Used for running program and data
- Array of memory block with size of bytes, words
- Each memory block has an address
 - **Physical address**

Chapter 3 Memory management

1. Introduction

1.2. Memory and program

Program



0101011001111001010101010101101

Header Data Code

- Store on external storage devices
- Executable binary files
 - File's parameters
 - Machine instruction (binary code),
 - Data area (global variable), . .
- Must be brought into internal memory and put inside a process to be executed (process executes program)
- Input queue
 - Set of processes kept in external memory (*normally: disk*)
 - Wait to be brought into internal memory and execute

Execute a program

- Load the program into main memory
 - Read and analysis executable file (e.g. *.com, file *.exe)
 - Ask for a memory area to load program from disk
 - Set values for parameters, registers to a proper value
- Execute the program
 - CPU reads instructions in memory at location determined by program counter
 - 2 registers CS:IP for Intel's family processor (e.g. : 80x86)
 - CPU decode the instruction
 - May read more operand from memory
 - Execute the instruction with operand
 - If necessary, store the results into memory at a defined location
- Finish executing
 - Free the memory area that allocated to program
- Problem
 - Program may be loaded into any location in the memory
 - When program is executed, a sequence of addresses are generated
- How to access memory?

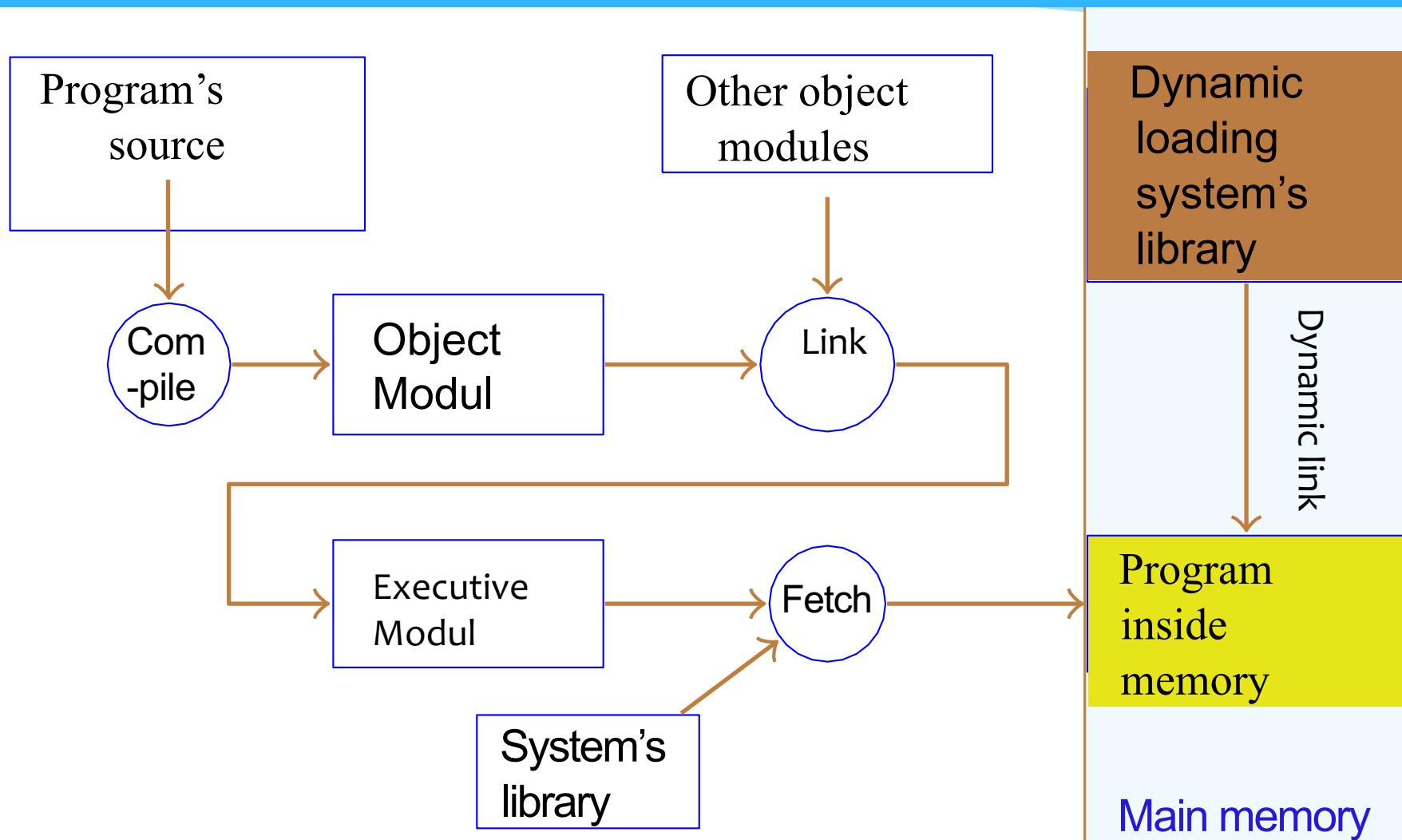
Chapter 3 Memory management

1. Introduction

1.3. Address binding

- Example
- Memory and program
- **Address binding**
- Program's structures

Application program processing steps



Types of address

- **Symbolic**

- Name of object in the source program
- Example: counter, x, y,...

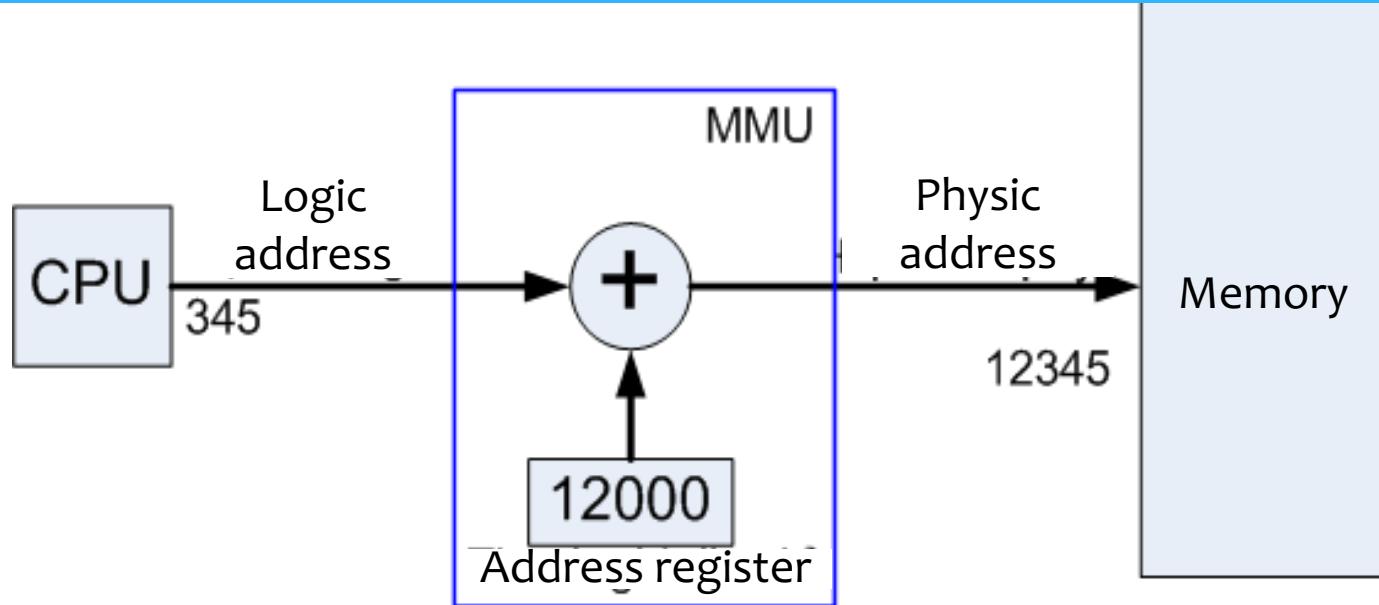
- **Relative address**

- Generated from symbolic address by compiler
- Relative position of an object from the module's first position
 - Example: Byte number 10 from the begin of module

- **Absolute address**

- Generate from relative address when program is loaded into memory
 - For IBM PC: relative address <Seg :Ofs> \rightarrow Seg * 16+Ofs
- Object's address in physical memory – physical address
- Example: JMP 010Ah \Rightarrow jump to the memory block at 010Ah at the same code segment (CS)
 - if CS=1555h, jump to location: 1555h*10h+010Ah =1560Ah

Physical address-logic address



- Logic address
 - Generate from process, (CPU brings out)
 - Converted to physical address when access to object in the program by the Memory management unit (MMU)
- Physical address
 - Address of an element (byte/word) in main memory
 - Correspond to the logic address bring out by CPU
- Program work with logical address

Chapter 3 Memory management

1. Introduction

1.4. Program's structures

- Example
- Memory and program
- Address binding
- Program's structures

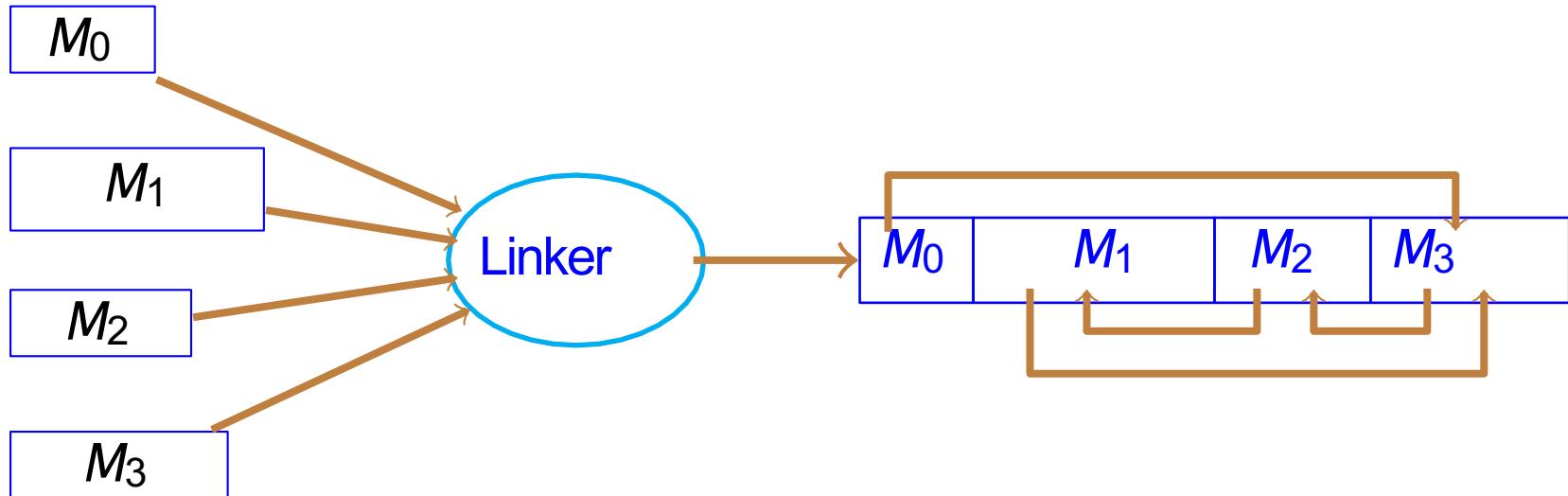
① Linear structure

② Dynamic loading structure

③ Dynamic link structure

④ Overlays structure

Linear structure



- After linking, modules are merged into a complete program
 - Contain sufficient information to be able to execute
 - External pointers are replaced by defined values
 - To execute, required only one time to fetch into the memory

Linear structure

- Advantages
 - Simple, easy to link and localizing the program
 - Fast to execute
 - Highly movable
- Disadvantages
 - Waste of memory
 - Not all parts of the program are necessary for the program's execution
 - It's not possible to run the program that larger than physical memory's size

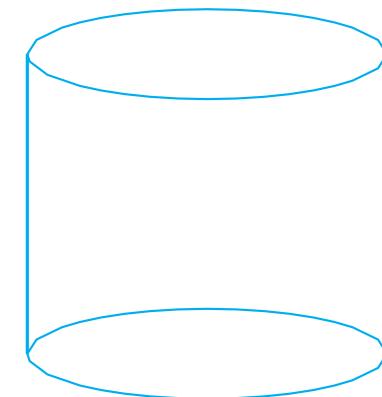
Dynamic loading structure

M_0

M_1

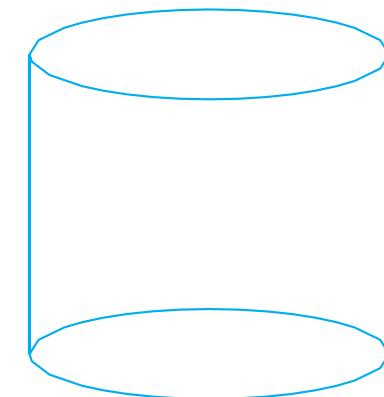
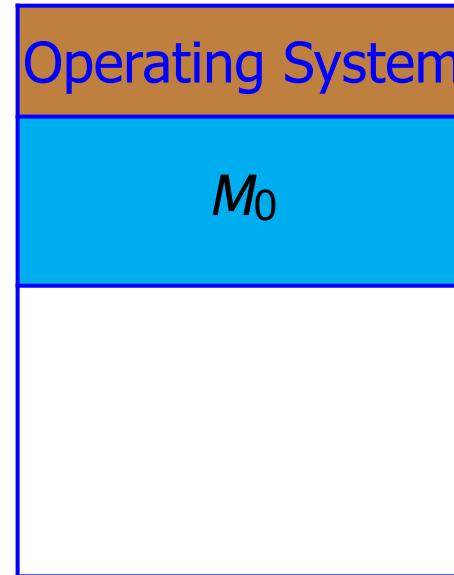
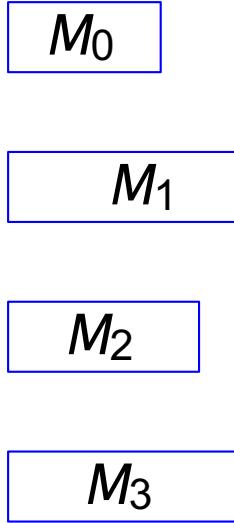
M_2

M_3



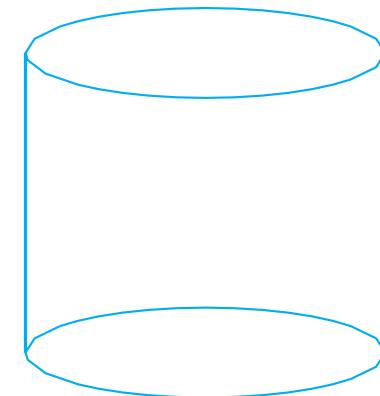
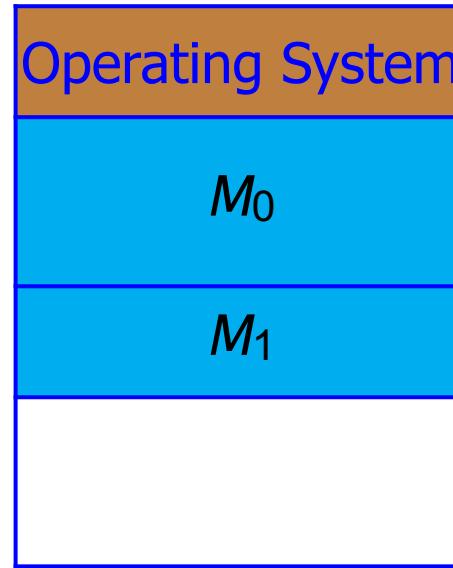
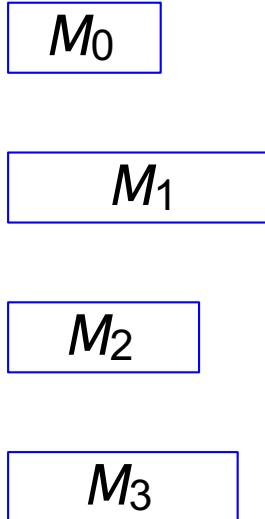
- Each module is edited separately

Dynamic loading structure



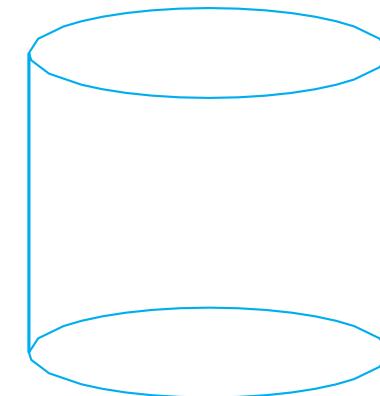
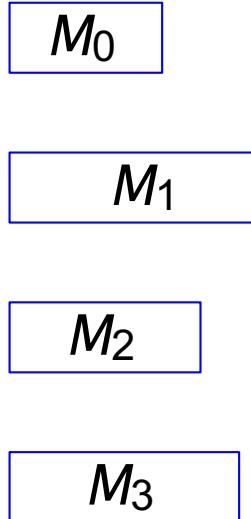
- Each module is edited separately
- When executing, system will load and localize the main module

Dynamic loading structure



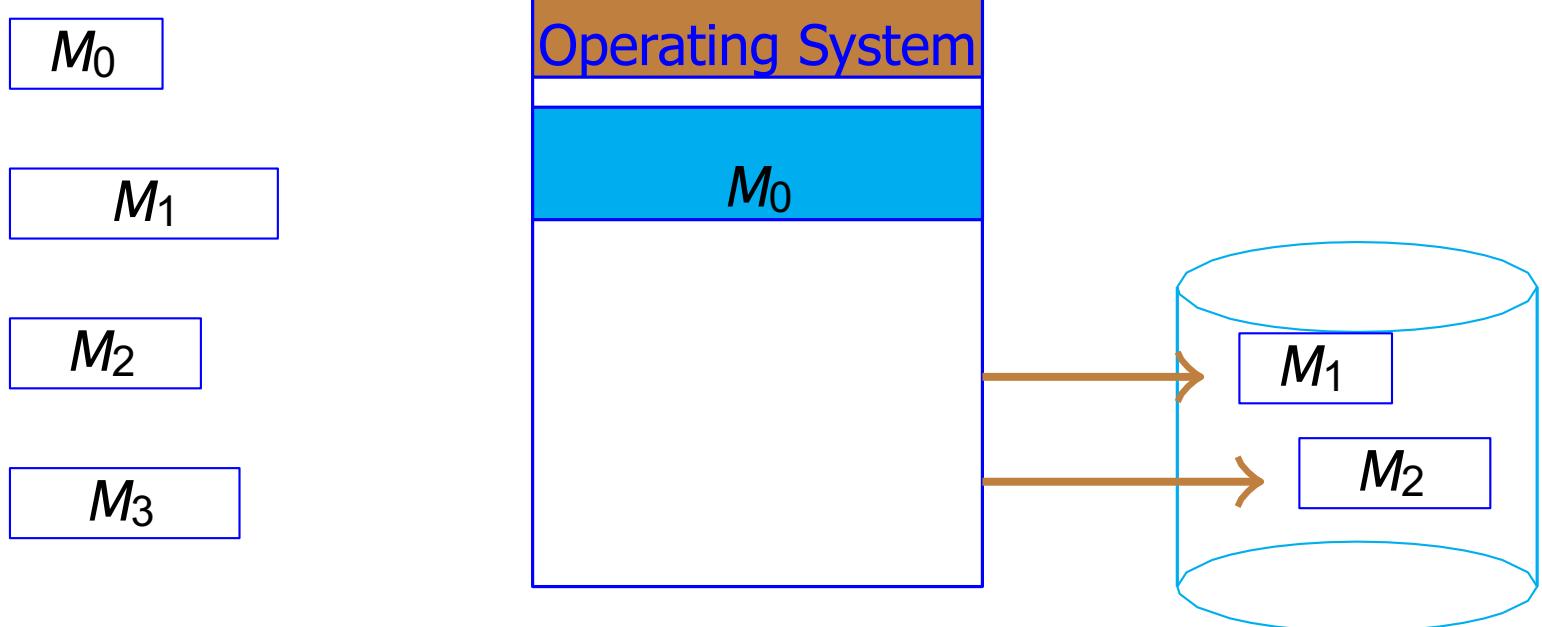
- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

Dynamic loading structure



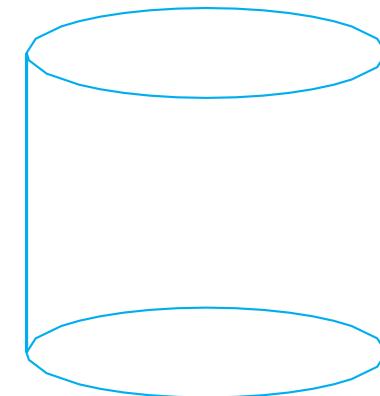
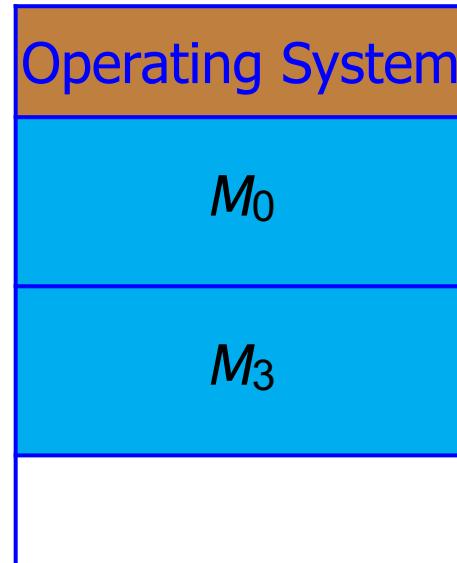
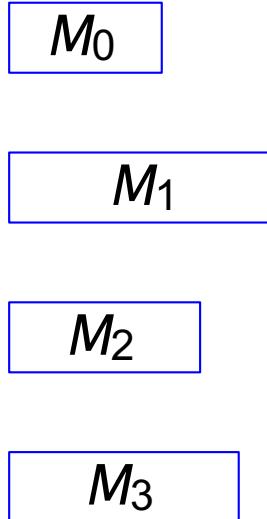
- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory

Dynamic loading structure



- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory
- When a module is finished using or not enough memory, bring unnecessary modules out

Dynamic loading structure



- Each module is edited separately
- When executing, system will load and localize the main module
- When module is needed, request for memory and load module into memory
- When a module is finished using or not enough memory, bring unnecessary modules out

Dynamic loading structure

Advantage

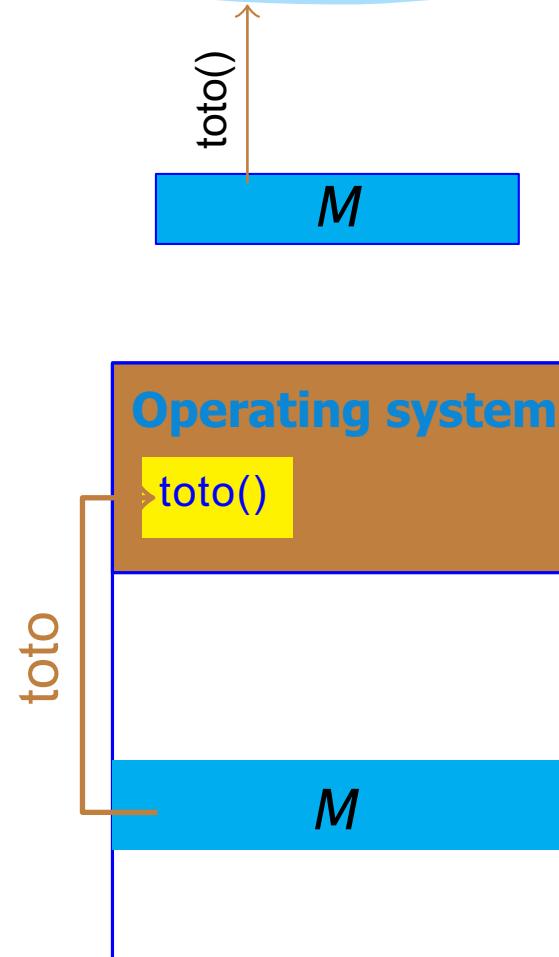
- Can use memory are smaller than the program's size
- High memory usage effectiveness if program is managed well

Disadvantage

- Slow when execution
- Mistake may cause waste of memory and increase execution time
- Require user to load and remove modules
 - User must understand clearly about the system
 - Reduce the program's flexible

Dynamic-link structure

- Links will be postponed when program is executing
- Part of the code segment (stub) is utilized to search for corresponding function in the library in the memory
- When found, stub will be replaced by the address of the function and function will be executed
- Useful for constructing library



