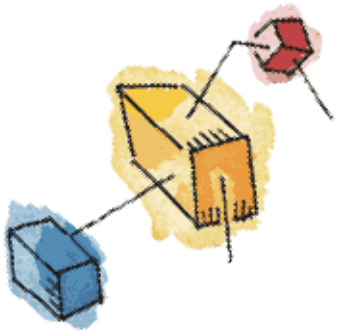
Operating Systems

Memory Management

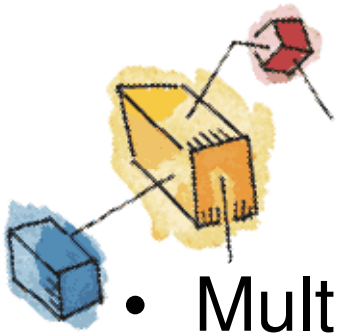
Dr. Shamim Akhter



CPU Utilization

- CPU utilization = $1 - P^n$
  - n process in memory
  - P is the I/O block time by one process
  - P^n probability of all process are waiting for I/O
- I/O waiting 80% of time
  - CPU utilization = $1 - 0.8^1$ =20% [1 process @ memory]
  - CPU utilization = $1 - 0.8^3$ =49% [3 processes @ memory]
  - CPU utilization = $1 - 0.8^{10}$ =89% [10 processes @ memory]

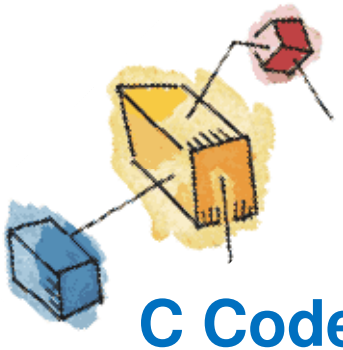
Note: in reality all process do not spend the same amount of I/O waiting time



Multiprogramming

- Multiple Programs (>1) loaded at the same time
  - into different areas of memory
- Processor needs to context switch
- Programs use memory addresses / references

- Proper addressing is required
  - no matter where the prog is located in memory!



Memory ref. @ program

C Code

A[12]=h + A[12]

Assembly Code

PC →

```
lw   $t0,  48 ($t1)
add  $t0, $s2, $t0
Sw $t0, 48 ($t1)
```

Memory

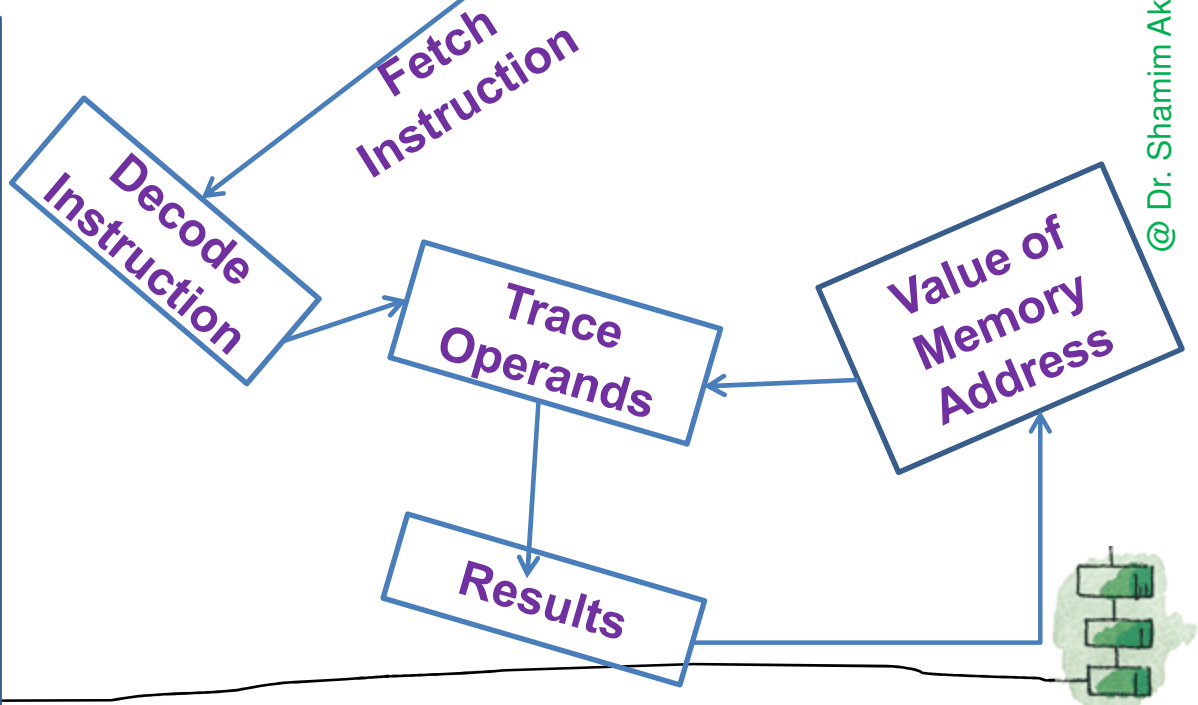
```
1000110100101000
0000000000110000
0000001001001000
0100000000100000
```

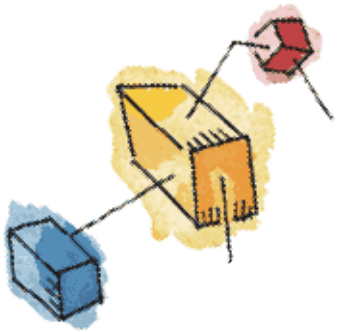
Memory Unit

Computer Hardware Unit having all memory ref.