Overlays structure

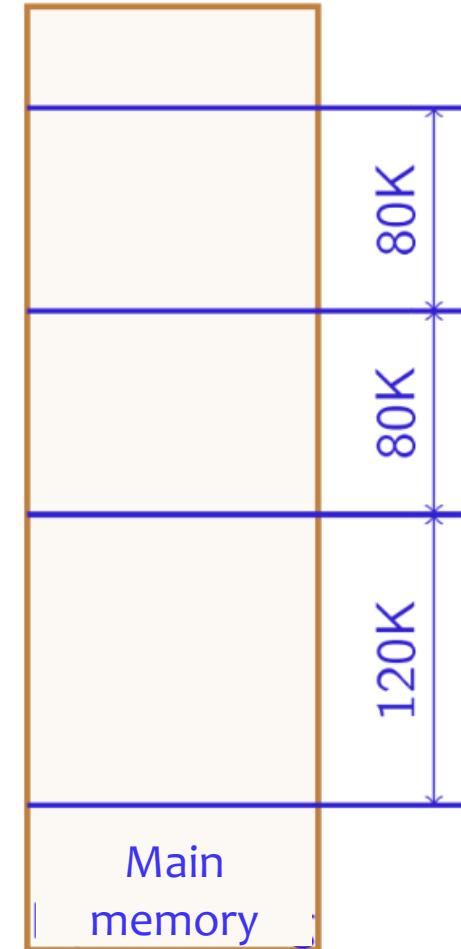
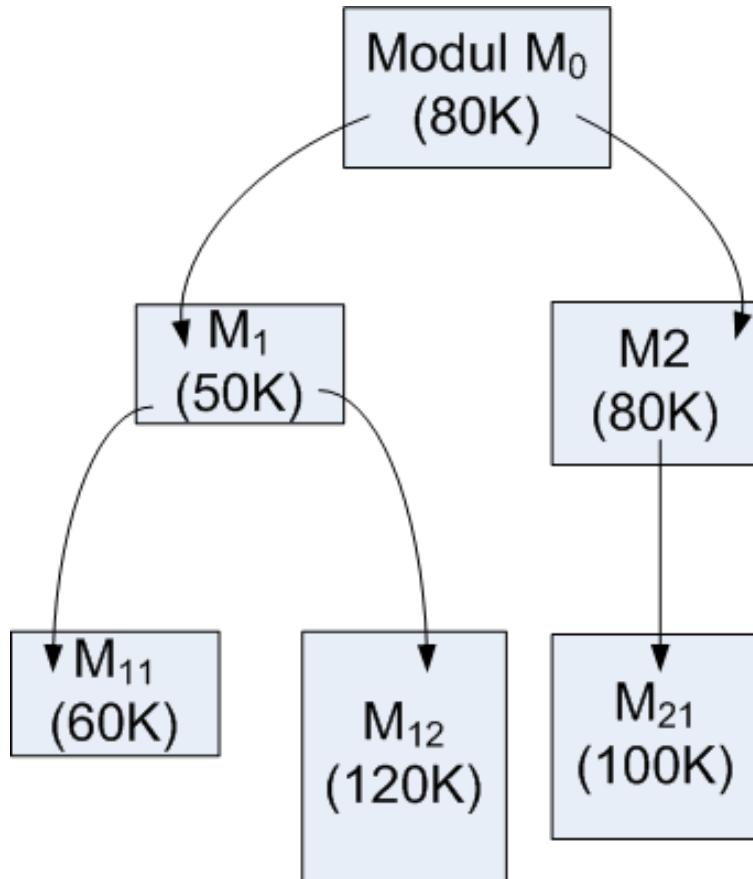
- Modules are divided into different levels
 - Level 0 contains main modul, load and localize the program
 - Level 1 contains modules called from level 0's module and these modules do not exist at the same time
 - ...
- Memory is also divided into levels corresponding to program's levels
 - Size equal to the same level's largest module's size
- Overlay structure requires extra information
 - Program is divided into how many levels, which modules are in each levels
 - Information is stored in a file (overlay map)
- Module at level 0 is edited into an independent executable file
- When program is executed
 - Load level 0 module similar to a linear structure program
 - When another module is needed, load that module into corresponding memory's level
 - If there are module in the same level exist, bring that module out

Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

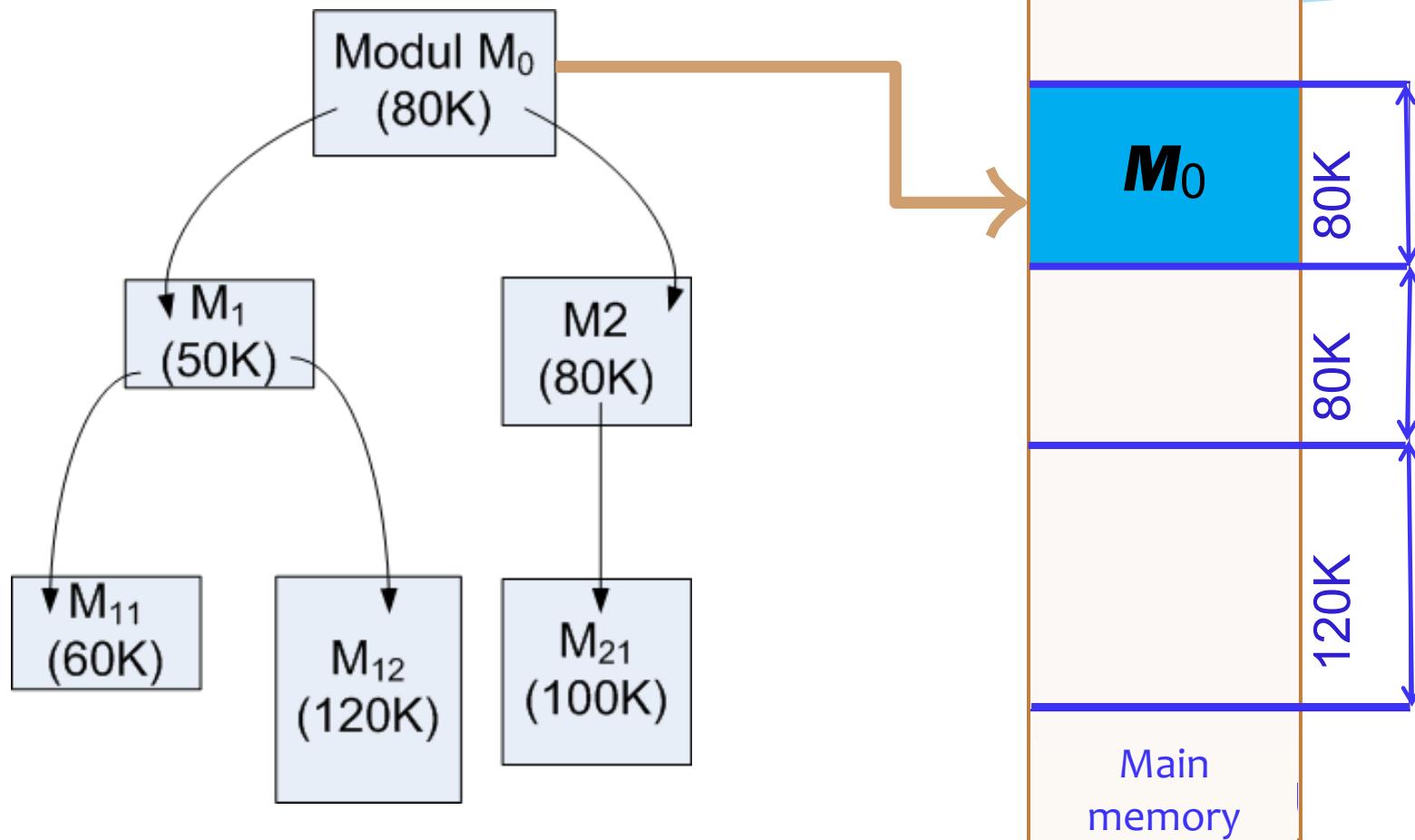


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

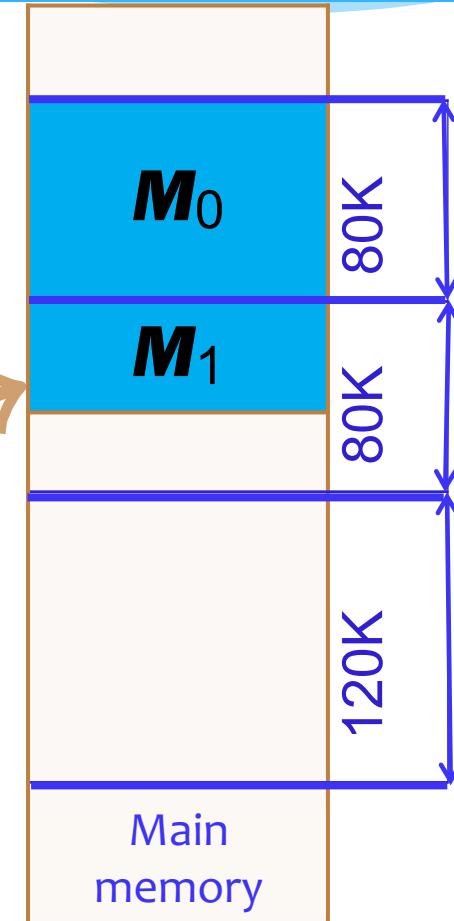
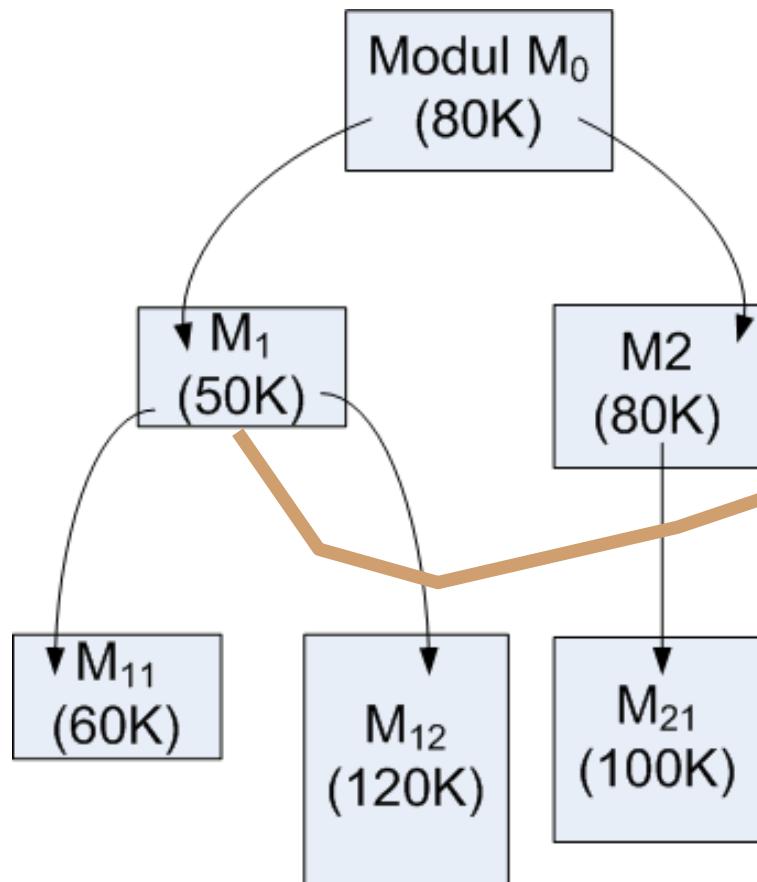


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

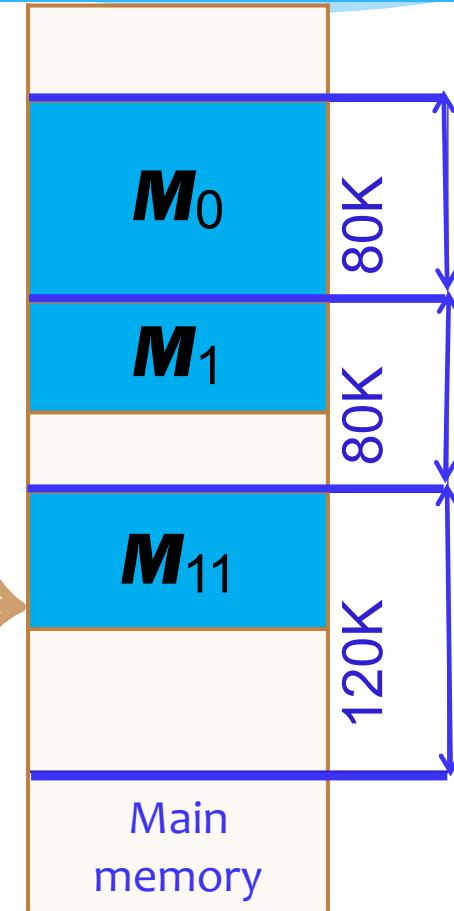
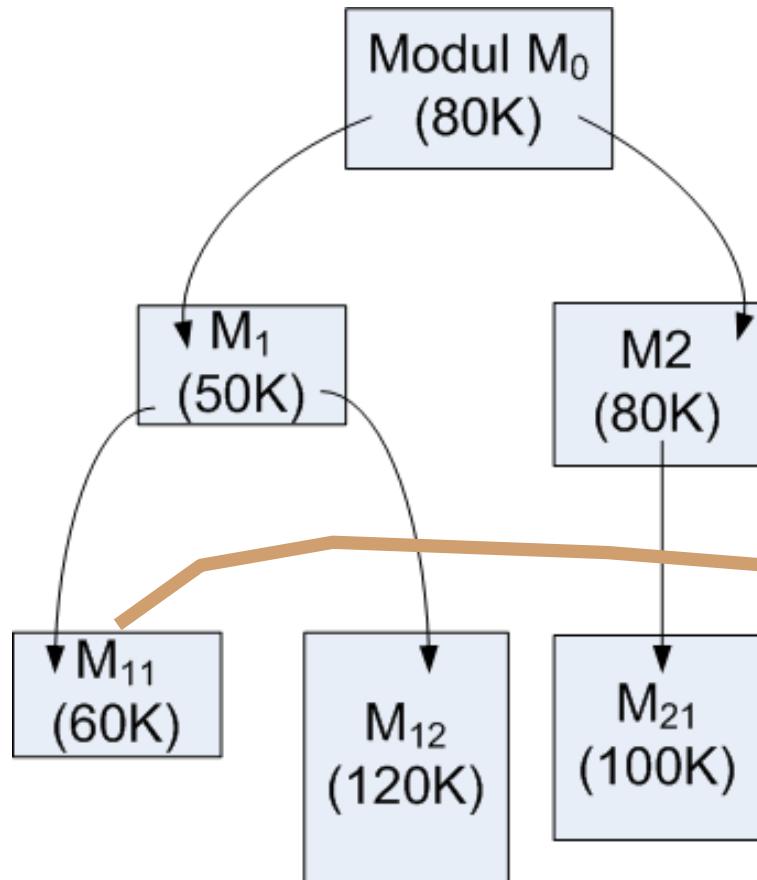


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

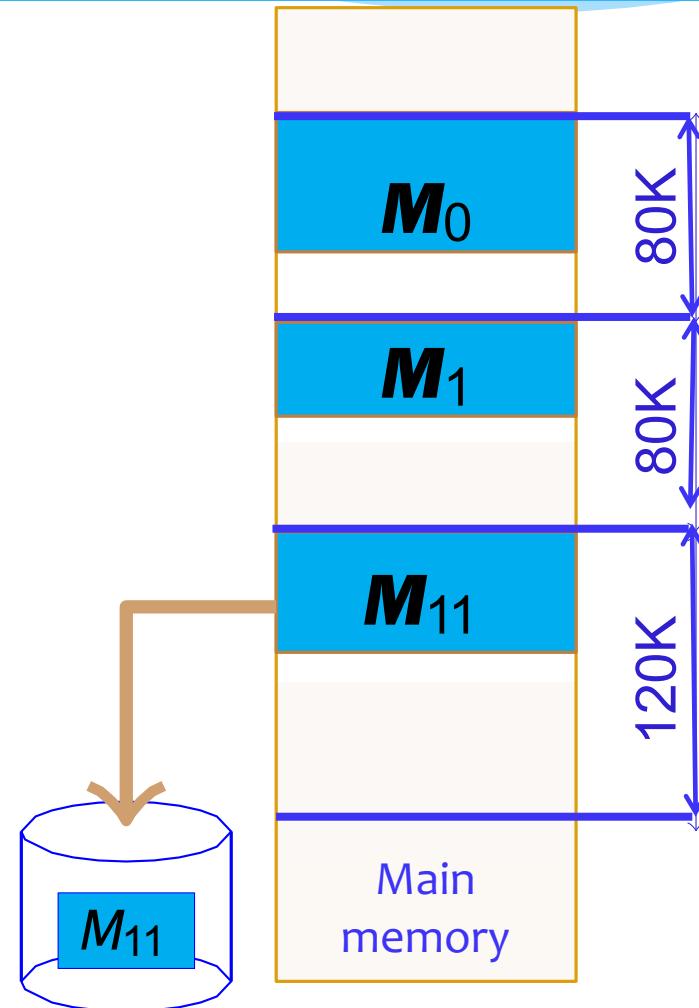
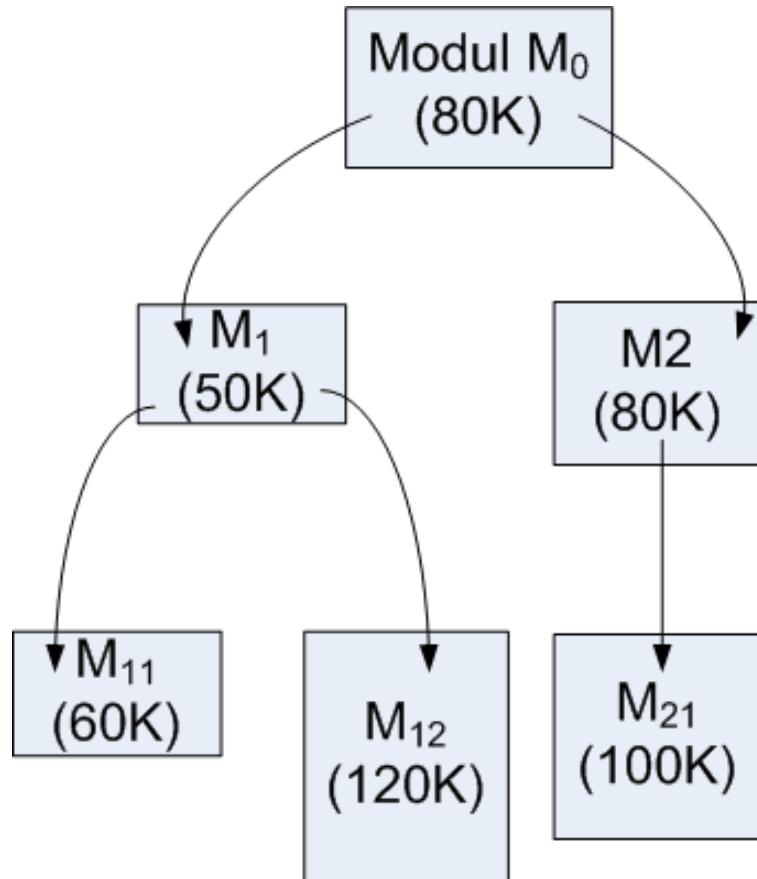


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

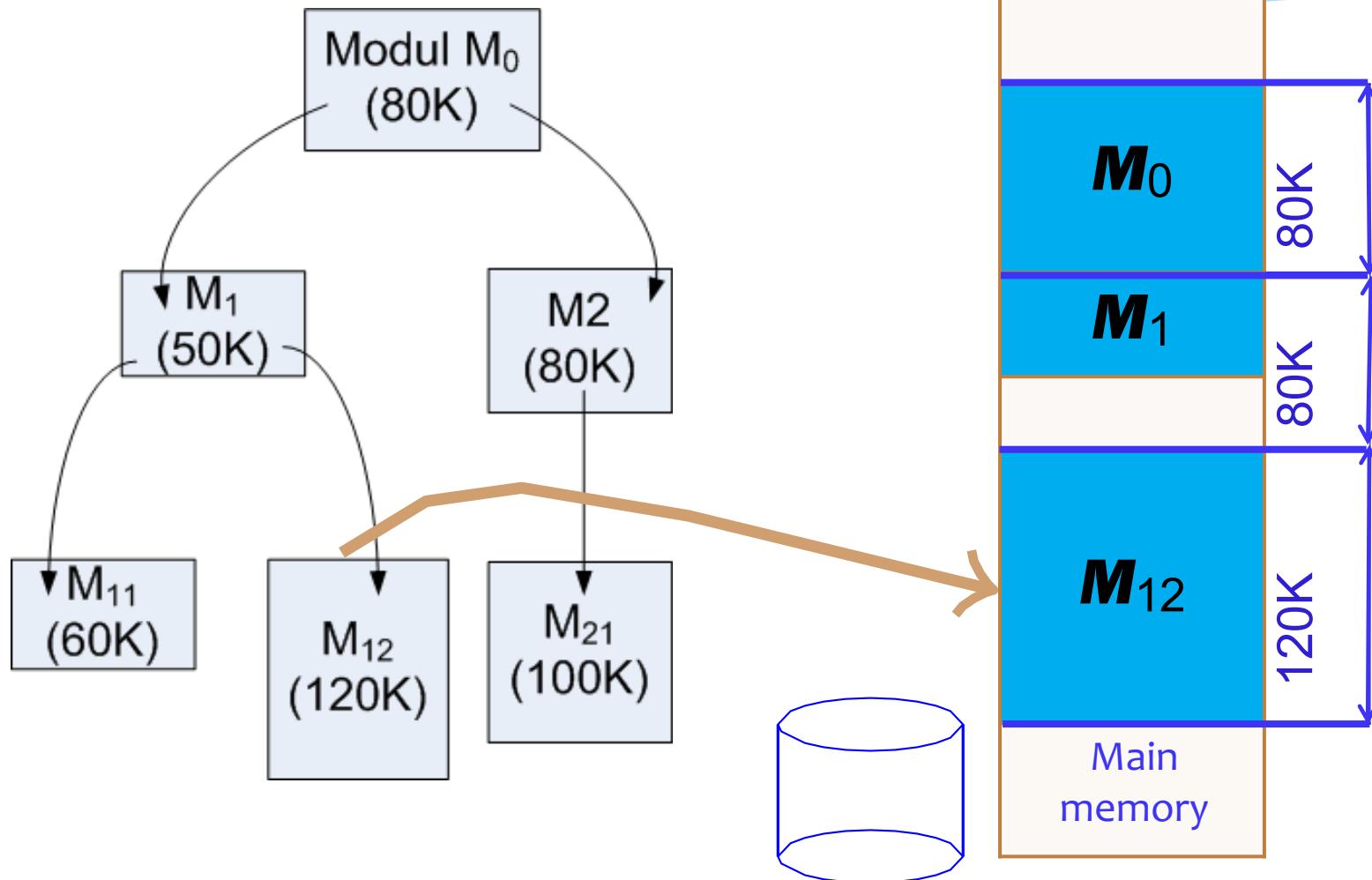


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

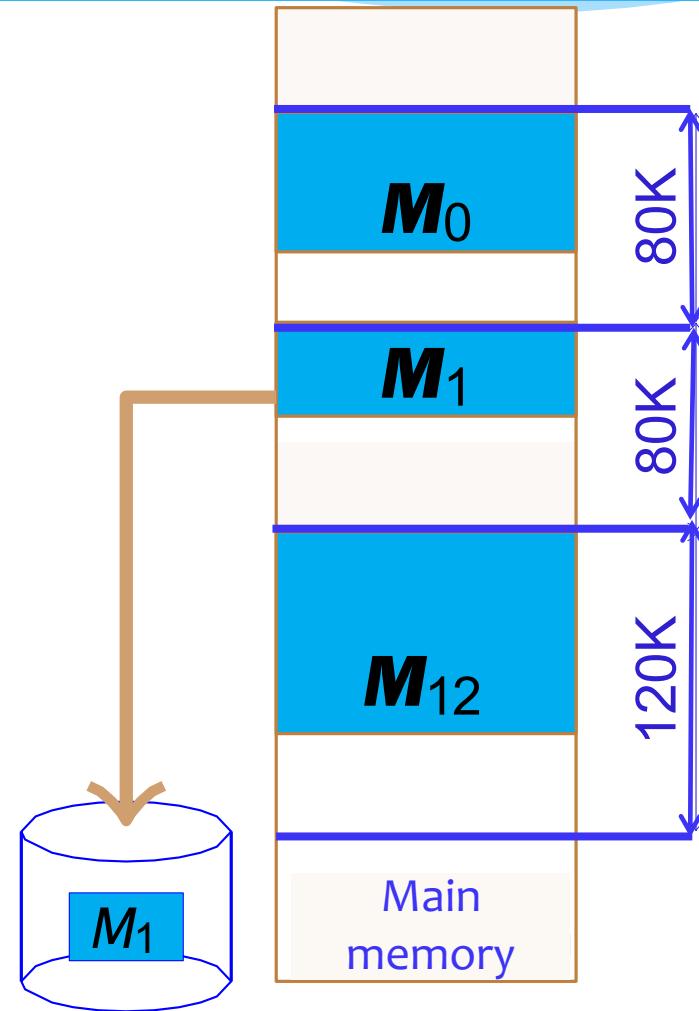
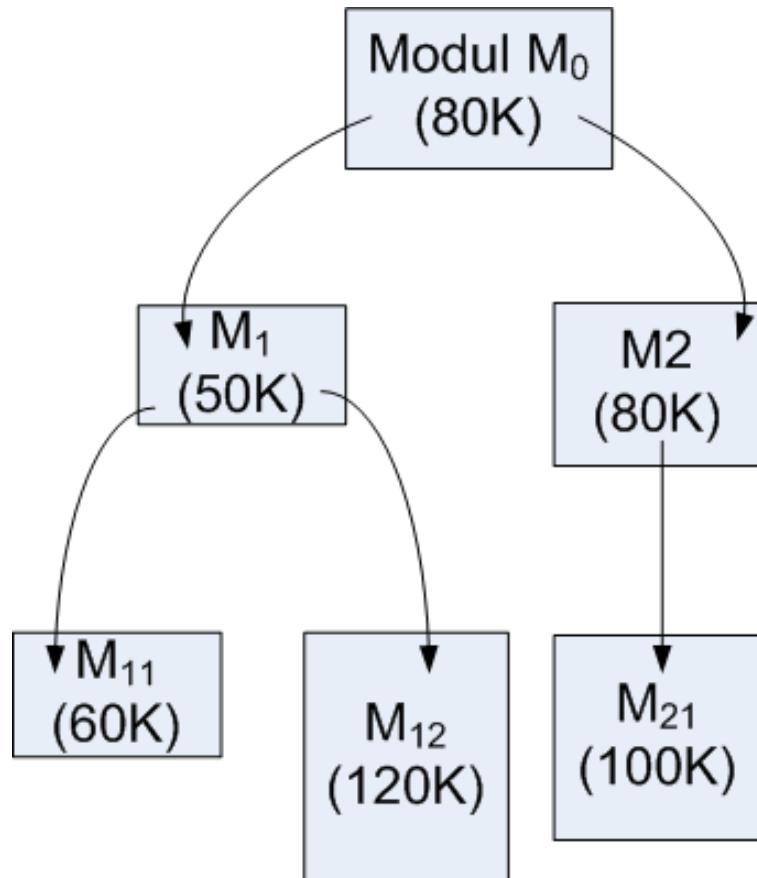