Sees only a stream of memory addresses, does not know how they are generated.

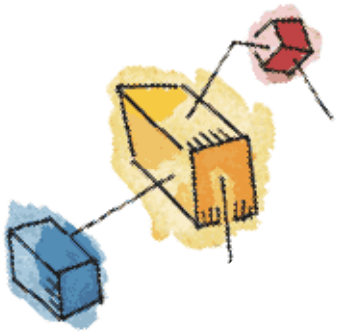
By the:
Instruction counter-PC
Indexing- Memory Index
Indirection-accessing a variable through the use of a pointer.





Memory Management

- Subdividing memory
  - to accommodate multiple processes
- Memory needs to be allocated
  - to ensure a reasonable supply of ready processes to consume available processor time

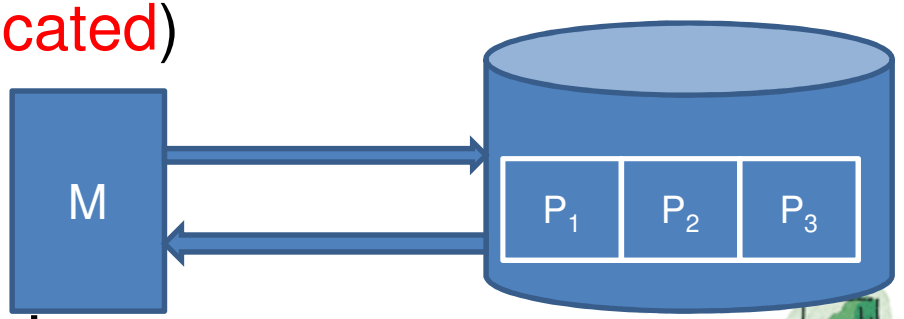


Memory Management Requirements

- Relocation
- Protection
- Logical organization
- Physical organization

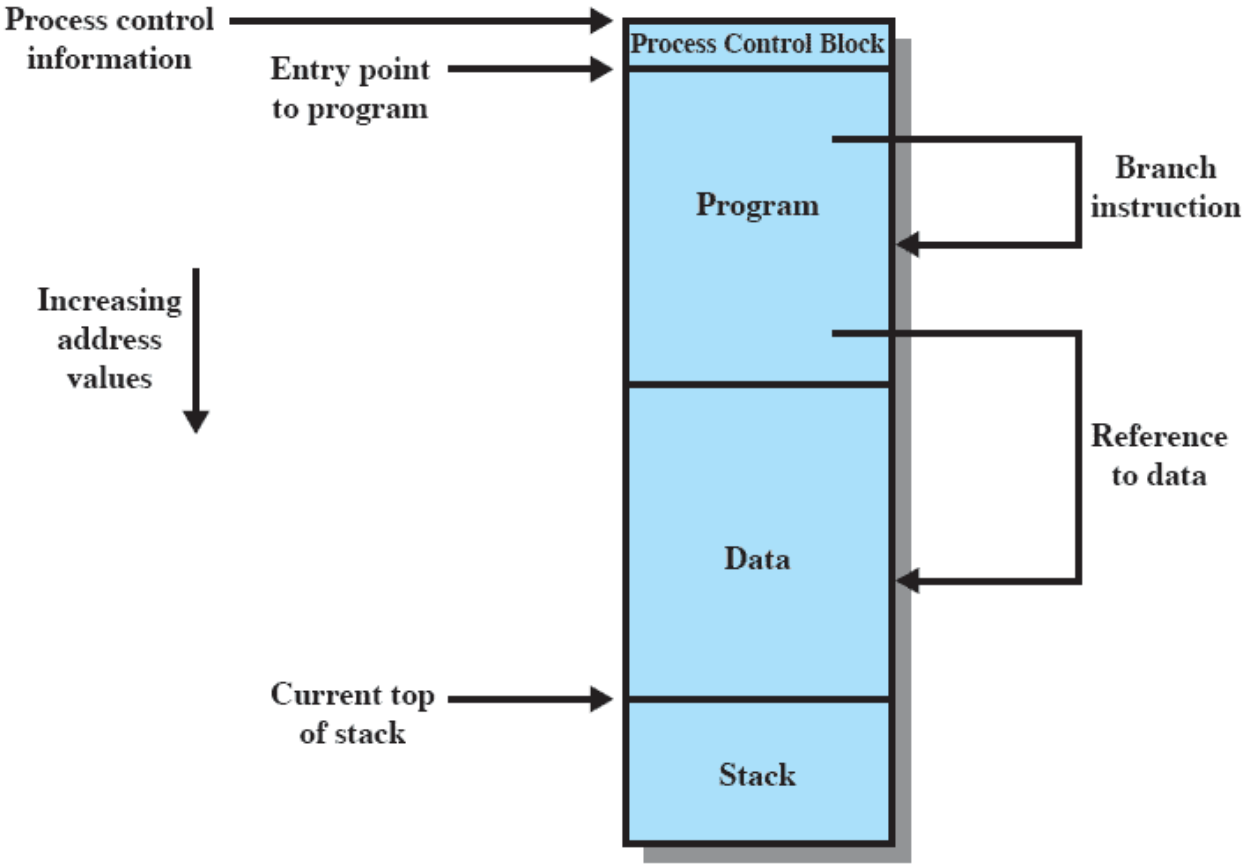
Memory Management Requirements

- 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)



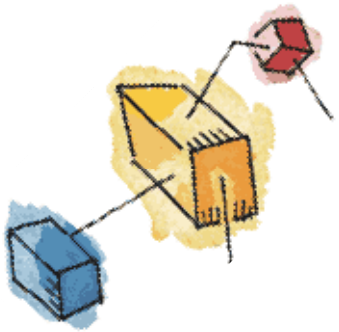
  - Memory references must be translated in the code to actual physical memory address