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure

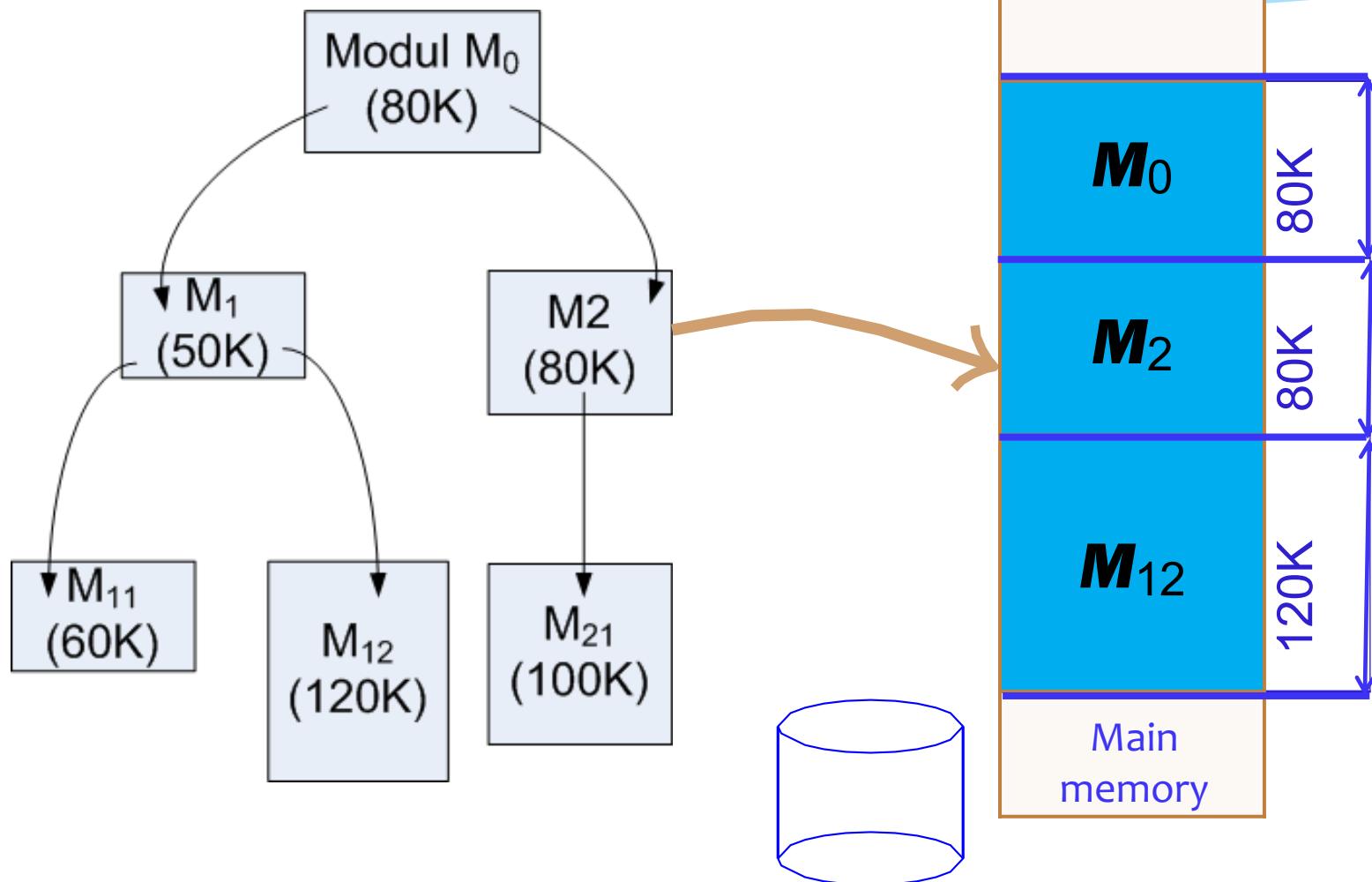


Chapter 3 Memory management

1. Introduction

1.4. Program's structures

Overlays structure



Overlays structure - Conclusion

- Allow program with larger size than memory area size allocated by the operating system
- Require extra information from user
 - Effectiveness is depend on provided information
- Memory usage effectiveness is depend on how program's modules are organized
 - If there are exist modules that larger than other modules in the same level ⇒ the effective is reduced
- Module loading process is dynamic but program's structure is static ⇒ not change at each time running
 - Provide more memory, the effectiveness does not increase

Chap 3 Memory Management

- ① Introduction
- ② **Memory management strategies**
- ③ Virtual memory
- ④ Memory management in Intel's processors family

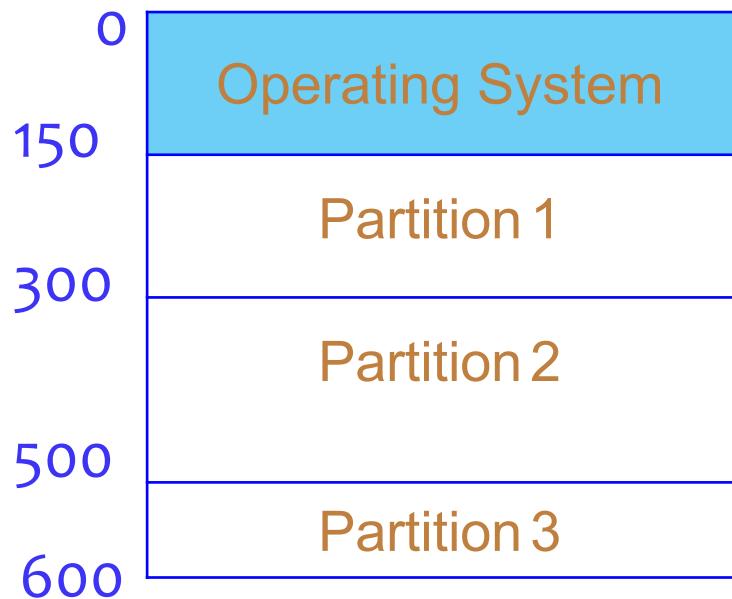
Chapter 3 Memory management

2. Memory management strategies

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

Rule

- Memory is divided into n parts
- Each part is called a *partition*
 - size: can be unequal
 - Utilized as an independent memory area
 - At a single time, only one program is allowed to exist
 - Programs lie inside memory until finish
- Example: Consider the following system



Process	Size	time
P_1	120	20
P_2	80	15
P_3	70	5
P_4	50	5
P_5	140	12
Queue		

Conclude

- Simple, easy for memory protection
 - Program and memory area have a protection lock
 - Compare 2 locks when program is loaded
- Reduce searching time
- Must copy controlling module into many versions and save at many places
- Parallel cannot be more than n
- Memory is segmented
 - Program's size is larger than the largest partition's size
 - Total free memory is large enough but can not load any program
⇒ Fix partition structure, merge neighboring partition
- Application
 - Large size disk management
 - IBM OS/360 operating system

Chapter 3 Memory management

2. Memory management strategies

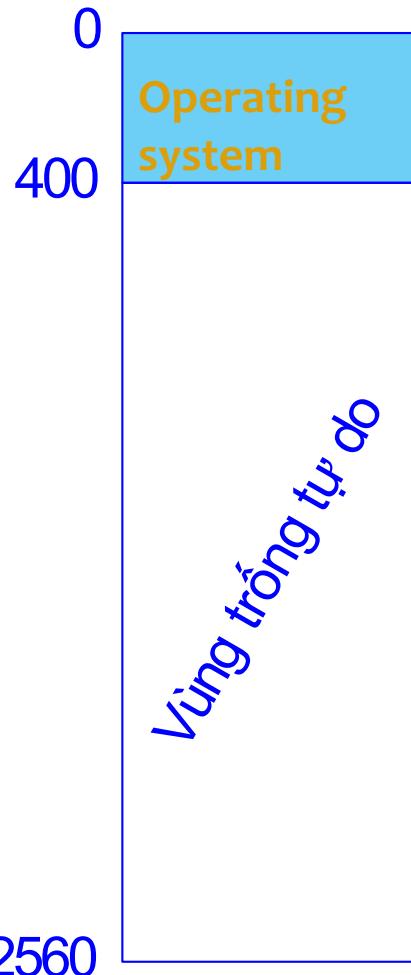
- Fixed partition strategy
- **Dynamic partition strategy**
- Segmentation strategy
- Paging strategy
- Segmentation and paging combination strategy

Rule

Only one management list for free memory

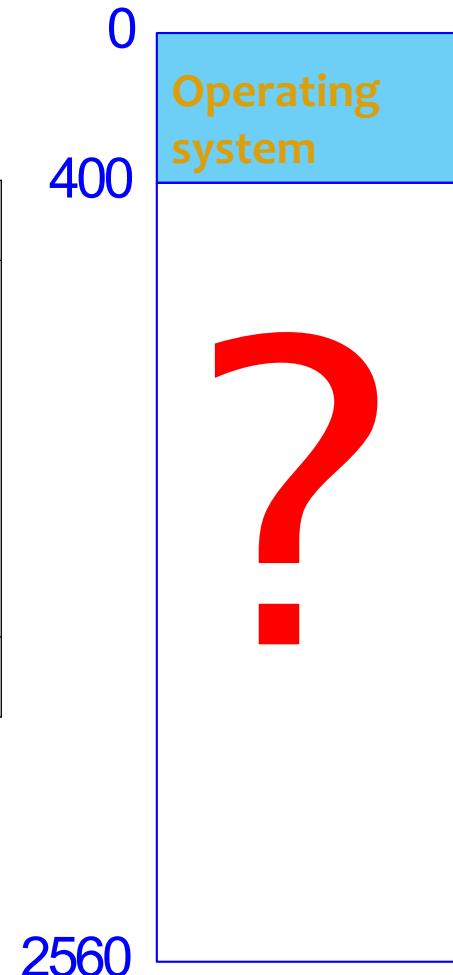
- At the start, the whole memory is free for processes ⇒ largest *hole*
- When a process requests for memory
 - Search in the list for a large enough hole for request
 - If found
 - Hole is divided into 2 parts
 - One part allocate to process as requested
 - One part return to the management list
 - If not found
 - Wait until there is a hole large enough
 - Allow another process in the queue to execution (if the priority is guaranteed)
- When the process finish
 - Allocated memory area is returned to the free memory management list
 - Combine with other neighboring holes if necessary

Main memory



Process	Size	time
P_1	600	10
P_2	1000	5
P_3	300	20
P_4	700	8
P_5	500	15

Waiting file queue



Strategies to select free area for process's request

First Fit : First free area satisfy request

Best Fit : Most fitted area

Worst Fit : Largest area that satisfy request

Free memory area selection strategy

- Suppose free memory area have the size 100K, 500K, 200K, 300K, and 600K (consequently),
- First-fit, Best-fit, and Worst-fit
How will process with size 212K, 417K, 112K, and 426K loaded?

Memory reallocation problem

After a long working time, the free holes are distributed and caused memory lacking phenomenon ⇒ Need to rearrange memory

- Move process
 - Problem: internal objects when move to new place will have new address
 - Use relocation register to store process's relocation value
 - Select method for lowest cost
 - Move all process to one side ⇒ largest free holes
 - Move processes to create a sufficient free hole immediately
- Swapping process
 - Select a right time to suspend process
 - Bring process and corresponding state to external memory
 - Free allocated memory area and combine with neighboring areas
 - Reallocation to former place and restore state
 - Use relocation register if process is moved to different places

Concludes

- No need to copy the controlling modules to different places
- Increase/decrease parallel factor depend on the number and size of programs
- Cannot run program with size larger than the physical memory size
- Cause memory waste phenomenon
 - Memory area is not used and not in the memory management list
 - Cause by the operating system error
 - By malicious software
- Cause external memory defragment phenomenon
 - Free memory area is managed but distributed -> cannot used
- Cause internal memory defragment phenomenon
 - Memory allocated to process but not used by process

Chapter 3 Memory management

2. Memory management strategies

- Fixed partition strategy
- Dynamic partition strategy
- **Segmentation strategy**
- Paging strategy
- Segmentation and paging combination strategy

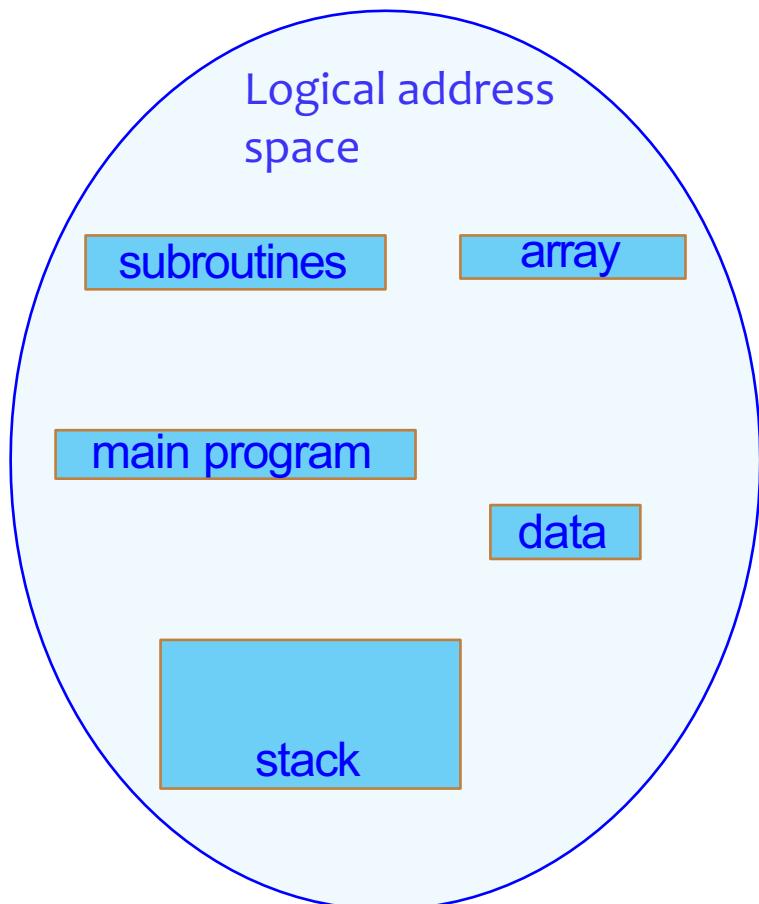
Program

- In general, a program includes following modules
 - main program
 - Set of sub-routine
 - Variables, data structures, . . .
- Modules, objects in program are defined by name
 - function sqrt(), procedure printf() . . .
 - x, y, counter, Buffer . . .
- Member inside a module is determined based on the distance from the head of the module
 - Instruction 10th of function sqrt() . . .
 - Second element of array Buffer . . .

How program is placed inside memory?

- Stack stay at the higher area or Data stay at the higher area?
 - Object's physical address . . . ?
- ⇒ Users donot care

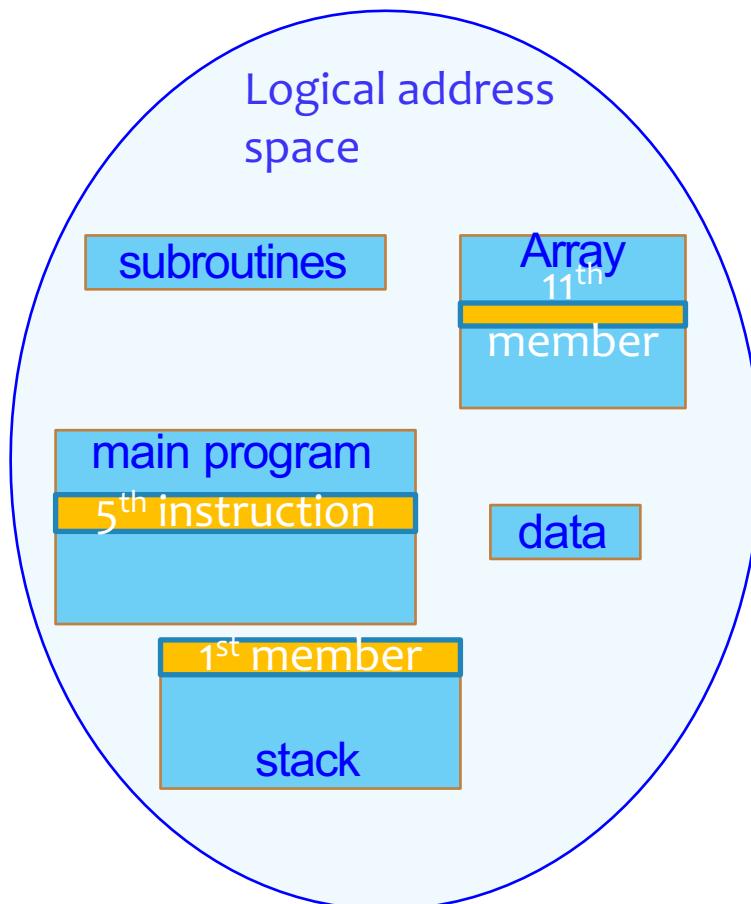
User's perspective



When program is loaded into memory to execute. Program is combined of several segments

- Each segment is a logic block, corresponding to a module
 - **Code**: main(), procedure, function. . .
 - **Data**: Global objects
 - Other segments: **stack, array**. . .
- Each segment occupy an **contiguous memory area**
 - Has a **start position and size**
 - Can be located at **any place** in the memory

User's perspective



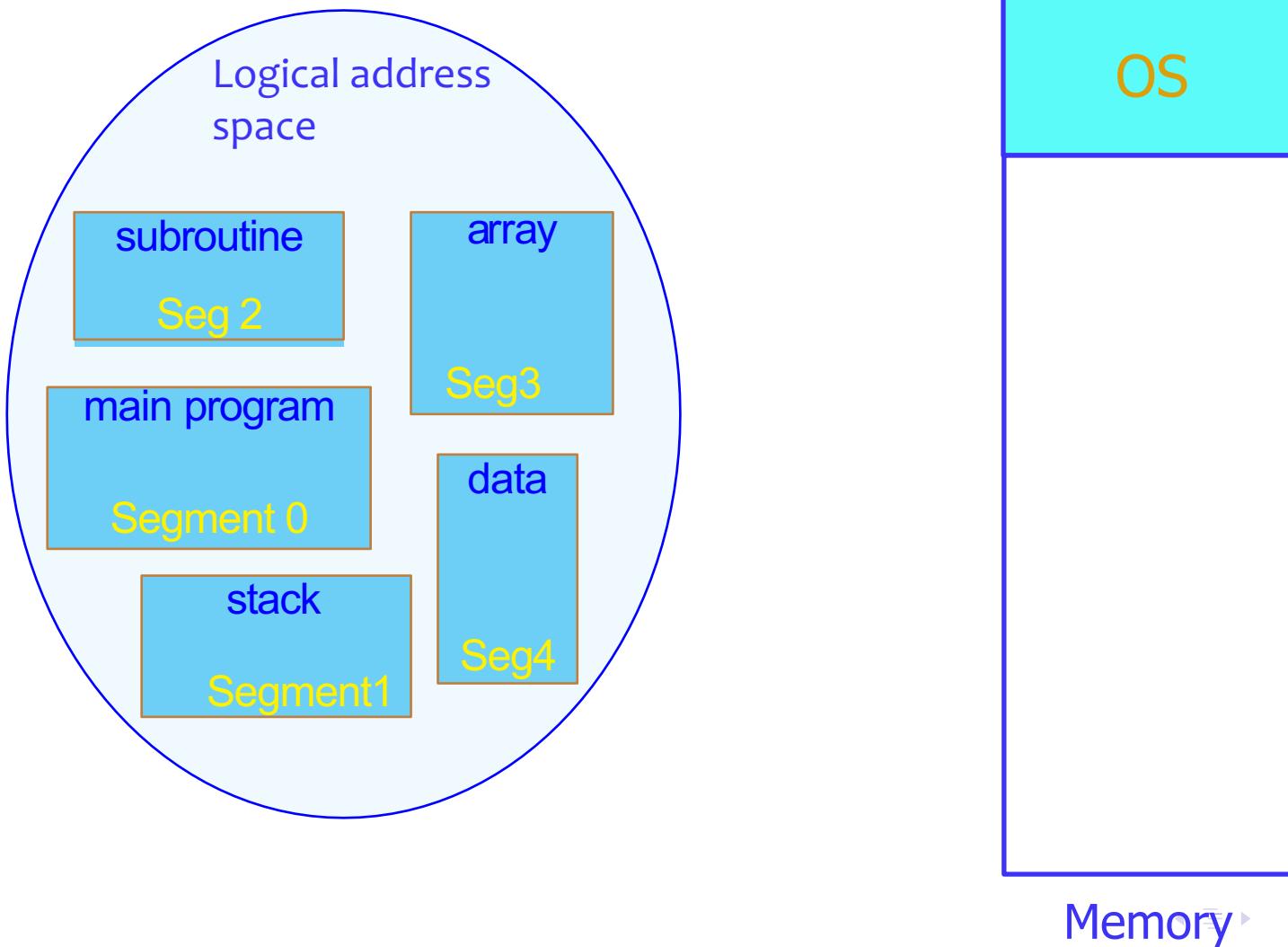
- Object in a segment is defined based on the relative distance from the start of the segment
 - 5th instruction of the main program
 - 1st member of the stack. . .
- Where is the location of these objects in the memory?

Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

Example

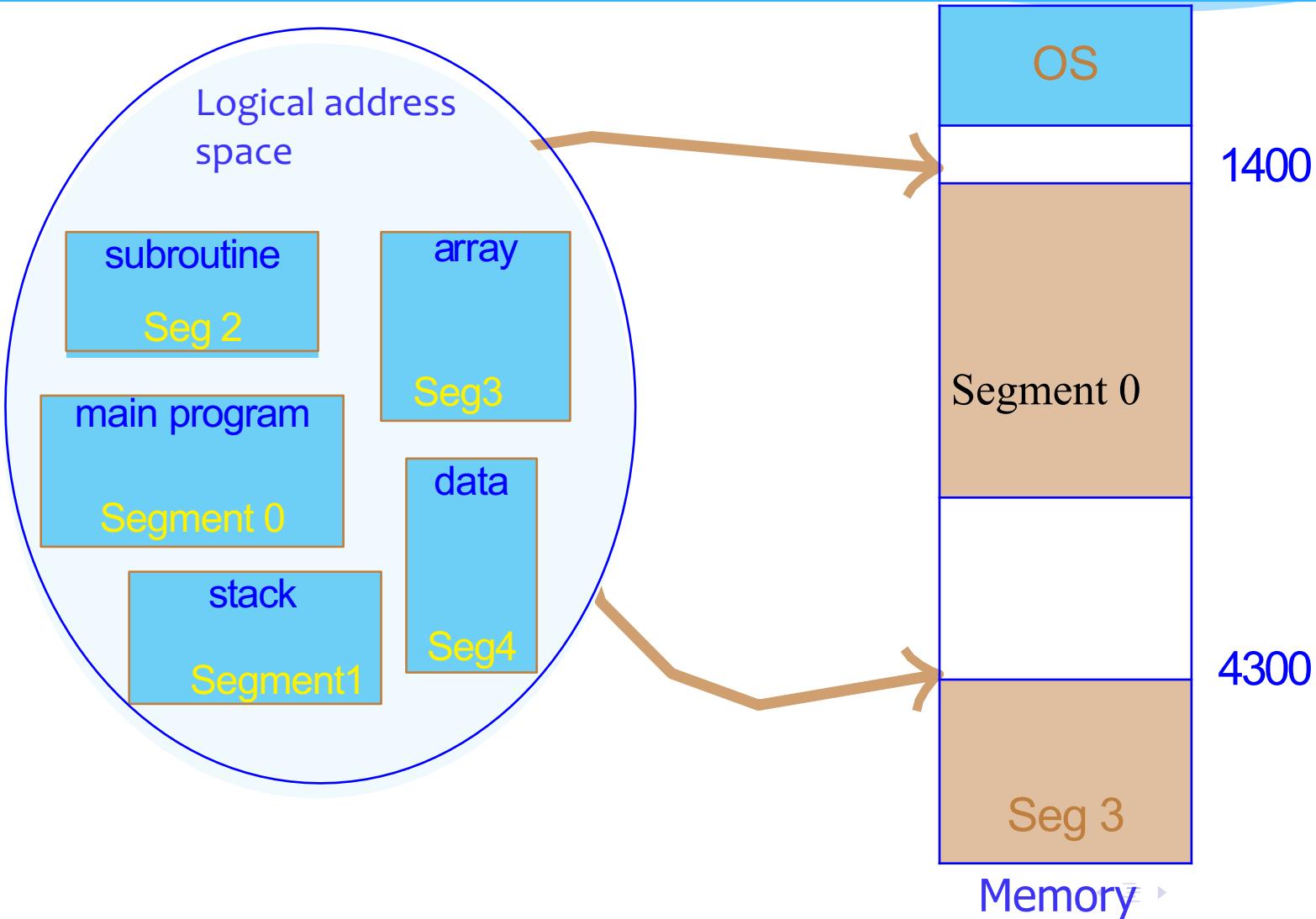


Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

Example

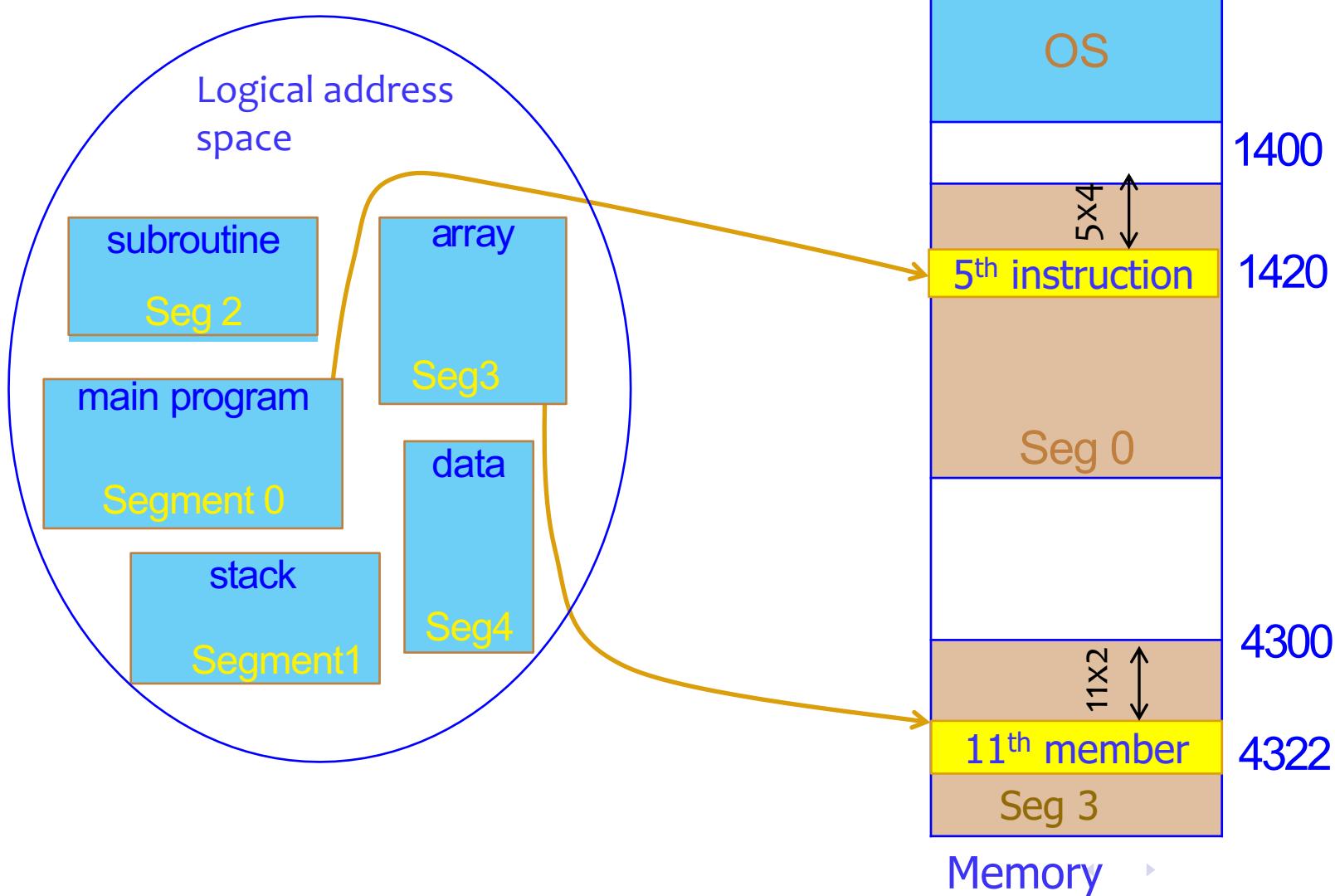


Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

Example



Segmentation structure

- Program is a combination of modul/segment
 - Segment number, segment's length
 - Each segment can be edited independently.
- Compile and edit program -> create SCB (Segement Control Block)
- Each member of SCB is corresponding to a program's segment

	Mark	Address	Length
0			
...			
n	⋮⋮⋮	⋮⋮⋮	⋮⋮⋮

- Mark(0/1) : Corresponding segment is already inside memory
- Address: Segment's base location in memory
- Length: Segment's length
- Accessing address: segment's name (number) and offset

Problem: Convert from 2 dimension address to 1 dimension address

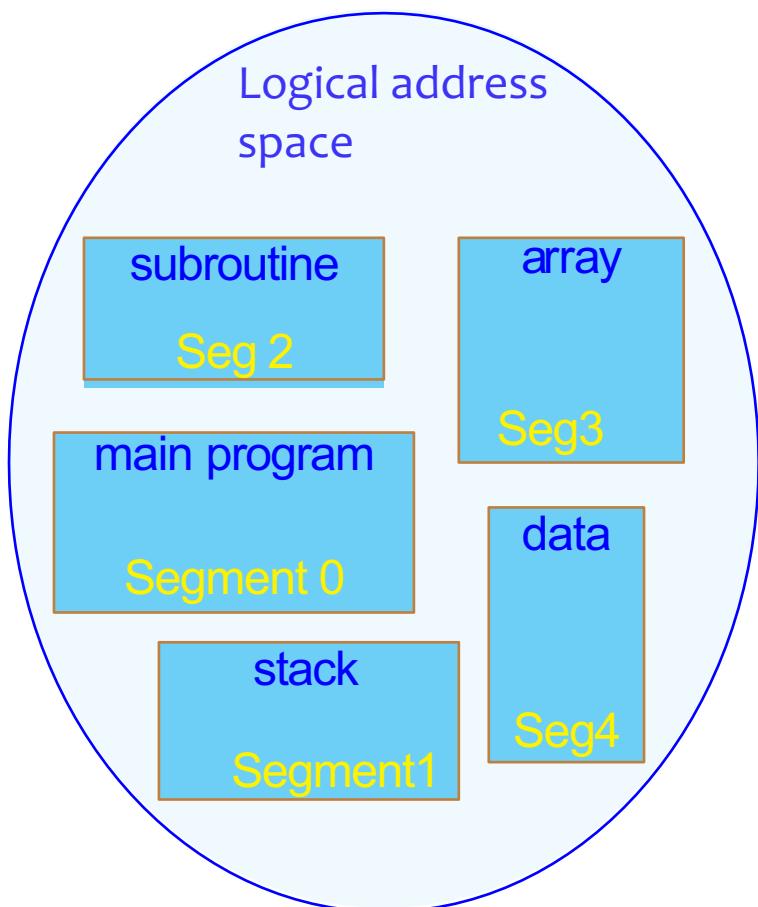


Chapter 3 Memory management

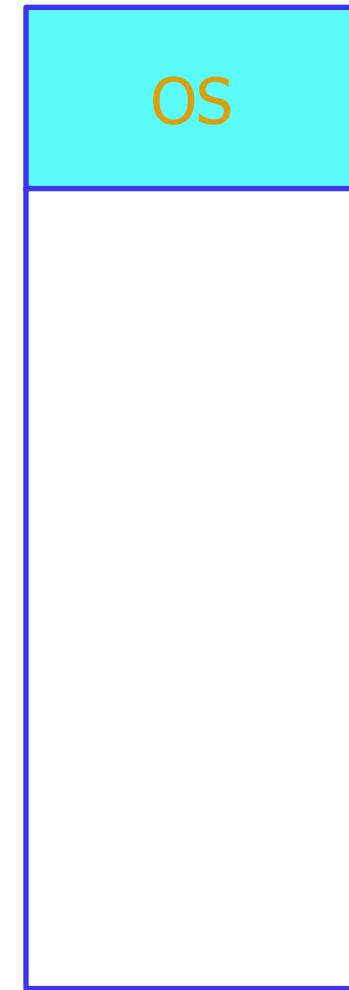
2. Memory management strategies

2.3 Segmentation strategy

Example



M	A	L
0	-	1000
0	-	400
0	-	400
0	-	1100
0	-	1000
SCB		



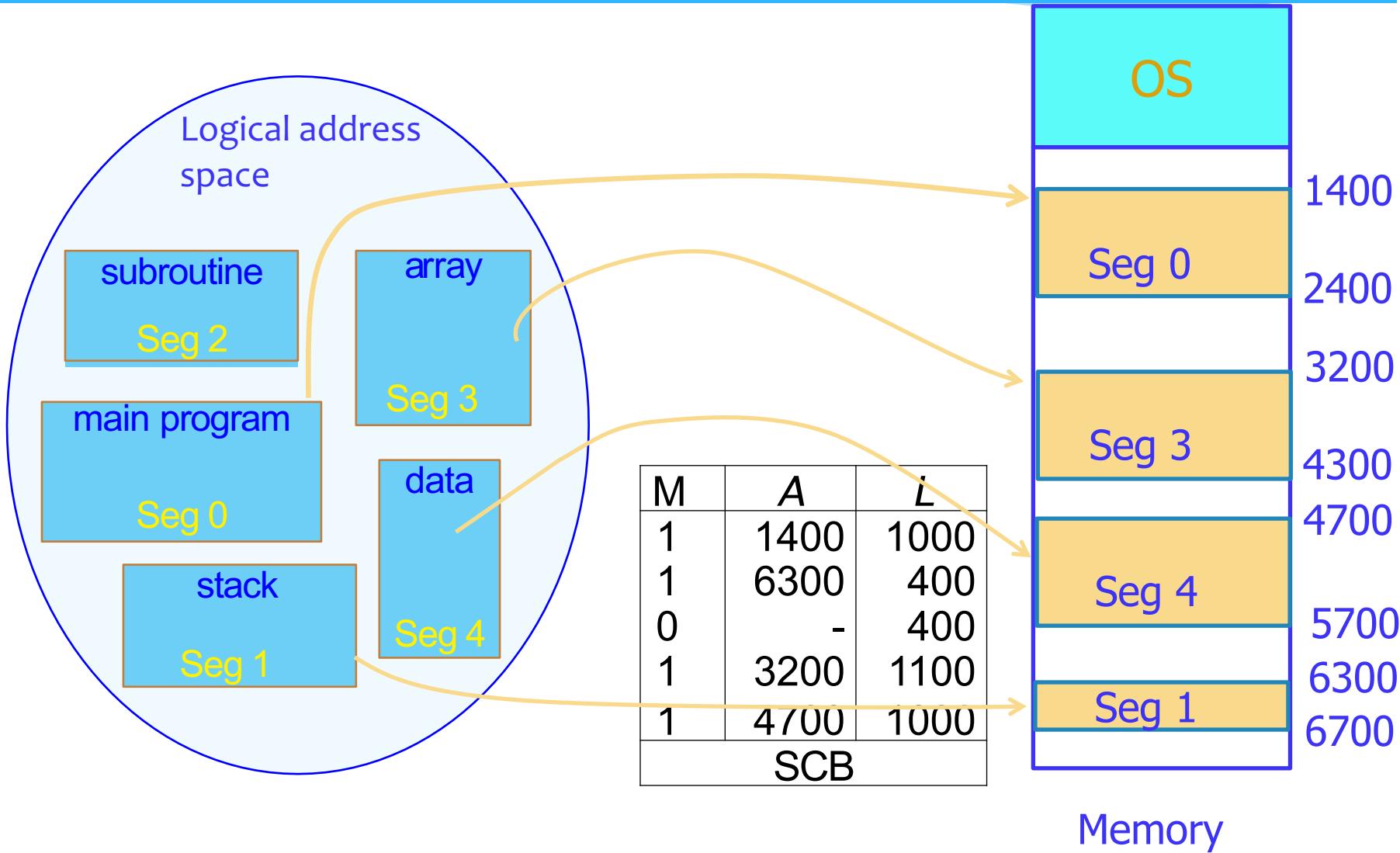
Memory

Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

Example

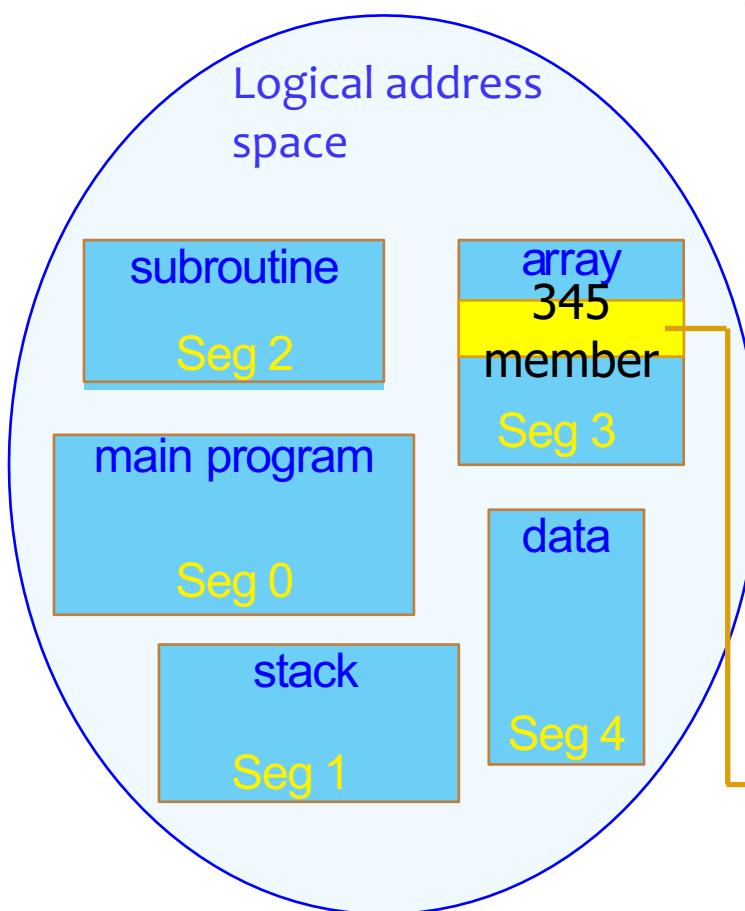


Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

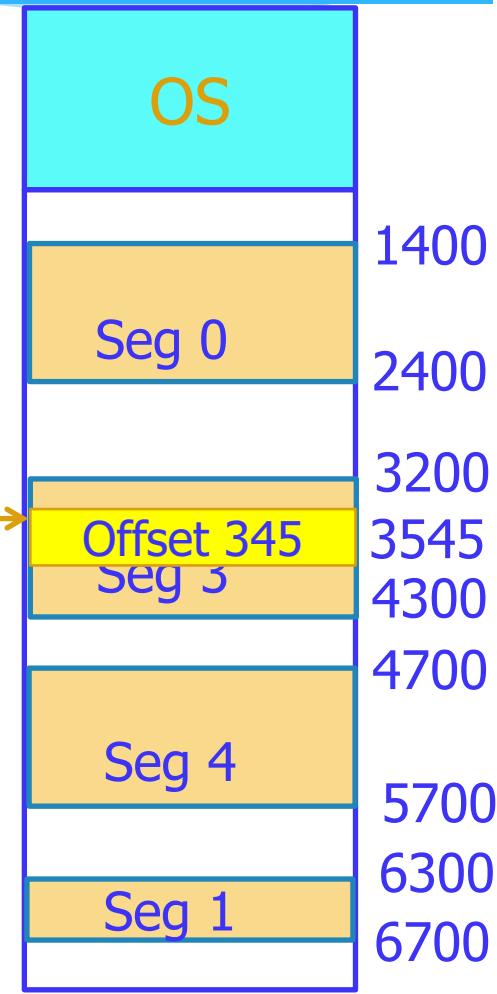
Example



Address <3,345> = ?

M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000

SCB



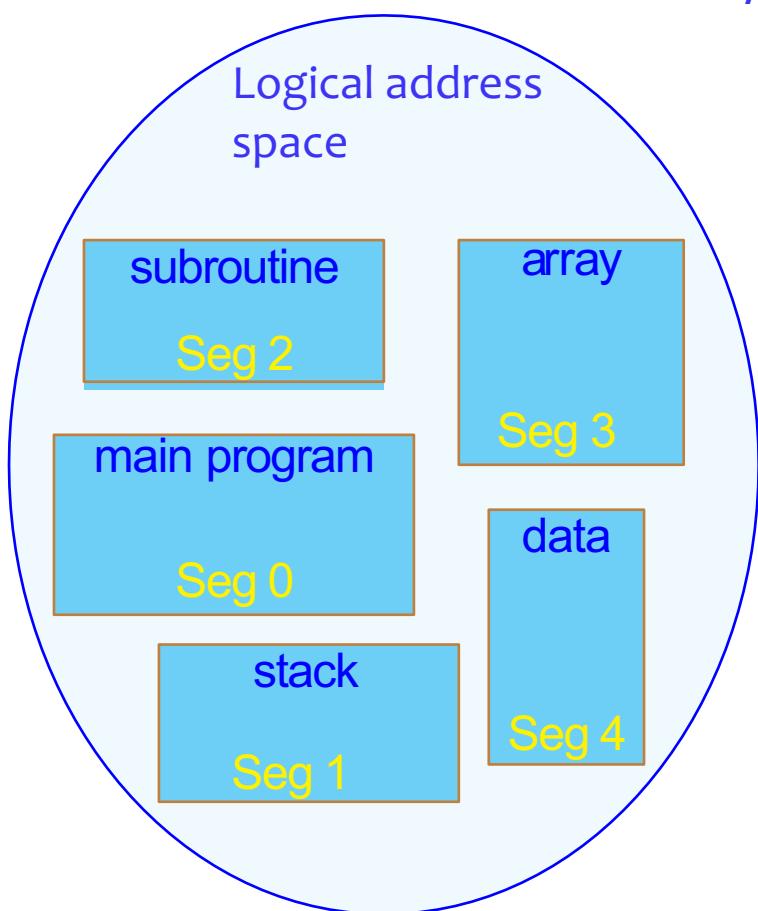
Memory

Chapter 3 Memory management

2. Memory management strategies

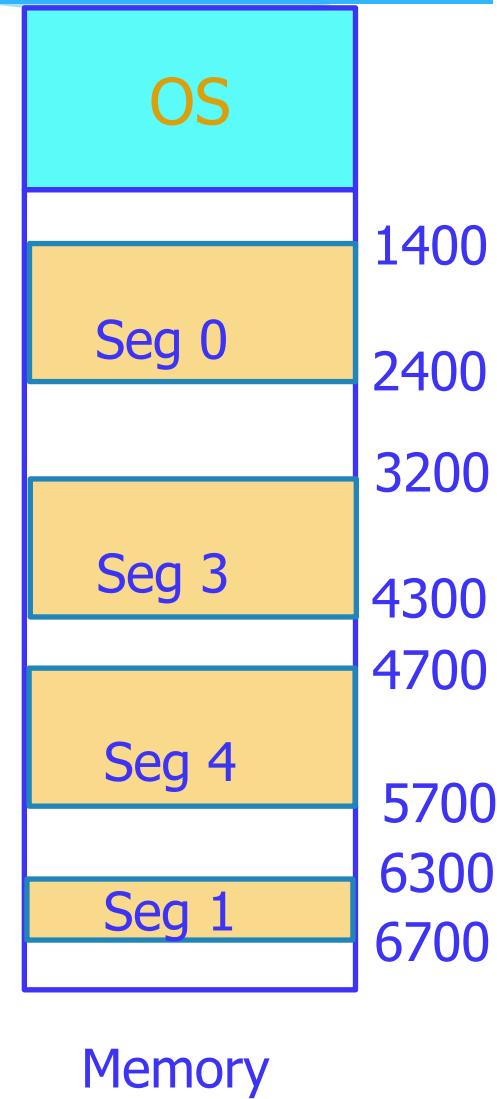
2.3 Segmentation strategy

Example



Address <4,185> = 4885

M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

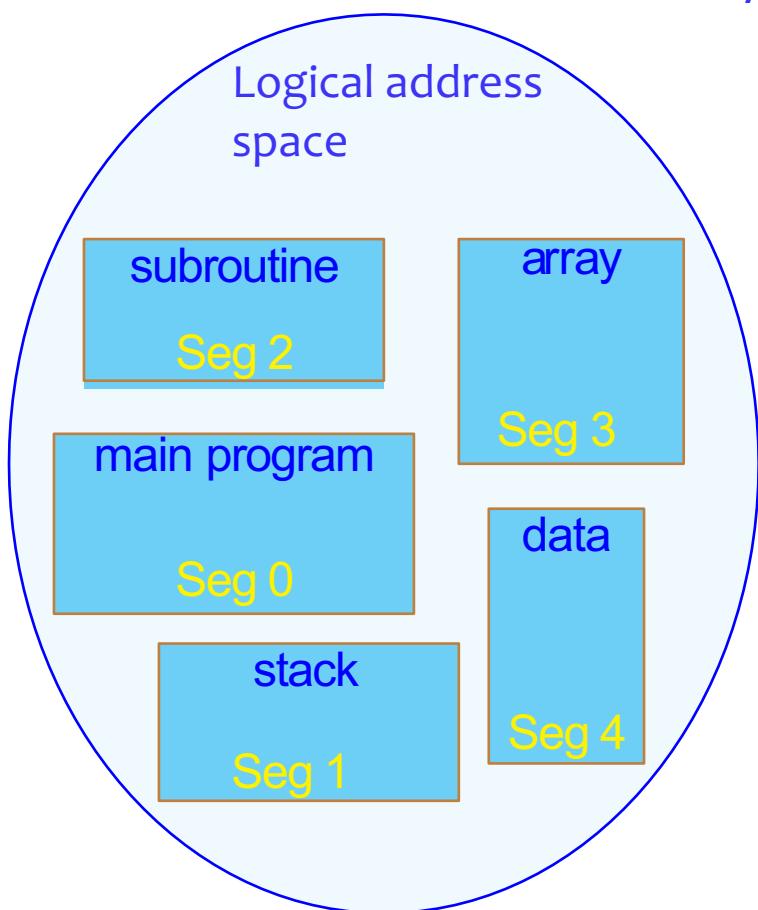


Chapter 3 Memory management

2. Memory management strategies

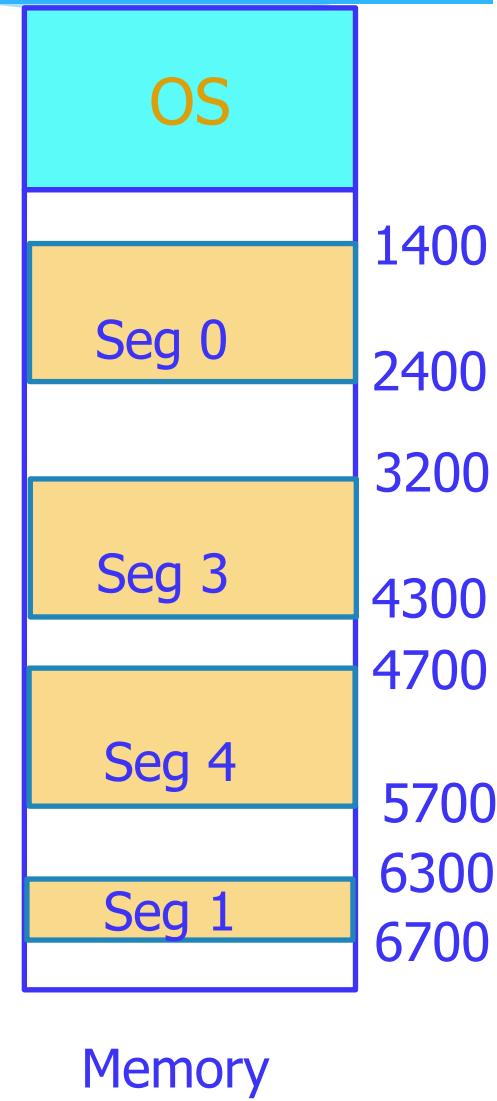
2.3 Segmentation strategy

Example



Address <2,120> = ?