Addressing Requirement



The processor hardware and operating system software must be able to translate the memory references found in the code of the program into actual physical memory addresses (current memory location)

Figure 7.1 Addressing Requirements for a Process



- User programs go through several steps before being executed

Steps for Loading a Process in Memory

Source file 1

Source file 2

Source file 3

Assembler or Compiler

Object module 1

Object module 2

Object module 3

System Library

static linking

Linker

Executable binary file (load module)

Loader

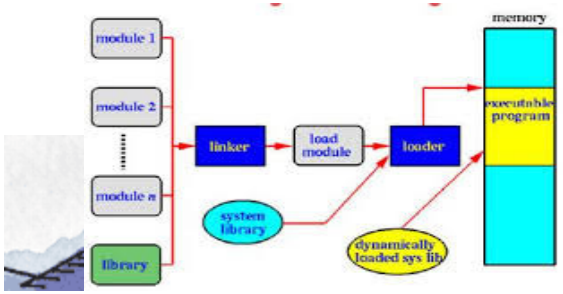
Process image in memory

Linking Loader

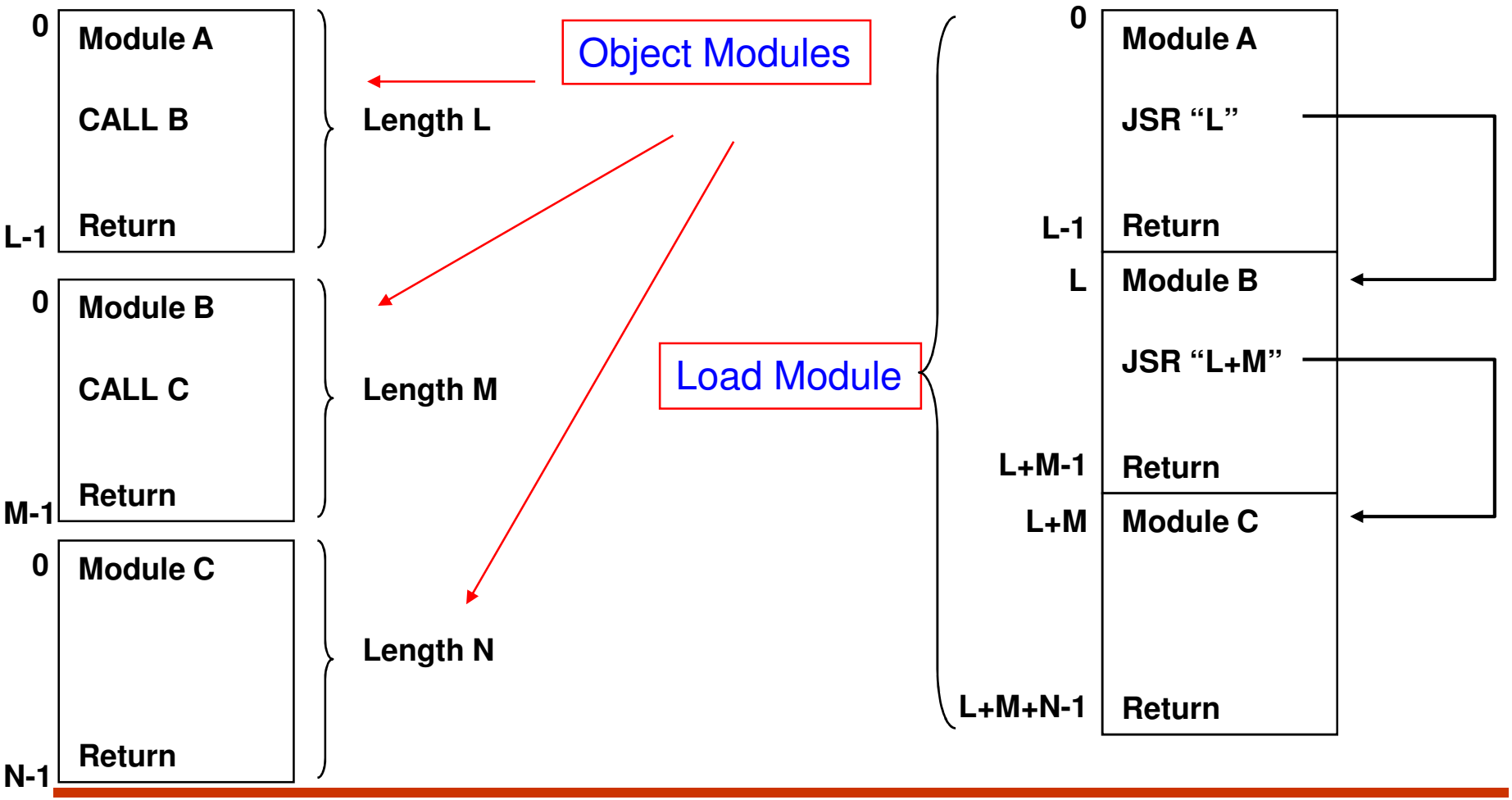
System Library

dynamic linking

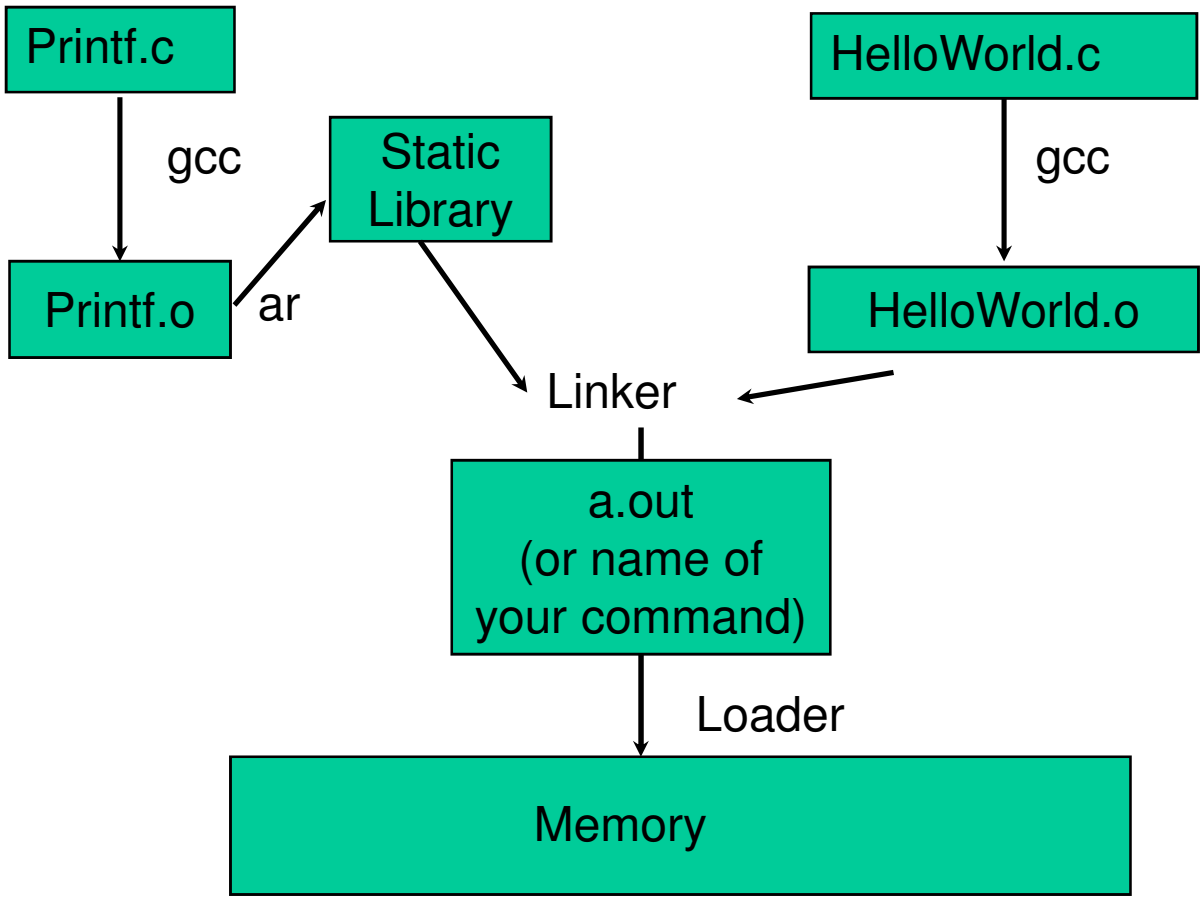
Linking: combine all the object modules of a program into a binary program image



The Linking Function

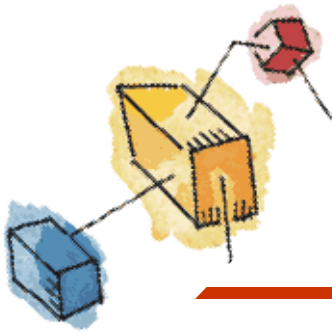


Static Linking and Loading



Dynamic Linking

- The linking of some external modules is done after the creation of the load module (executable file)
  - Windows: external modules are .DLL files
  - Unix: external modules are .SO files (shared library)
- The load module contains references to external modules which are resolved either at:
  - Loading time (load-time dynamic linking)
  - Run time: when a call is made to a procedure defined in the external module (run-time dynamic linking)
- OS finds the external module and links it to the load module
  - Check to see if the external module has been loaded into memory



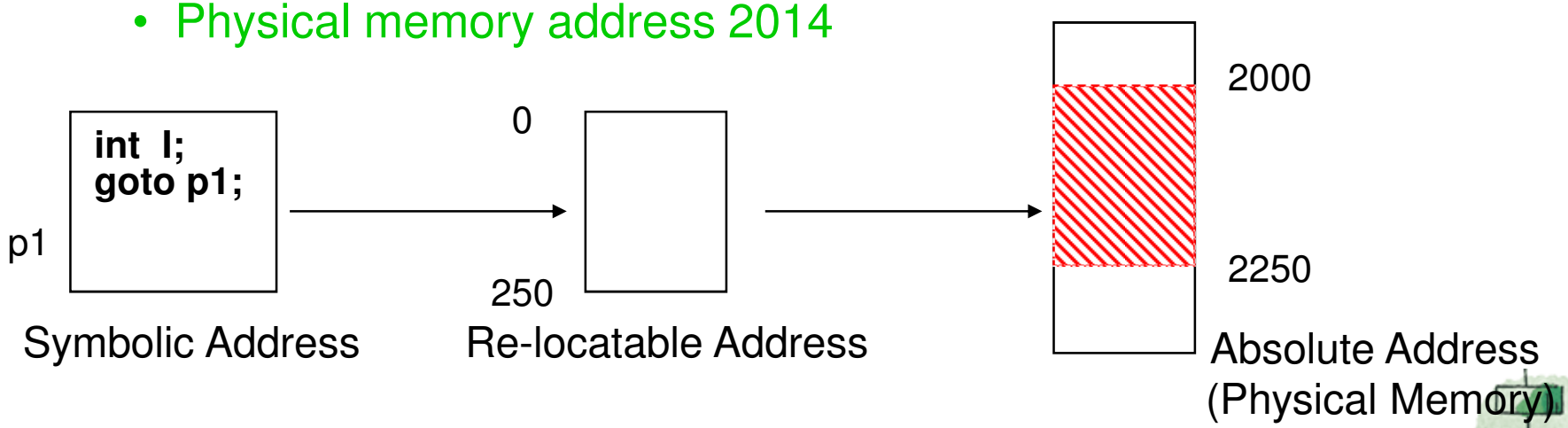
Advantages of Dynamic Linking

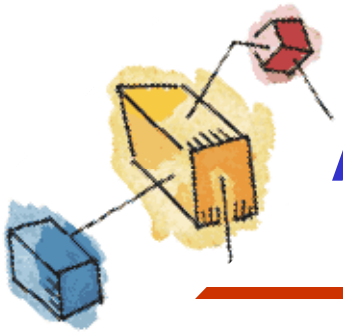
- The external module is often an OS utility. Executable files can use another version of the external module without the need of being modified

- Code sharing: the same external module needs to be loaded in main memory only once. Each process is linked to the same external module
  - Saves memory and disk space

Address Binding – Mapping from one address space to another

- Address representation
  - Source program: symbolic (such as count)
  - After compiling: re-locatable address
    - 14 bytes from the beginning of this module
  - After linkage editor, loader or run-time referring: absolute address
    - Physical memory address 2014

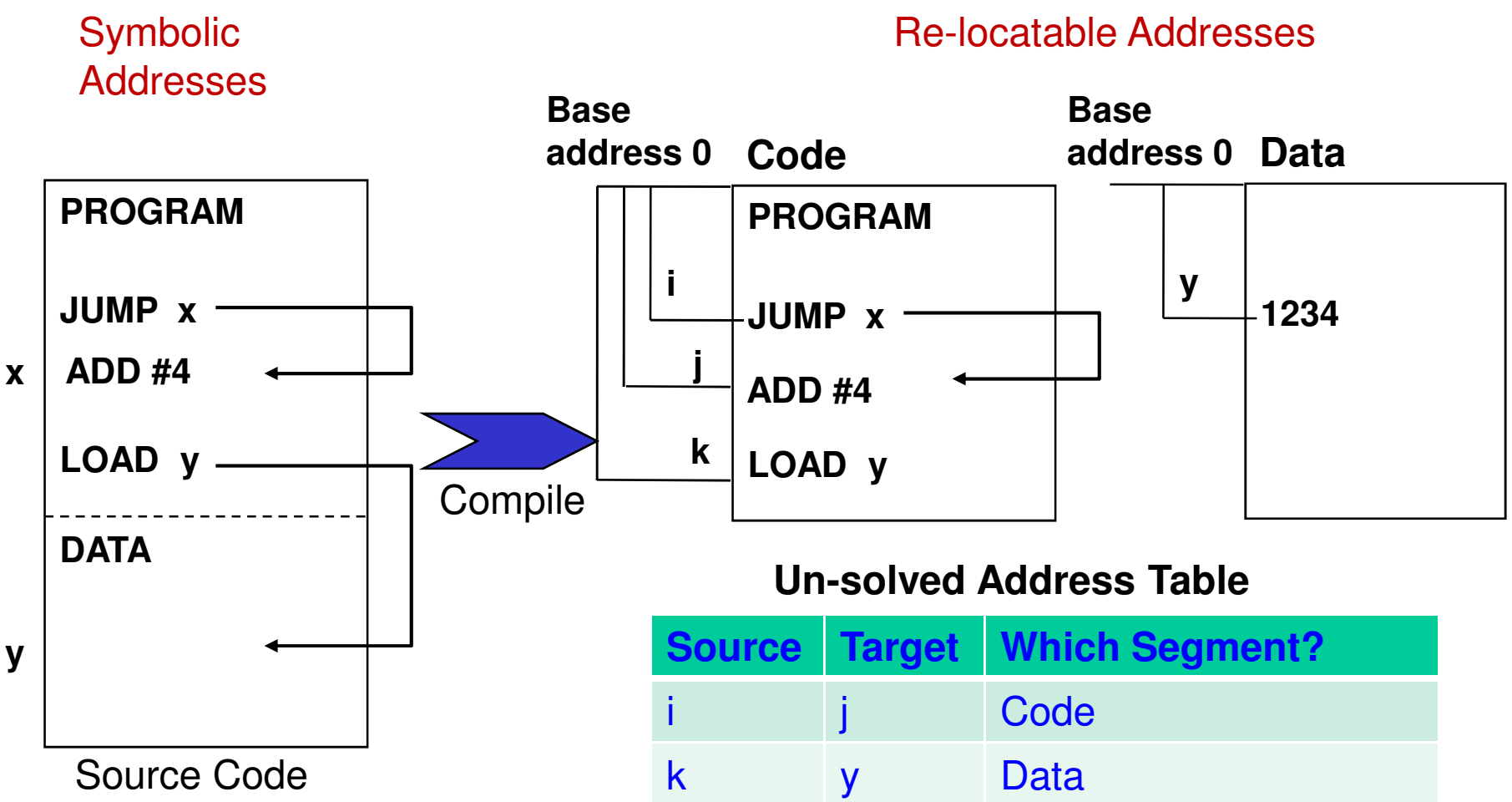




Address Binding (Cont.)