M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

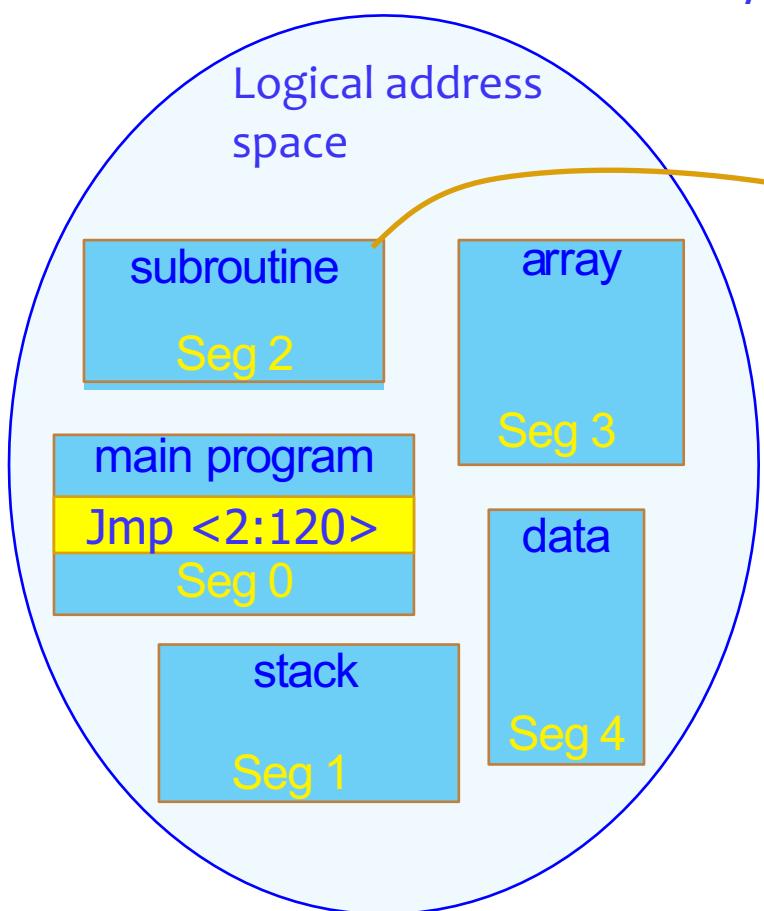


Chapter 3 Memory management

2. Memory management strategies

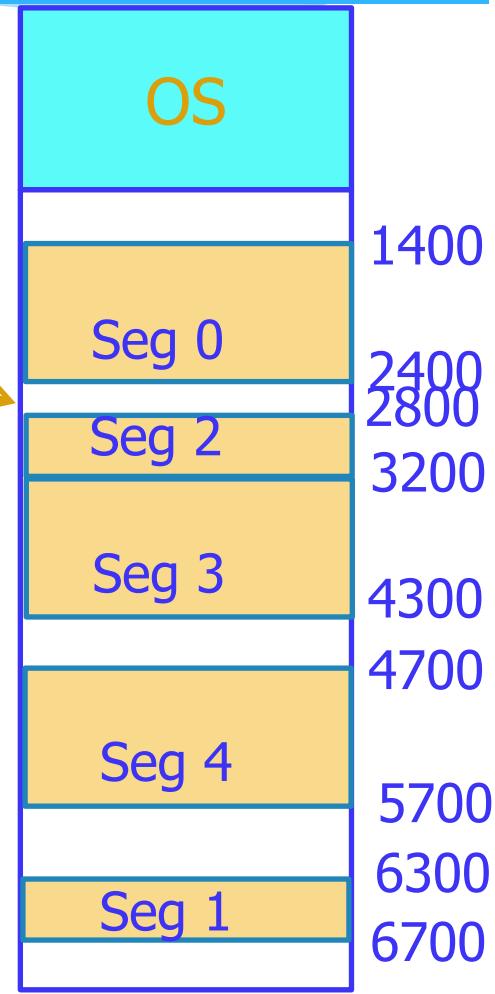
2.3 Segmentation strategy

Example



Address <2,120> = ?

M	A	L
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		



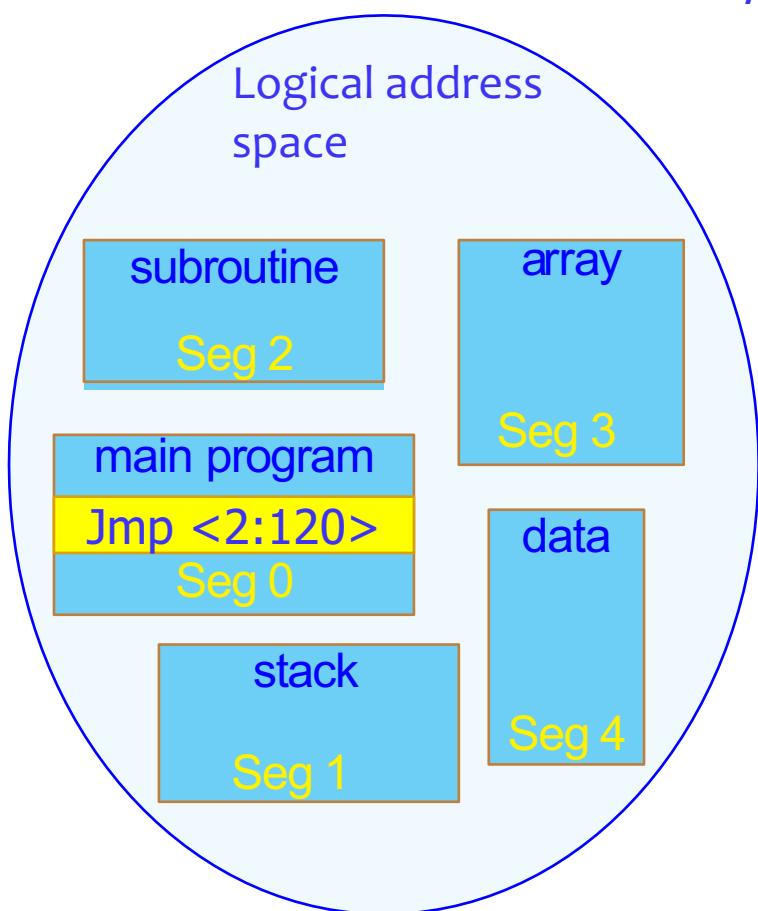
Memory

Chapter 3 Memory management

2. Memory management strategies

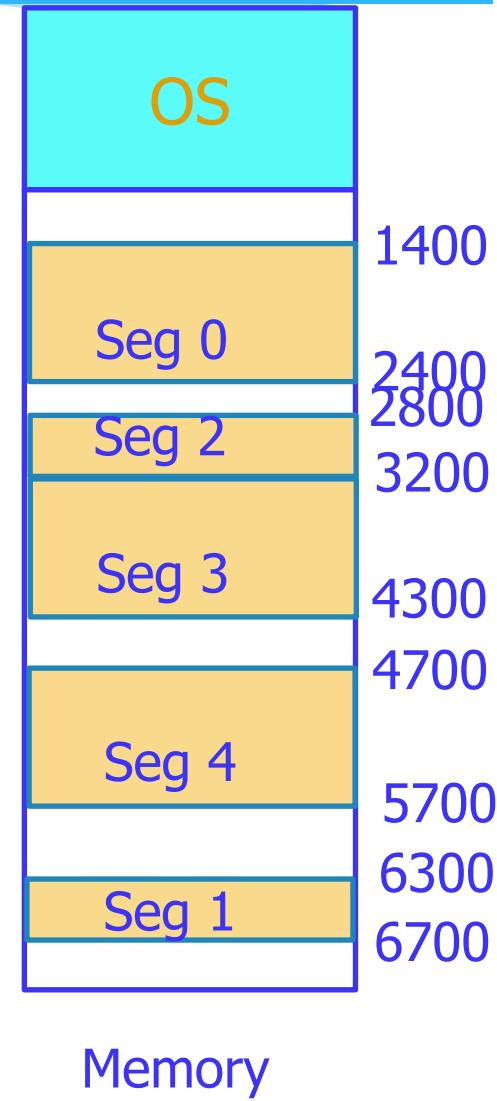
2.3 Segmentation strategy

Example



Address <2,120> = ?

M	A	L
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		

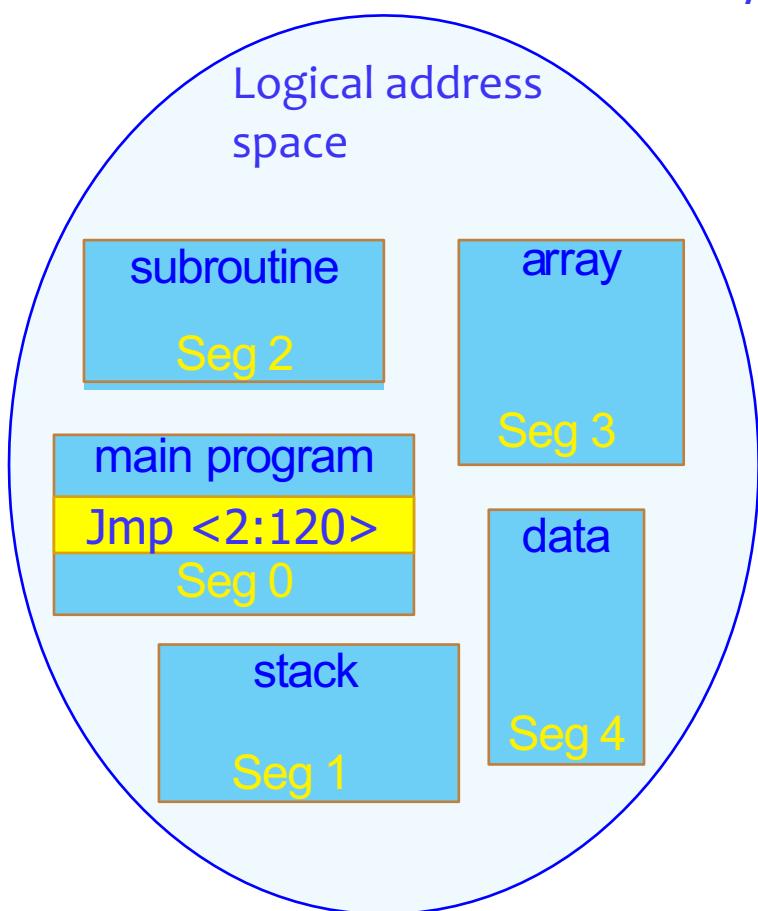


Chapter 3 Memory management

2. Memory management strategies

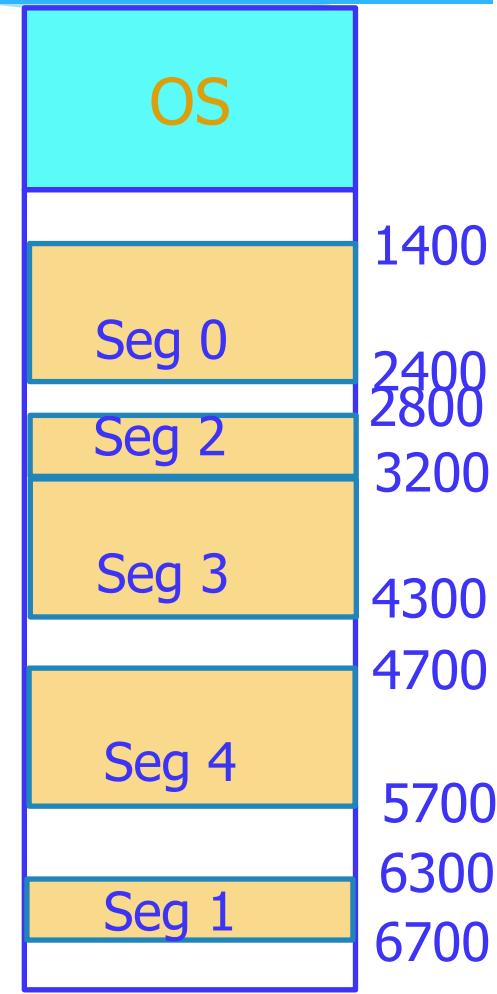
2.3 Segmentation strategy

Example



Address <2,120> = 2920

M	A	L
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		



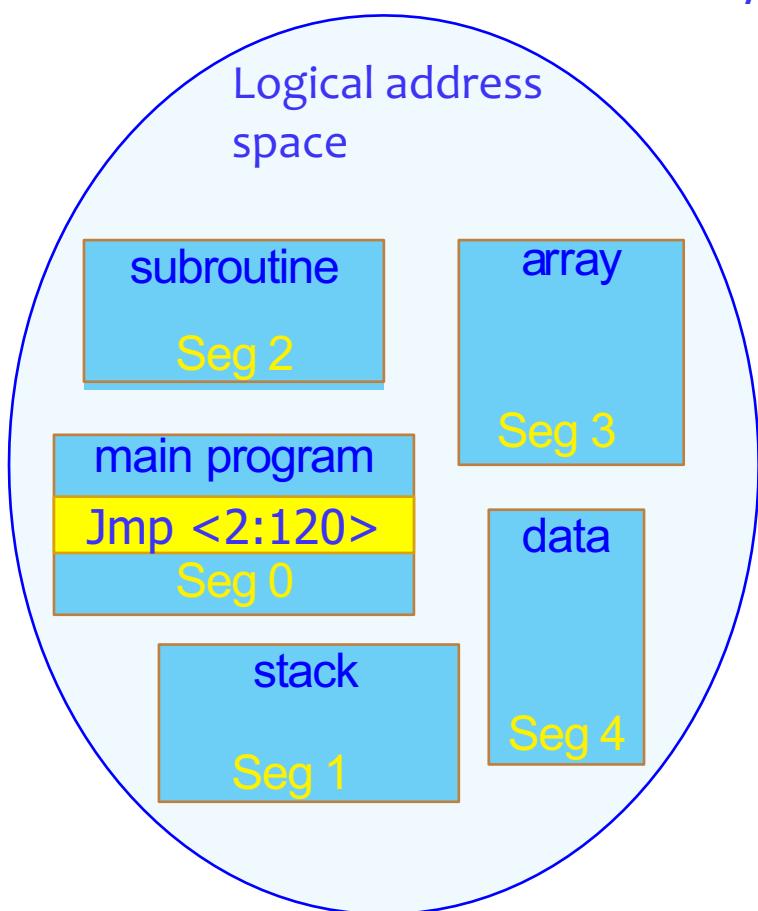
Memory

Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

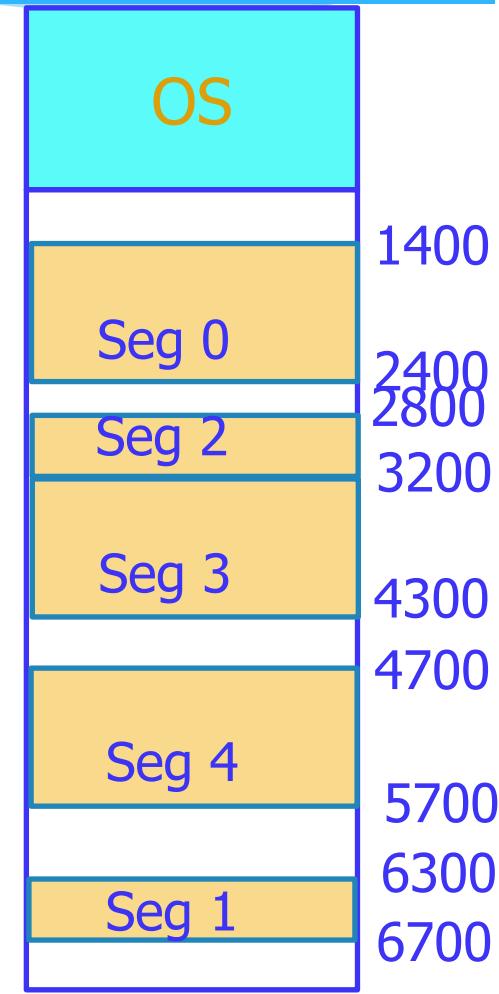
Example



Address <2,450> = ?

Access error!

M	A	L
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		



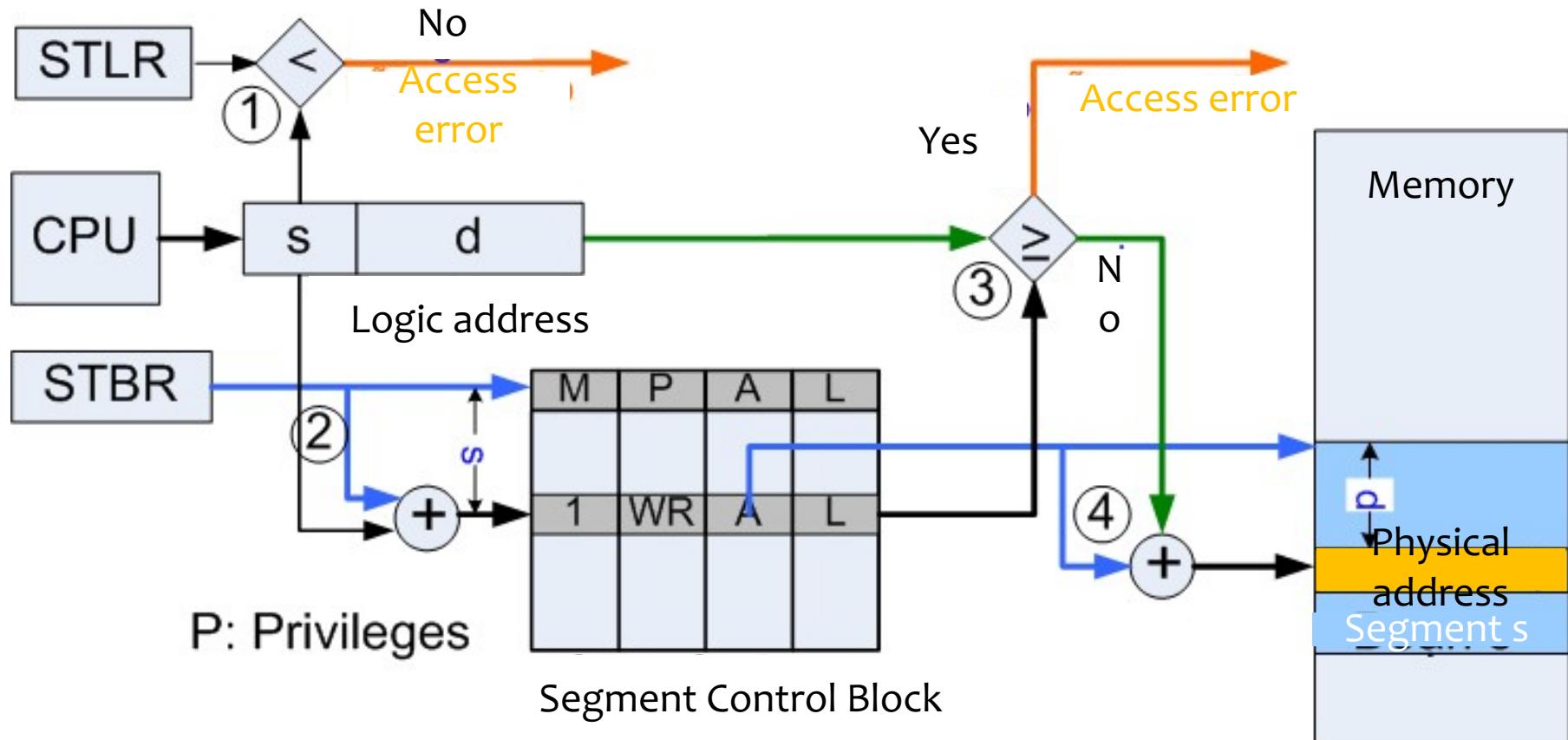
Memory

Chapter 3 Memory management

2. Memory management strategies

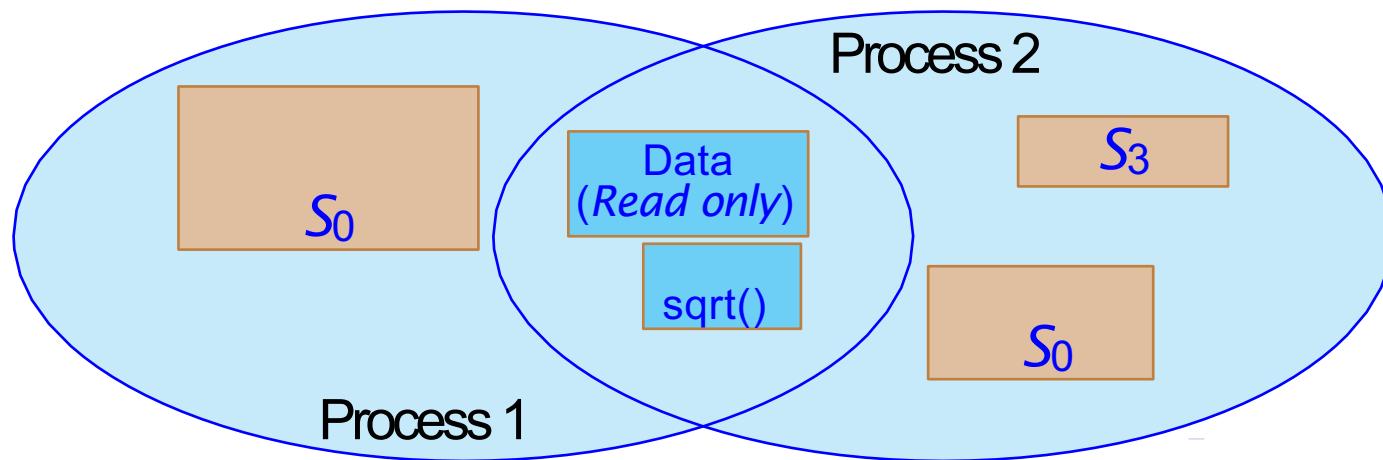
2.3 Segmentation strategy

Address conversion: memory accessing diagram



Conclusion: pros

- Module loading diagram does not require user's participation
- Easy to protect segments
 - Check memory accessing error
 - Invalid address : more than segment's length
 - Check accessing's property
 - Code segment: read only -> Write into code segment: accessing error
 - Check the right to access module
 - Add accessing right (user/system) into SCB
- Allow segment sharing (Example: Text editor)