- Address binding of instructions and data to physical memory addresses can happen at three different stages
  - Compile time: If memory location known a priori, absolute code can be generated
    - Must recompile code if starting location changes
    - MS-DOS .COM-format program

Binding at Compile Time (Mapping)



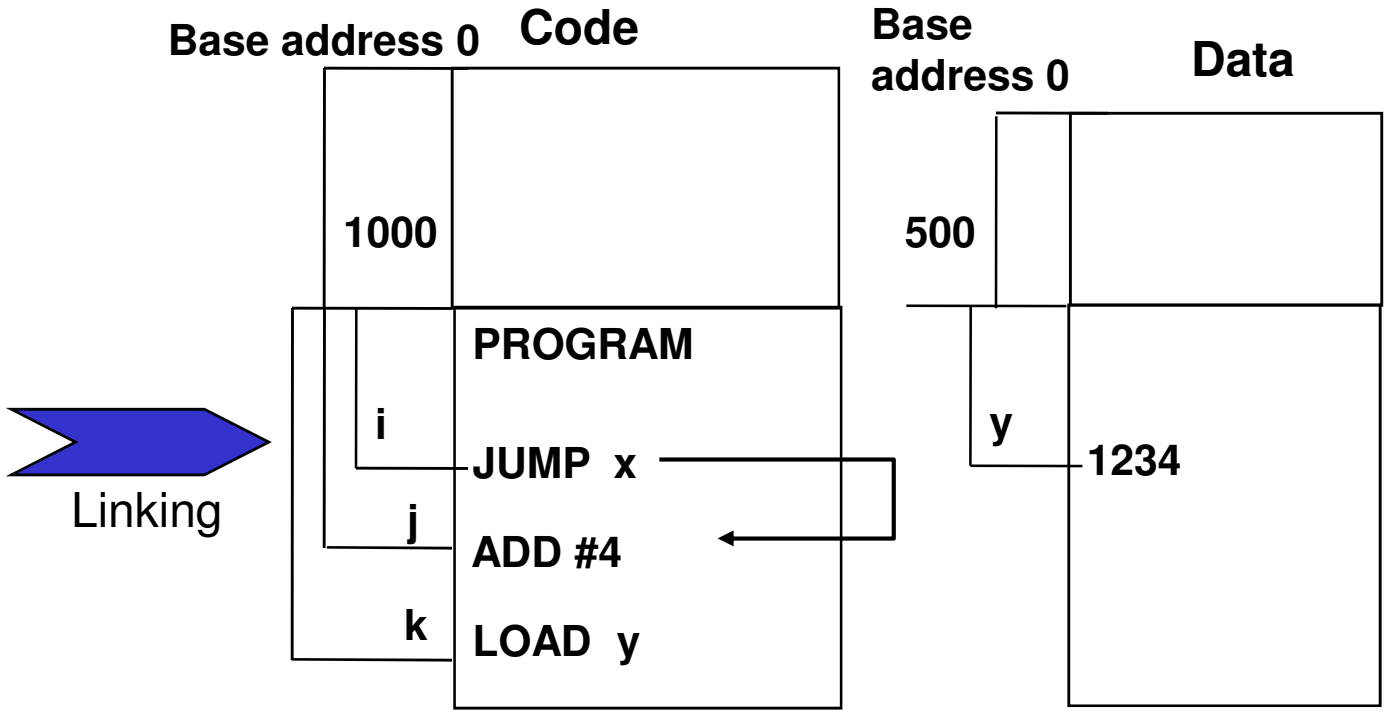
Un-solved Address Table

Source	Target	Which Segment?
i	j	Code
k	y	Data

The CPU generates the absolute addresses

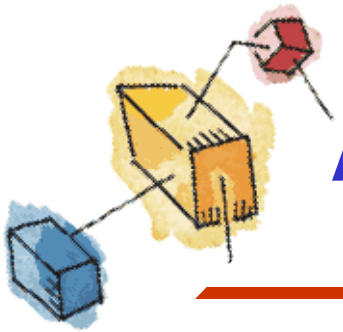
Binding at Compile Time (Cont.)