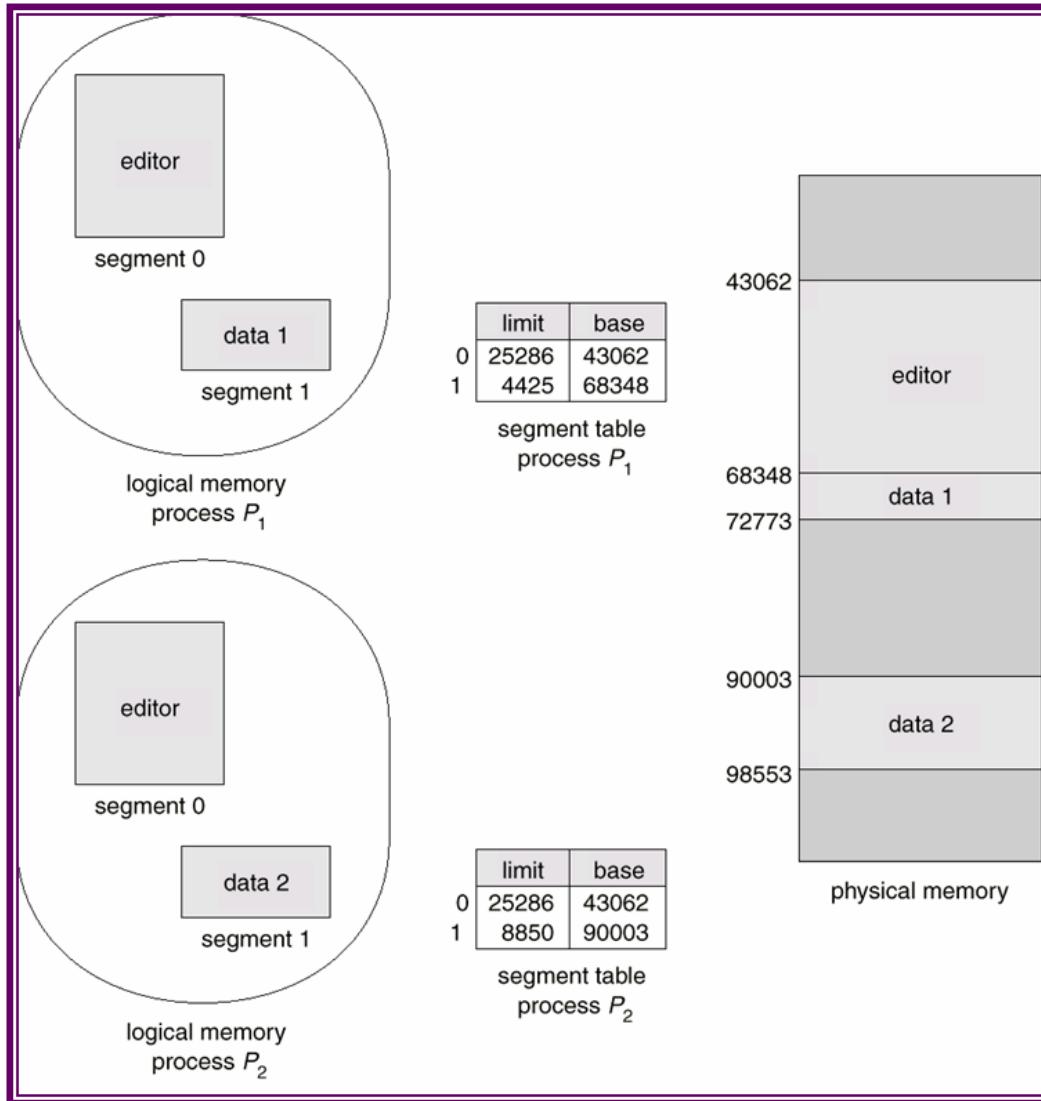


Chapter 3 Memory management

2. Memory management strategies

2.3 Segmentation strategy

Segment sharing: Main problem



- Sharing segment
 - Call (0, 120) ?
 - Read (1, 245) ?
- =>must has the **same index number** in the SCB

Conclude: cons

- Effective is depended on the program's structure
- Memory is fragmented
 - Memory allocated by methods first fit /best fit...
 - Require memory rearrangement (relocation, swapping)
 - Easier with the help of SCB
 - $M \leftarrow 0$: Segment is not in memory
 - Memory's area defined by A and L is returned to the free memory management list
 - Select which module to bring out
 - Longest existed module
 - Last recently used module
 - Least frequently used module \Rightarrow Require media to record number and time that module is accessed
 - Solution: allocate memory for equal size segment ([page](#))?

Chapter 3 Memory management

2. Memory management strategies

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- **Paging strategy**
- Segmentation and paging combination strategy

- Physical memory is divided into equal size blocks: page frames
 - Physical frame is addressed by number 0, 1, 2, . . . : frame's physical address
 - Frame is the unit for memory allocation
- Program is divided into blocks that have equal size with frame (pages)

Rule (cont)

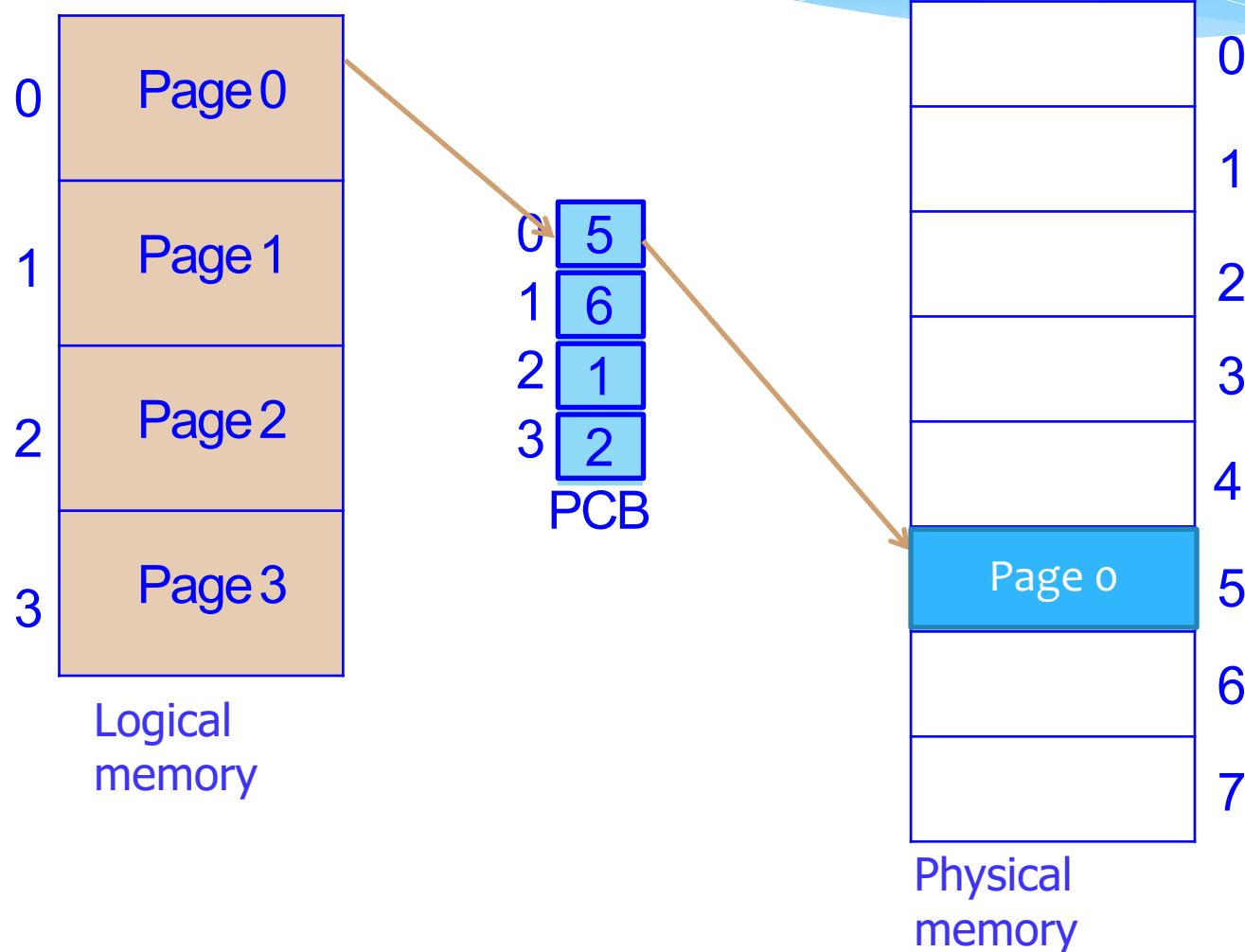
- When program is executed
 - Load logical **page** (from external memory) **into** page's frame
 - Construct a **PCB**(Page Control Block) to determine the **relation** between **physical frame** and **logical page**
 - **Each element** of PCB is corresponding to **a** program's **page**
 - Show which frame is holding corresponding page
 - Example $\text{PCB}[8] = 4 \Rightarrow ?$
 - Accessing Address is combined of
 - Page's number (p) : **Index** in PCB to find page's **base address**
 - **Displacement** in page (d): Combined with **base address** to find the **physical address**

Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Example

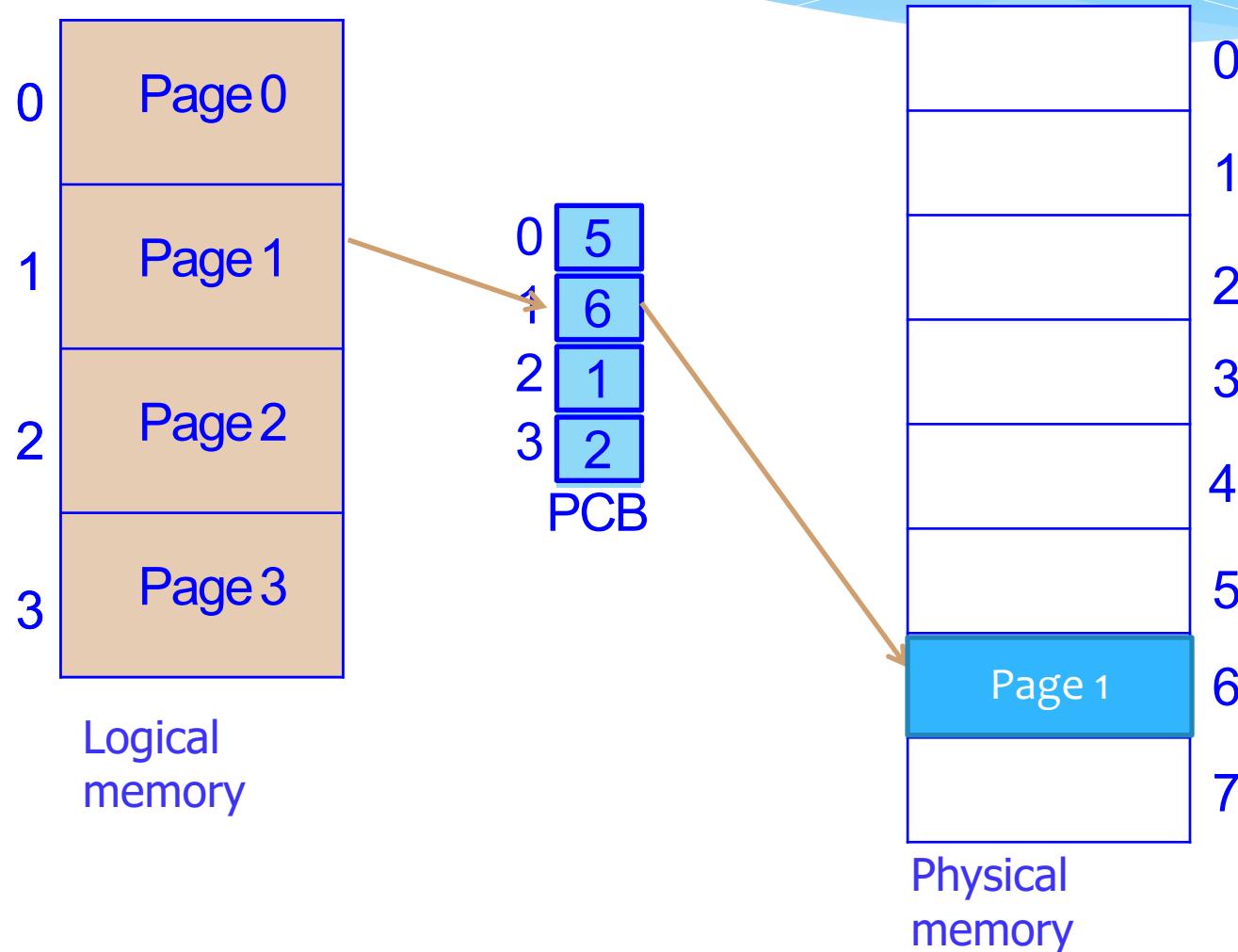


Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Example

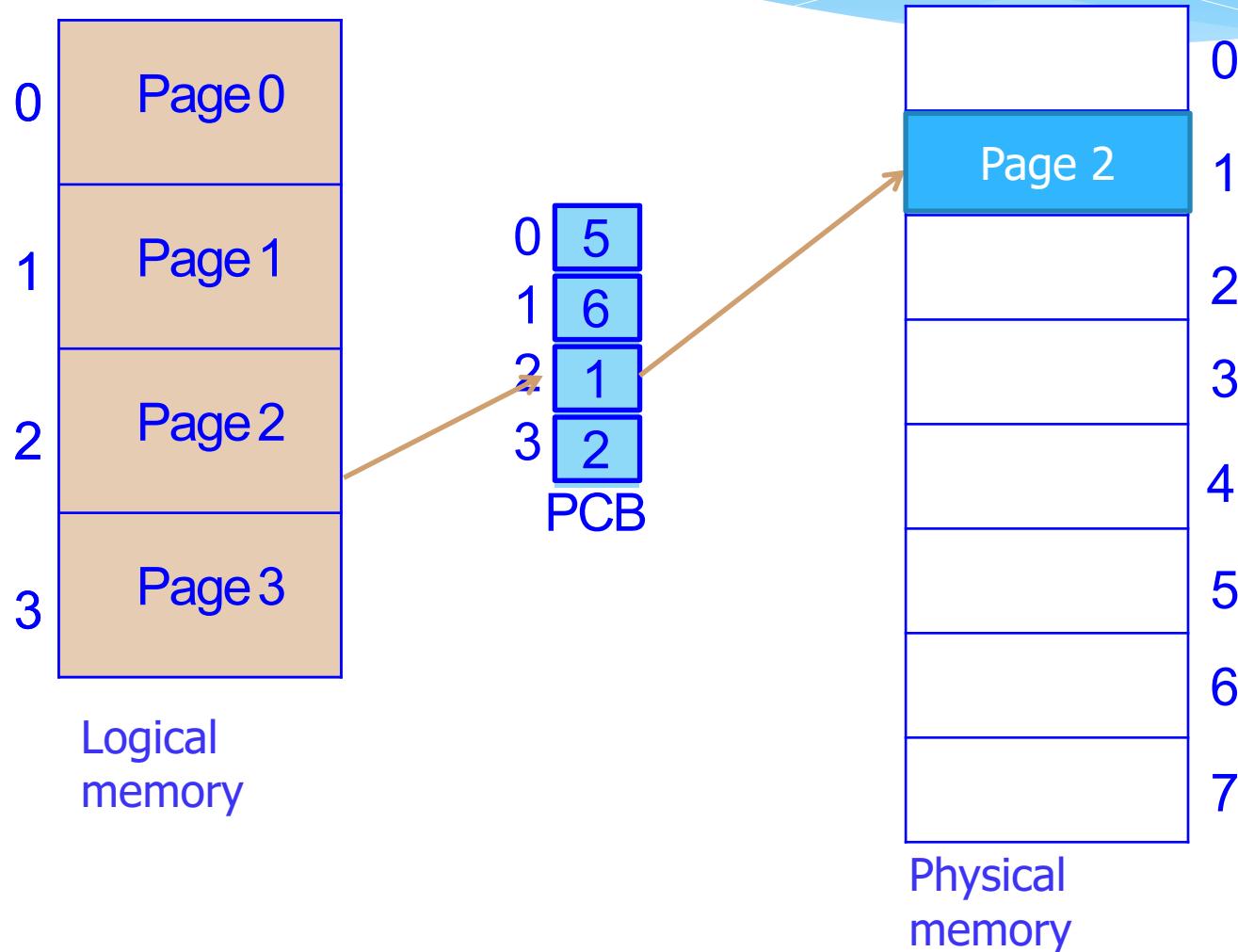


Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Example

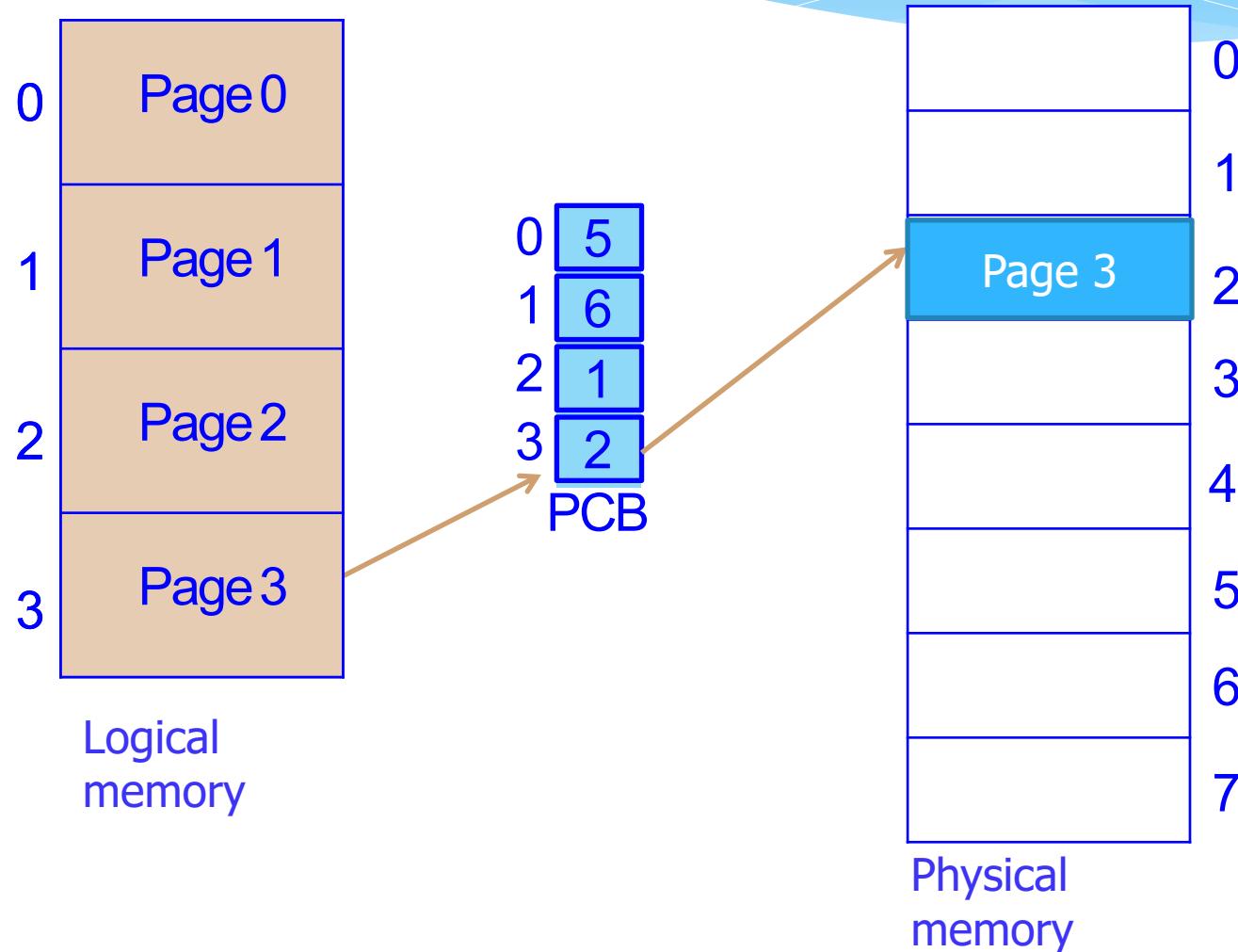


Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Example



Note

- Frame size is always power of 2
 - Allow connection between frame number and displacement
 - Example: memory is addressed by n bit, frame's size 2^k

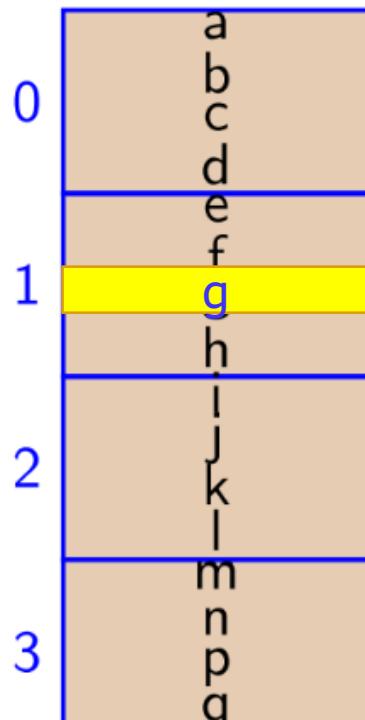
frame	displacement
$n - k$	k
- Not necessary to load all page into memory
 - Number of frame is limited by memory's size, number of page can be unlimited
 - PCB need Mark filed to know if page is already loaded into memory
 - $M = 0$ Page is not loaded
 - $M = 1$ Page is loaded
- Distinguish between paging and segmentation
 - Segmentation
 - Module is depend on program's structure
 - Paging
 - Block's size is independent from program
 - Block size is depend on the hardware (e.g.: $2^9 \rightarrow 2^{13}$ bytes)

Chapter 3 Memory management

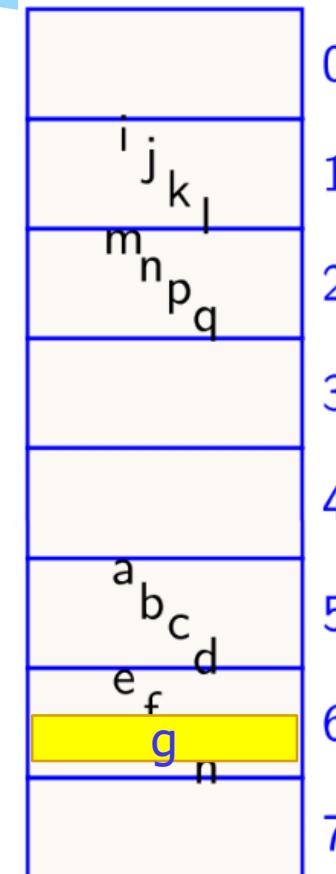
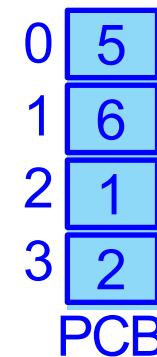
2. Memory management strategies

2.4 Paging strategy

Example



Logical memory



Physical
memory

Access the logic address [6] ?

Address [6]: Page 1, displacement 2

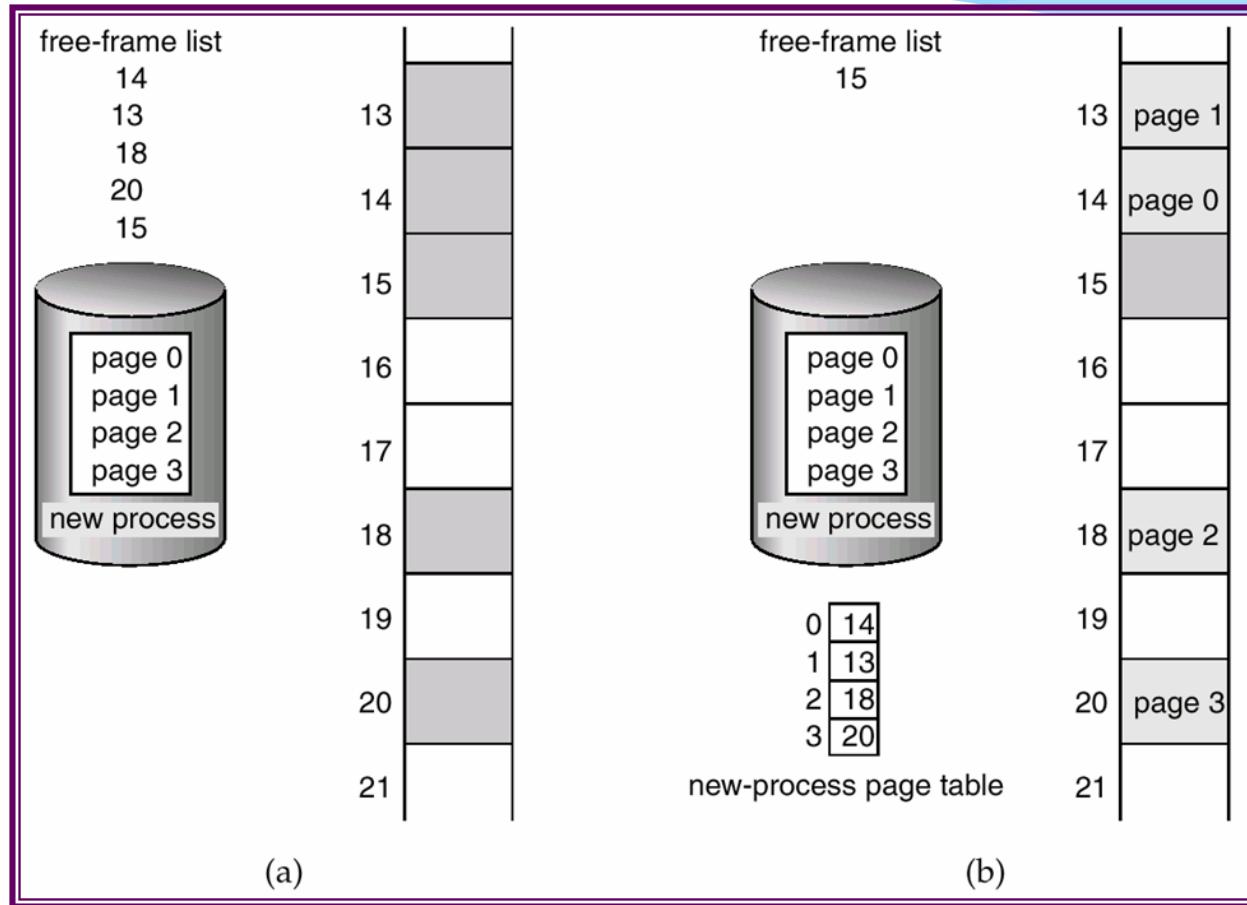
$$\text{Address } <1,2> = 6*4 + 2 = 26 \text{ (62}_4\text{)}$$

Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Program is running → Load program into memory



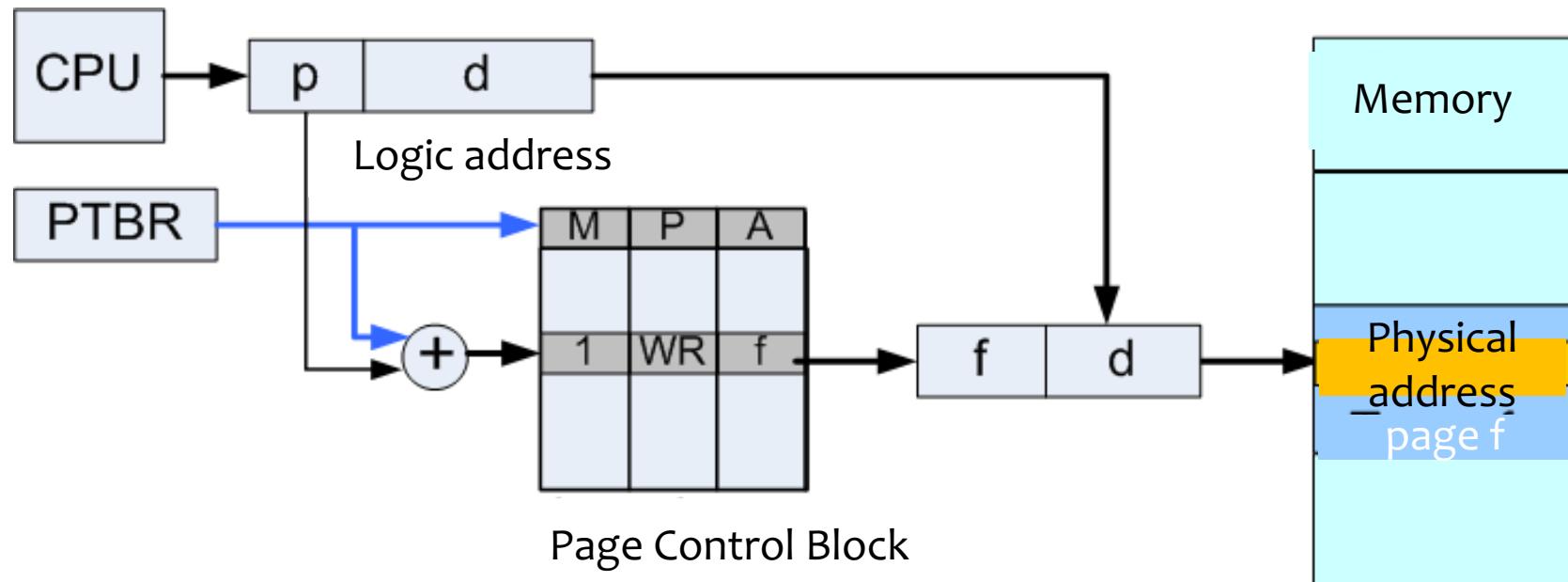
- If number of unused frame is enough ⇒ load all page
- If not enough ⇒ load parts of pages

Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Address conversion: Accessing diagram



- Remark

- Number of frame allocated to program
 - Large => Faster execution speed but parallel factor decrease
 - small => High parallel factor but execution speed slow because page is not inside memory
- ⇒ Effectiveness is depend on the page loading or page replacing strategy

- Page loading strategy

- Load all page: Load all program
- Prior loading: predict next page will be used
- Load on demand: Only load page when it's necessary

- Page replacing strategy

- FIFO First In First Out
- LRU Least Recently Used
- LFU Least Frequently Used
- ...

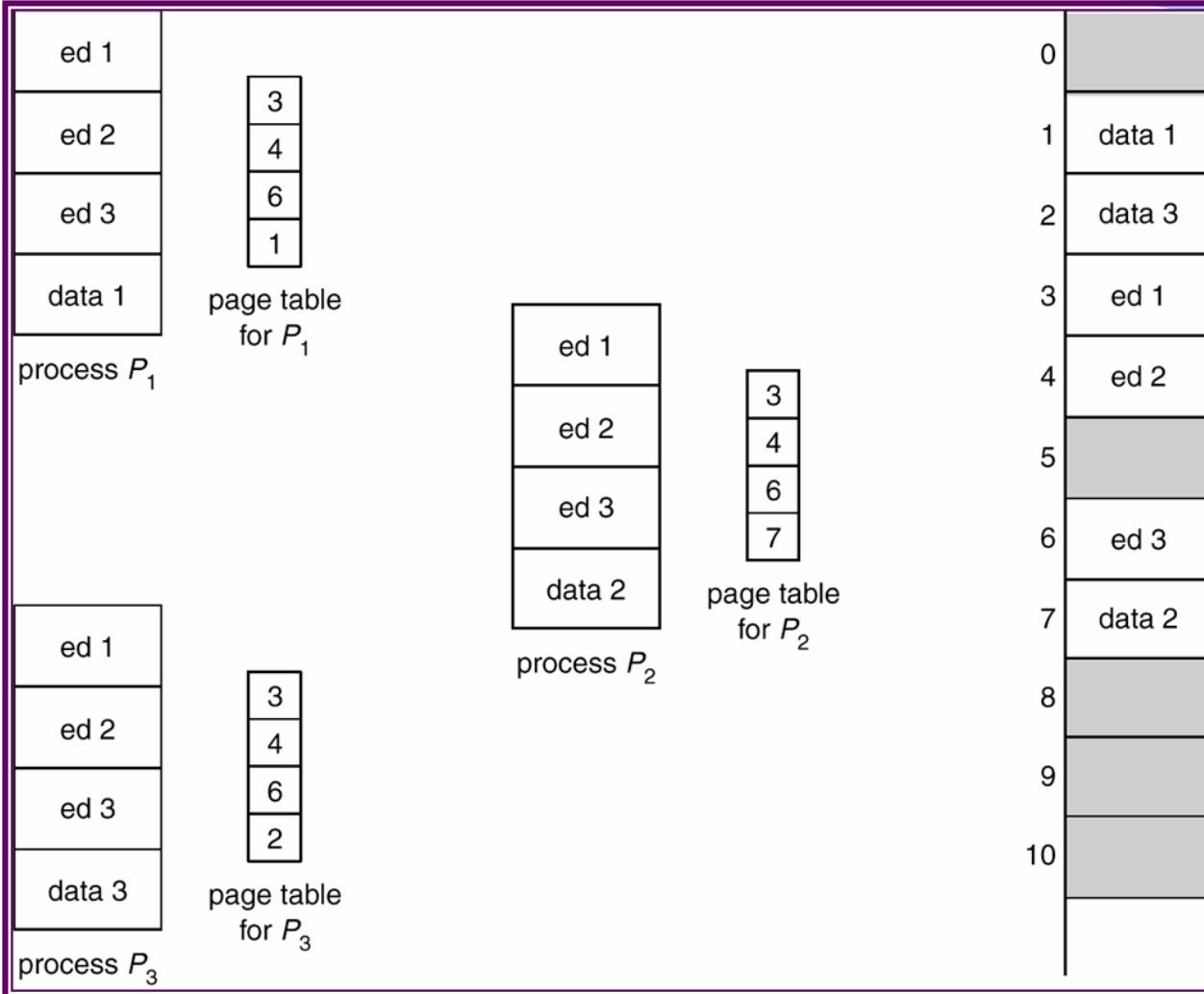
- Increase memory access speed
 - Access memory 2 times (PCB and required address)
 - Perform connecting instead of adding operation
- No external fragmentation phenomenon
- High parallel factor
 - Only need several program's page inside memory
 - Program can have any size
- Easy to perform memory protection
 - Legally access address (not more than page size)
 - Access property (read/write)
 - Access right (user/system)
- Allow sharing page between processes

Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Page sharing: Text editor



- Each page size 50K
- 3 pages for code
- 1 page for data
- 40 user
 - Need 8000K
- No sharing
 - Require 2150K
- Sharing
 - Require 2150K

- Necessary while working in sharing environment
 - Reduce the size of memory area for all processes
- Sharing code
 - Only one copy of sharing page in the memory
 - Example: text editor, compiler....
 - Problem: Sharing code can not change
 - Sharing page must be in the same logic address of all address
⇒ Same page id in the PCB
- The code and data are separately
 - Separate for each process
 - Can be in any position in the logic memory of the process

Disadvantage

- Have external memory fragmentation
 - Always appear at the last page
 - Reduce memory fragmentation by reduce page size ?
 - Page fault more frequent
 - Large page control table
- Require support from hardware
 - Cost for paging is high
- When the program is large, page control block has many members
 - Program size 2^{30} , page size $2^{12} \rightarrow$ PCB has 2^{20} members
 - Spend more memory for store PCB
 - Solution: multi level page

Rule: Divide PCB into pages

Example: 2 level paging

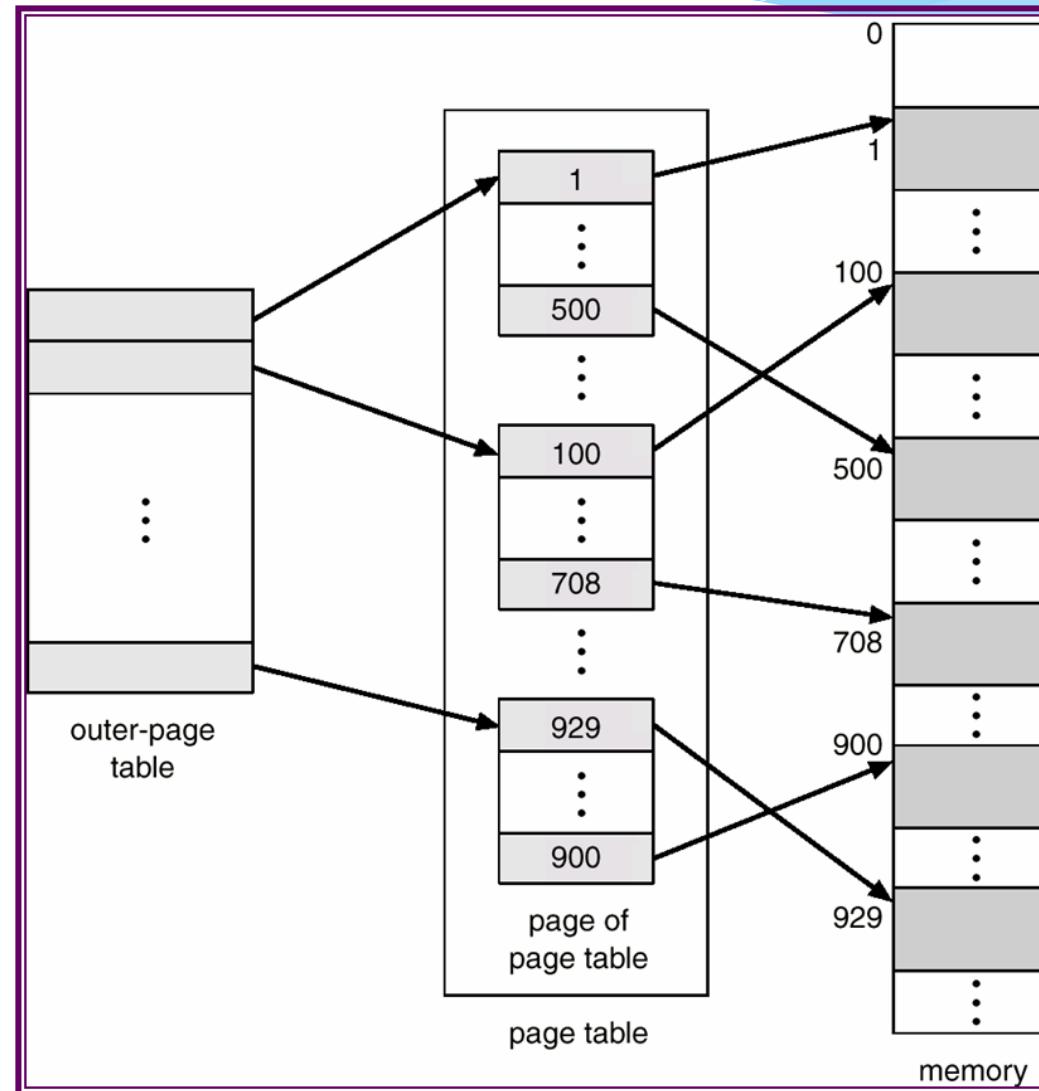
- Computer use 32 bit for addressing (2^{32}); Page size 4K (2^{12})
 - Page number - 20 bit
 - Offset in page - 12 bit
- PCB is paged. Page number is divided into
 - Outer page table (page directory) - 10 bit
 - Offset in a page directory – 10 bit
- Access address has the form $\langle p_1, p_2, d \rangle$

Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Multi-level page: Example 2 level paging

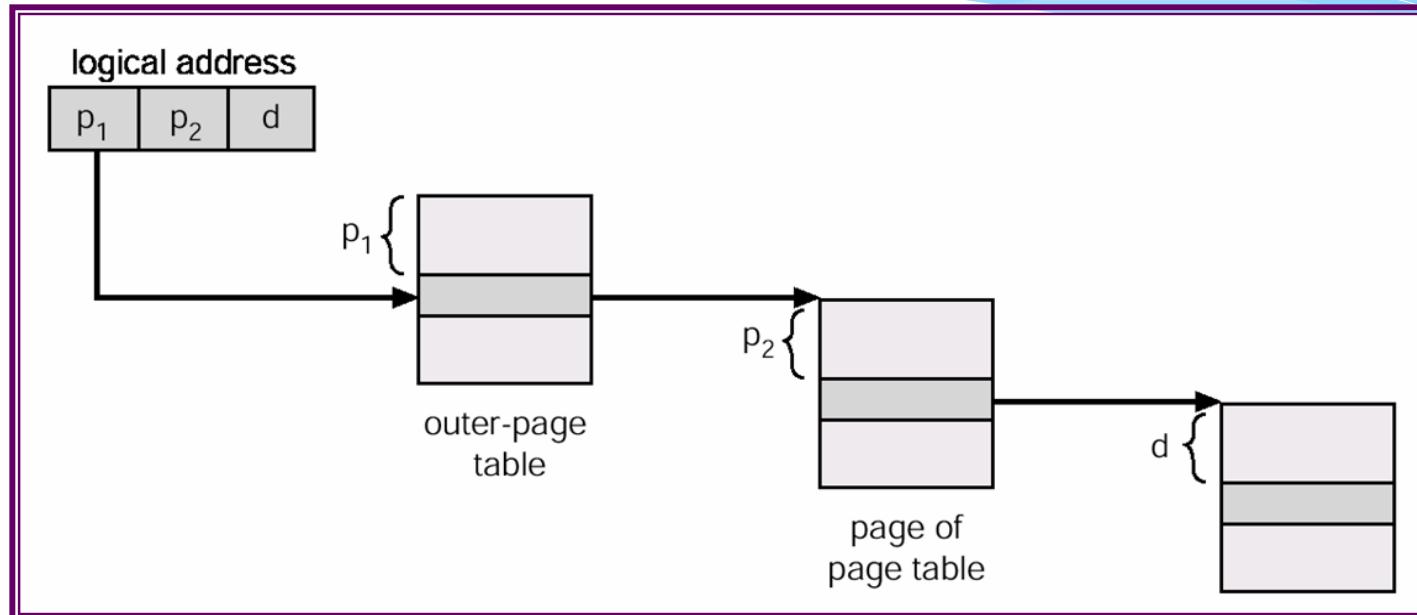


Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Multi-level page: Memory access



- When access: System load page directory into memory
- Unused page table and unused page are not necessary loaded into memory
- Require 3 times access memory
- Problem: For 64 bit system
 - 3, 4,... Level paging
 - Require access memory 4, 5,... times \Rightarrow slow
 - Solution: address translation buffer

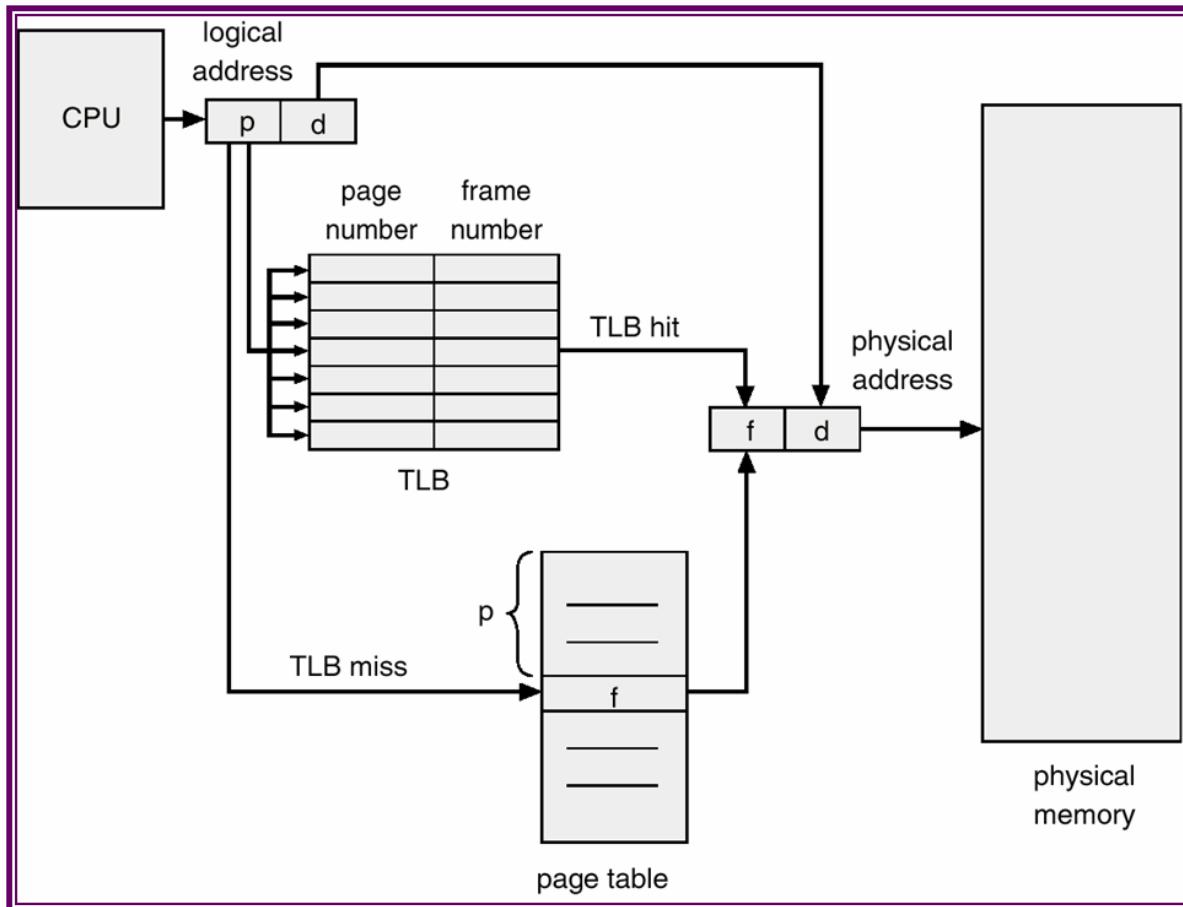
Chapter 3 Memory management

2. Memory management strategies

2.4 Paging strategy

Address translation buffer

TLB: translation look-aside buffers



- Associative registers
- Parallel access
- Each member contains
 - Key: Page number
 - Value: Frame number
- TLB contain recently accessing page
- When requested $\langle p, d \rangle$
 - Search p in TLB
 - Missing p , find p in PCB then add $\langle p, f \rangle$ into TLB

98% memory access is done via TLB

Chapter 3 Memory management

2. Memory management strategies

- Fixed partition strategy
- Dynamic partition strategy
- Segmentation strategy
- Paging strategy
- **Segmentation and paging combination strategy**

Rule

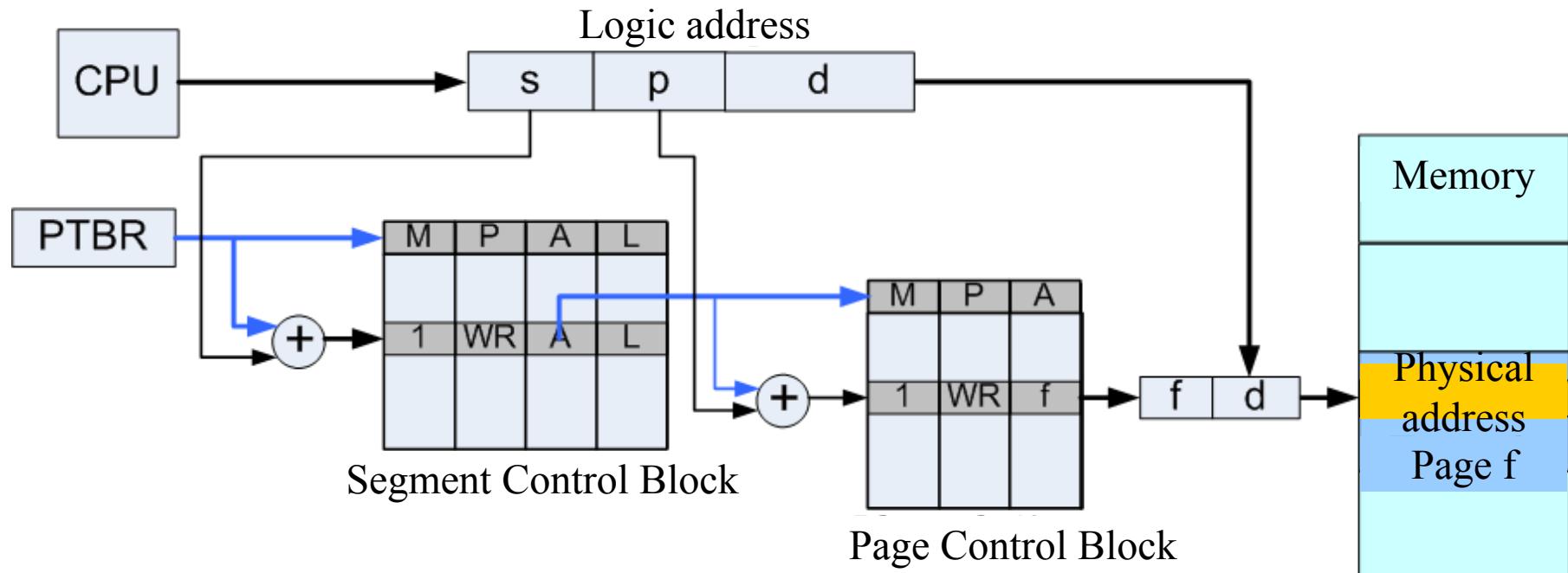
- Program is edited as in segmentation strategy
 - Create SCB
 - Each member of SCB correspond to one segment, has 3 fields M, A, L
- Each segment is edited separately as in paging strategy
 - Create PCB for each segment
- Memory accessing address: combination of 3 < s, p, d >
- Perform accessing address
 - STBR + s ⇒: address of s member
 - Check value of Ms, load PCBs if it's necessary
 - As + p ⇒ Load the address of member p in PCBs
 - Check value of Mp, load page p if it's necessary
 - Connect Ap with d => physical address if it's necessary
- Utilized in processor Intel 80386, MULTICS . . .