Address Generate: Static Linking



Un-solved Address Table

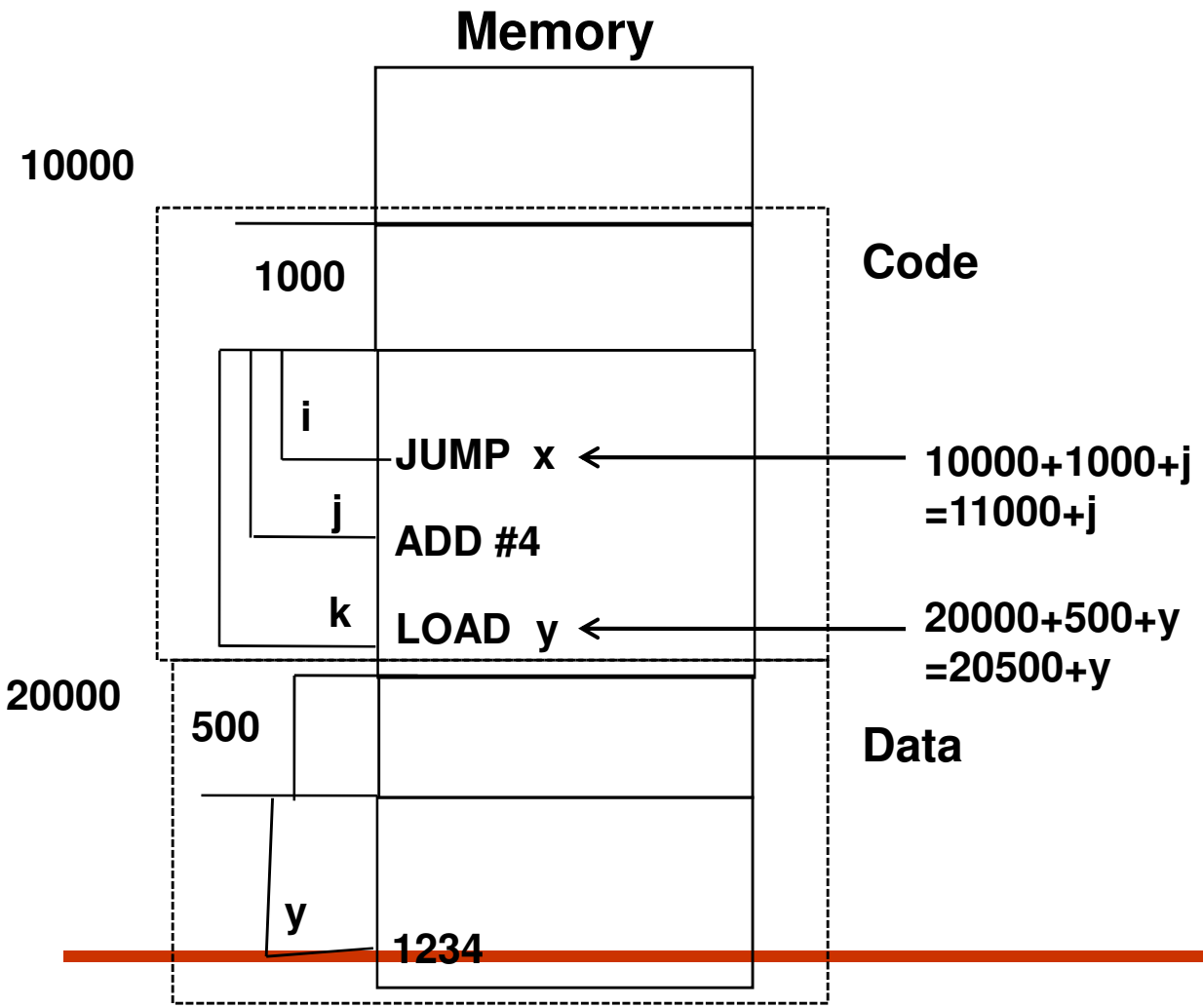
Source	Target	Which Segment?
i+1000	j+1000	Code
k+1000	y+500	Data



Address Binding (Cont.)

- Address binding of instructions and data to physical memory addresses can happen at three different stages
  - Load time: Must generate re-locatable code if memory location is not known at compile time
    - Physical memory address is fixed at load time
    - Must reload if starting location changes

Binding at Loader Time

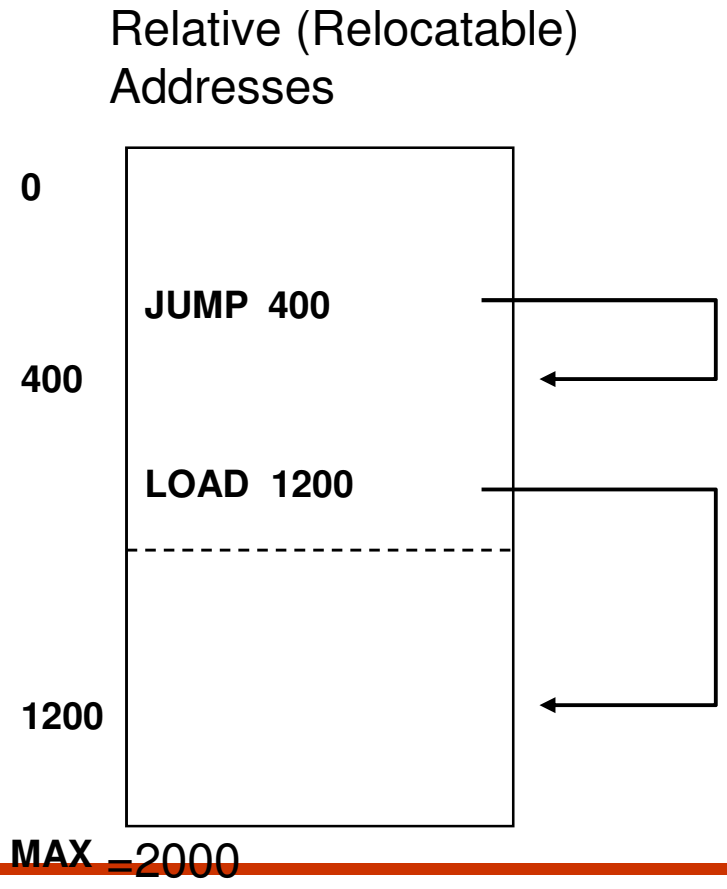


Code and Data are loaded into memory at address 10000 and 20000 respectively

Every unresolved address must be adjusted

Address Binding (Cont.)

- **Execution time: Binding delayed until run time**
  - **The process can be moved during its execution from one memory segment to another**
  - **The CPU generates the relative (virtual) addresses**
  - **Need hardware support for address maps (e.g., base and limit registers)**
  - **Most general-purpose OS use this method**
    - **Swapping, Paging, Segmentation**