Chapter 3 Memory management

2. Memory management strategies

2. 5 Segmentation and paging combination strategy

Memory access diagram



Chapter 3 Memory management

2. Memory management strategies

2. 5 Segmentation and paging combination strategy

Conclusion

M_0 2340B

M_1 5730 B

M_2 4264 B

M_3 1766 B

Segmentation

M	A	L
0	—	2340
1	2140	5730
0	—	4264
0	—	1766

SCB

Segmentation and paging combination

M	A	L
0	—	3
0	5	6
0	—	5
0	—	2

SCB

M	A
0	—
1	8
0	—
0	—
0	—
0	—

PCB_2

Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ **Virtual memory**
- ④ Memory management in Intel's processors family

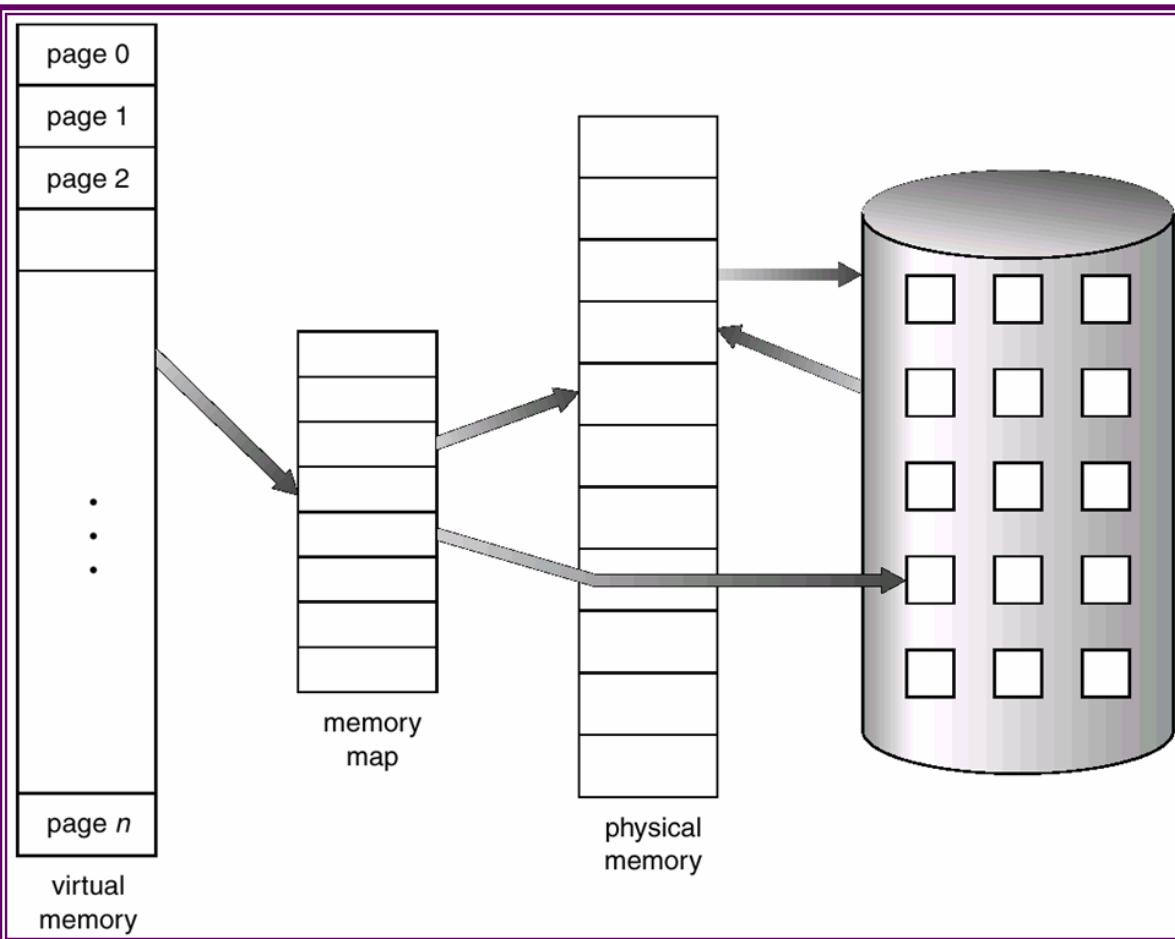
① Introduction

② Page replacement strategy

Program and memory issue

- Instruction must be placed in memory when executed !
- Whole program must stay inside memory ?
 - Dynamic loading, Overlays structures... : Partly loaded
 - Require special notice from programmer
- ⇒ Not necessary
- Program's segment for handling errors
 - Errors occur least frequently, least frequently executed
- Unused declared data
 - Declare a matrix 100x100, use 10x 10
- Run program with one part inside memory will allow
 - Write program in virtual address space
 - Unlimited size
 - Many program concurrently existed
 - ⇒ Increase CPU's productivity
 - Reduce I/O request for loading and swapping programs
 - The size of the swapped part is smaller

Concept of virtual memory



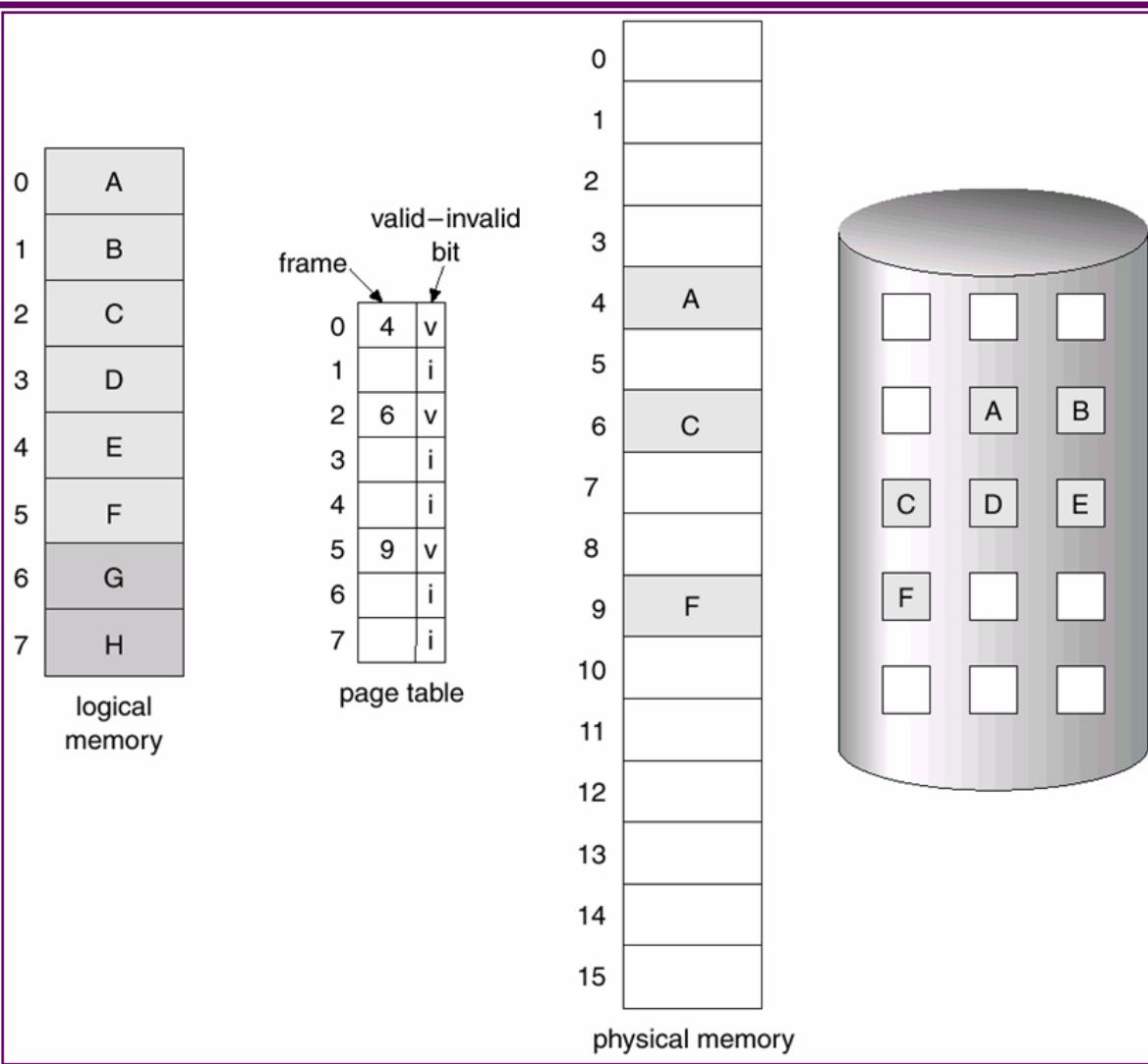
- Utilize the secondary storage (*HardDisk*) to store the unloaded program's part
- Separate logic memory (*user's space*) from physical memory
- Allow mapping large logical memory area to small physical memory area
- Implemented by
 - Segmentation
 - Paging

Chapter 3: Memory management

3. Virtual memory

3.1 Introduction

Load parts of program into memory



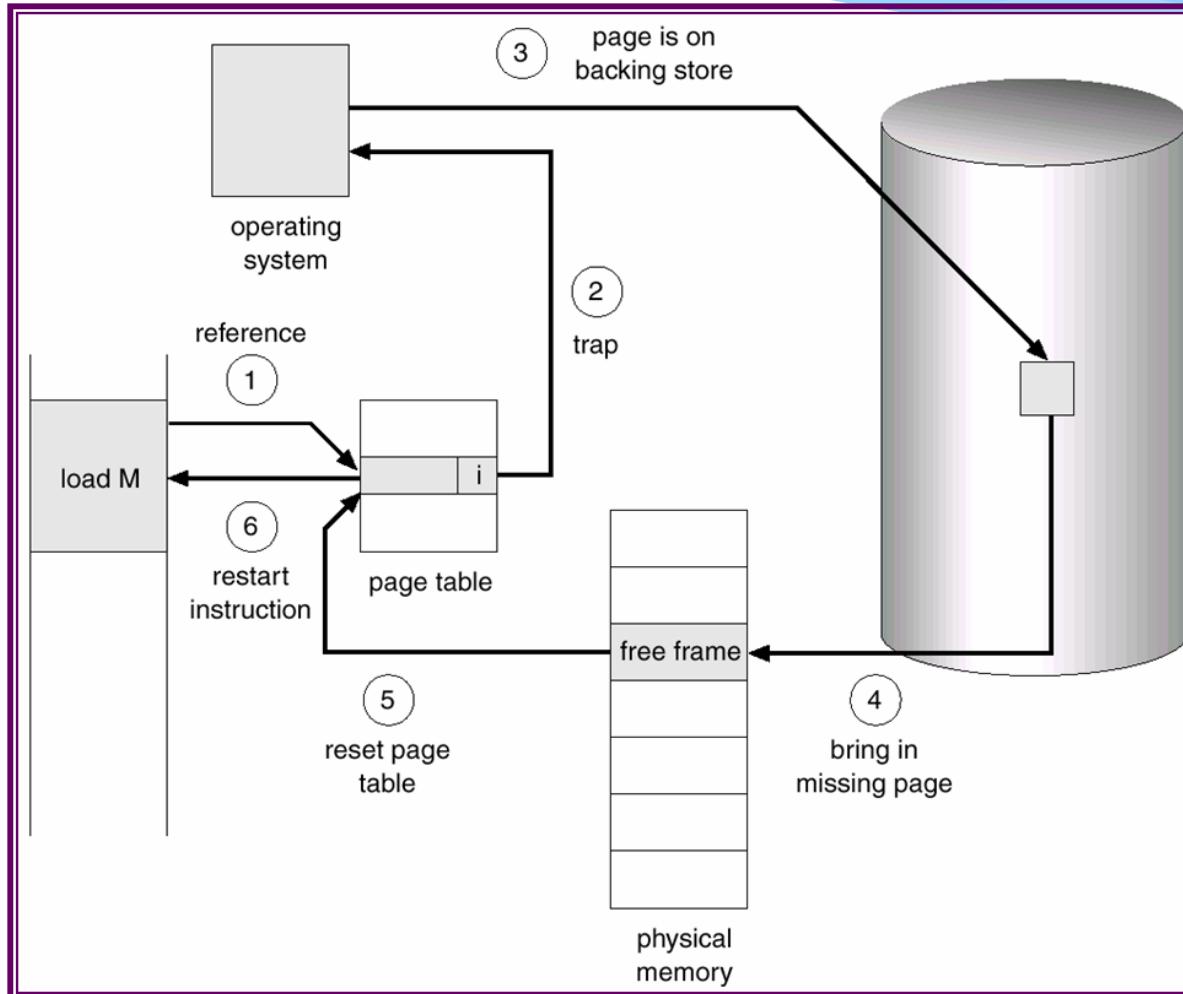
- Process's page:
 - Physical memory,
 - Some pages stay on disk (virtual memory)
- Represented by one bit in the PCB
- When a page is required, load page from secondary memory -> physical memory

Chapter 3: Memory management

3. Virtual memory

3.1 Introduction

Page fault handling



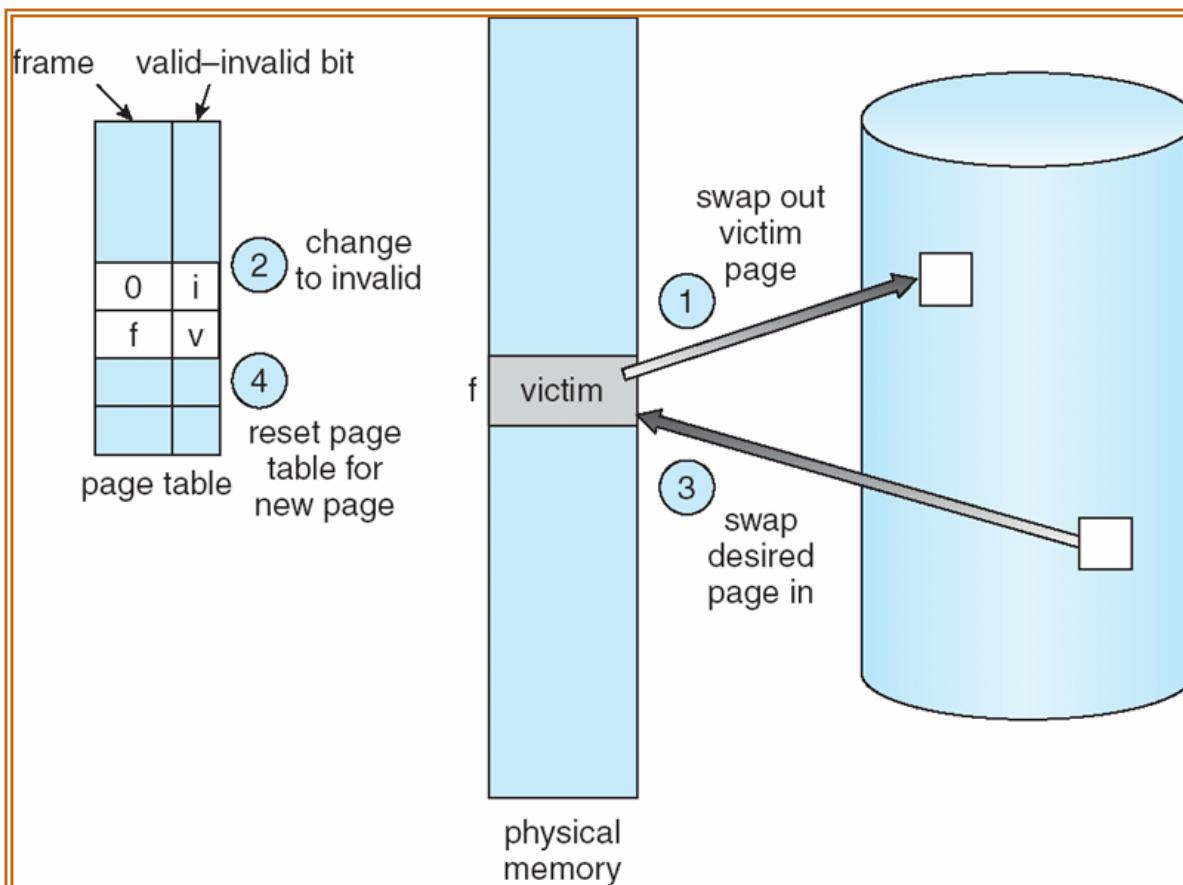
If there are no free frames, need to replace pages

Chapter 3: Memory management

3. Virtual memory

3.1 Introduction

Page replacement



- ① Determine location of logical page on disk
- ② Select physical frame
 - Write to disk
 - Modify bit **valid-invalid**
- ③ Load logic page into selected physical frame
- ④ Restart process

① Introduction

② Page replacement strategy

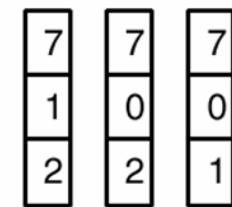
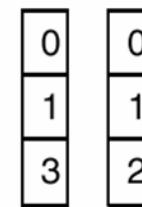
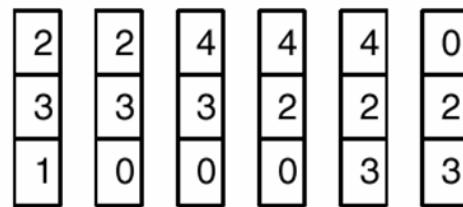
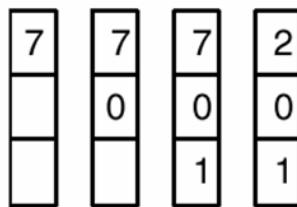
- FIFO: First In First Out
- OPT/MIN: Optimal page replacing algorithm
- LRU: page that is Least Recently Used
- LFU: page that is Least Frequently Used
- MFU: page that is Most Frequently Used
- . . .

FIFO

Example

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Remark

- Effective when the program has the linear structure
- Least effective when the program has different modules call
- Easy to implement
 - Utilize a queue to store program's pages inside memory
 - Insert into last position in the queue, replace page at the first position
- Increase physical page, no guarantee of reducing page fault
 - Accessing sequence: 1 2 3 4 1 2 5 1 2 3 4 5
 - 3 frames: 9 page faults; 4 frames: 10 page faults

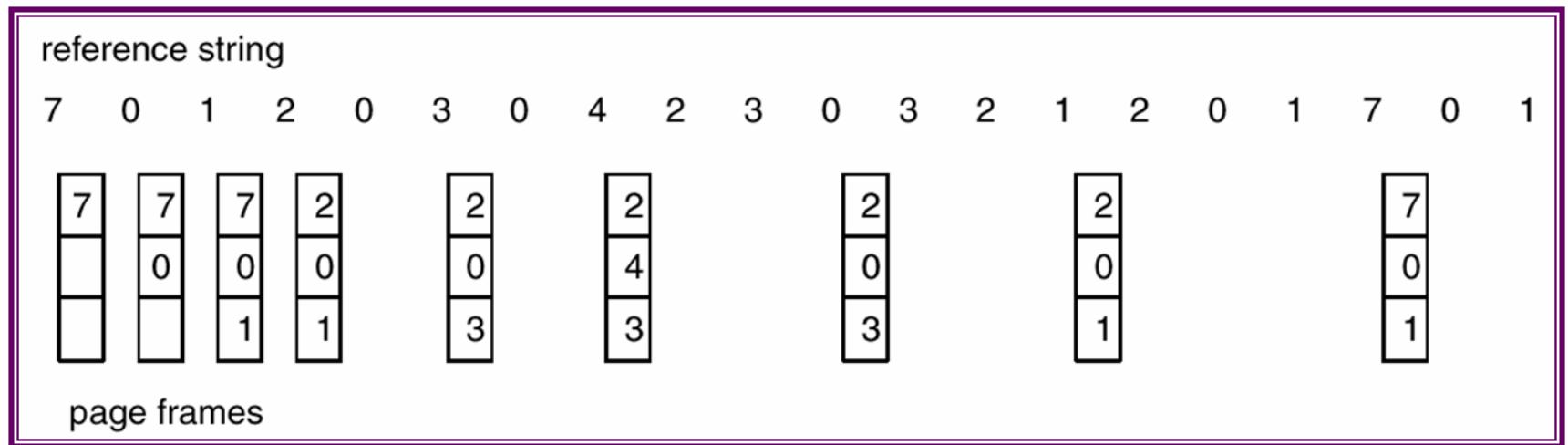
Chapter 3: Memory management

3. Virtual memory

3.2 Page replacement strategies

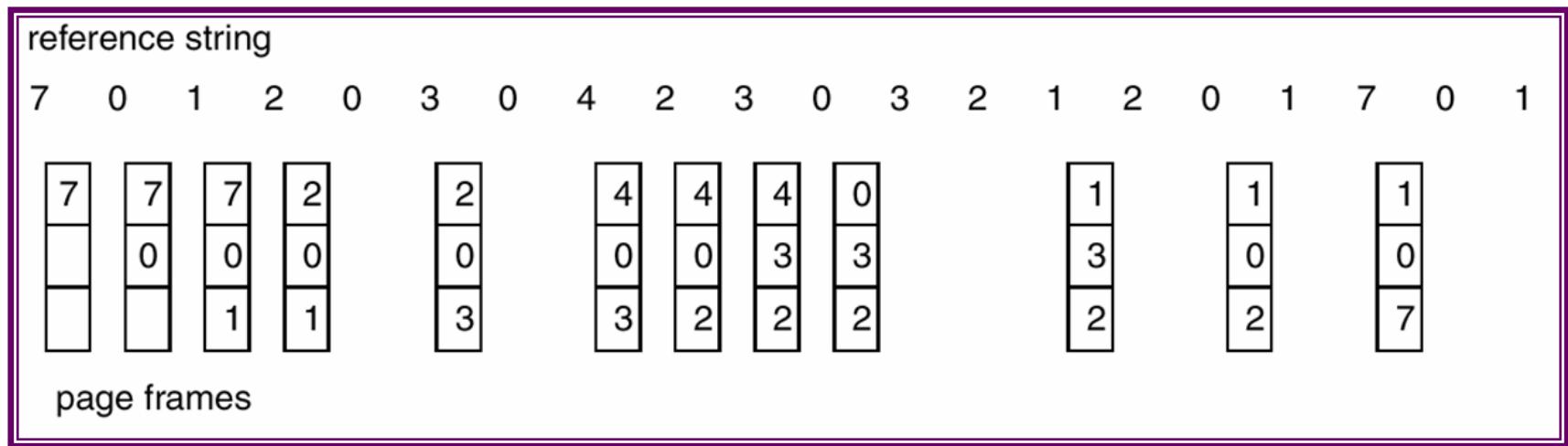
OPT

Rule: Replace page that has longest next used time



- Number of page fault is smallest
- Problem: it's hard to predict program sequence

Rule: Replace page that is least recently used



- Effective for page replacing strategy
- Guarantee that page fault is reduced when increase physical frame
 - Set of pages in memory with n frames is always a subset of pages in memory that have $n + 1$ frames
- Require support to know the last recently accessed time
- How to implement?

LRU: Implementation

- Counter
 - Add one more field to record the accessed time into each member of PCB
 - Add a clock/counter to the Control Unit
 - Where there is a page access request
 - Increase counter
 - Copy the counter content into the newly added field in PCB
 - Need a procedure to update PCB (write to the added field) and procedure to search for a smallest accessed time value
 - Number Overflow phenomenon !?
- List or Stack
 - Use a list or stack to record page number
 - Access to a page, put corresponding member to the top position
 - Replace: member in the last position
 - Usually implemented as a 2 dimension linked list
 - 4 pointer assigning operation ⇒ time consuming

Algorithm based on counter

Use counter (one field of PCB) to record the last time page is accessed

- LFU: Page that has smallest counter will be replaced
 - Page that is frequently used
 - Important page \Rightarrow reasonable
 - Initialization page, only used at start \Rightarrow unreasonable \Rightarrow Shift right the counter by 1 bit (time div 2)
- MFU: Replace page with largest counter
 - Page with smallest value, just recently loaded and not used much

Chap 3 Memory Management

- ① Introduction
- ② Memory management strategies
- ③ Virtual memory
- ④ Memory management in Intel's processors family

Memory management modes

- Intel 8086, 8088
 - Only one management mode: Real Mode
 - Managed memory area up to 1MB (20bit)
 - Xác định địa chỉ ô nhớ bằng 2 giá trị 16 bit: Segment, Offset
 - Segment register: CS, SS, DS, ES,
 - Offset register: IP, SP, BP...
 - Physical address: Seg SHL 4 + Ofs
- Intel 80286
 - Real mode, compatible with 8086
 - Protected mode
 - Utilize segmentation method
 - Exploit physical memory up to 16M (24bit)
- Intel 80386, Intel 80486, Pentium,..
 - Real mode, compatible with 8086
 - Protected mode : Combination of segmentation and paging
 - Virtual mode
 - Allow to run 8086 code in protected mode

Protected mode in Intel 386, 486, Pentium,..