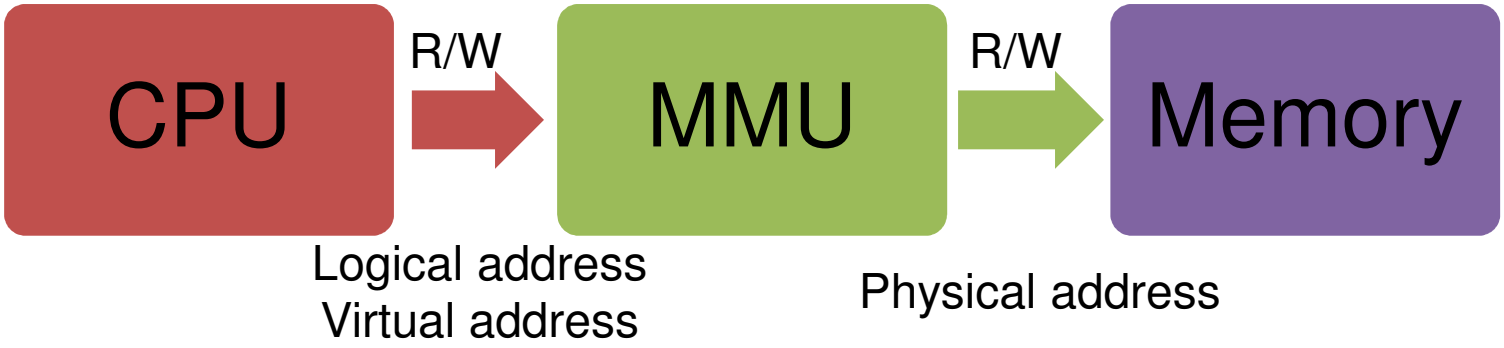


Need :

Dynamic relocation scheme with Automated Hardware System

MMU

Hardware Device



- ***Logical address*** – generated by the CPU; also referred to as *virtual address*
  - ***An integer value relative to the starting point of prog***
- ***Physical address*** – address seen by the memory unit
  - ***Add logical address with prog's starting address in main memory***

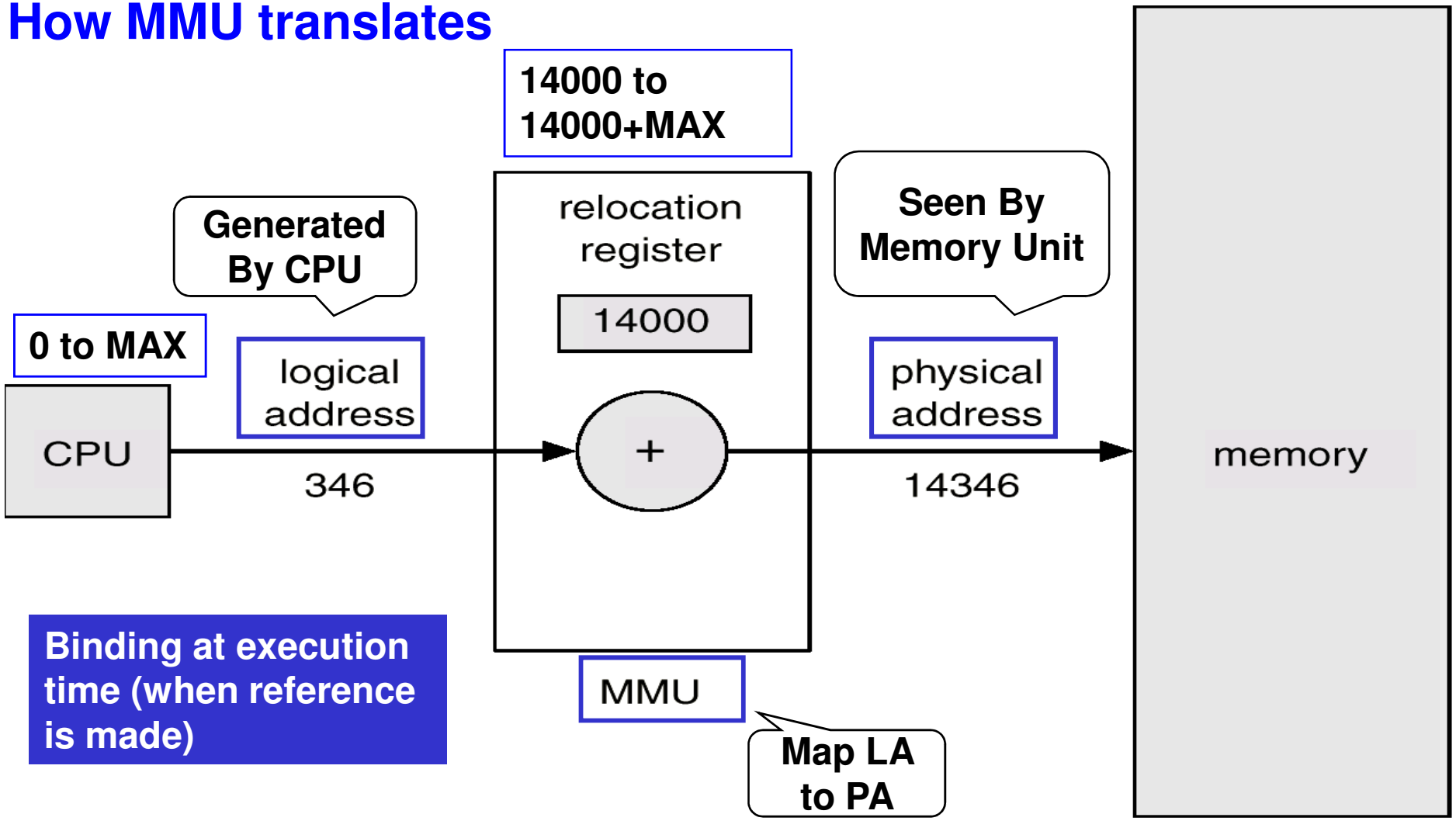
During **Compile and load time** address-binding methods generate identical logical & physical address.

However, **execution time** they differ.

WHY??-Answer Virtual Memory

Dynamic Relocation Using a Relocation Register

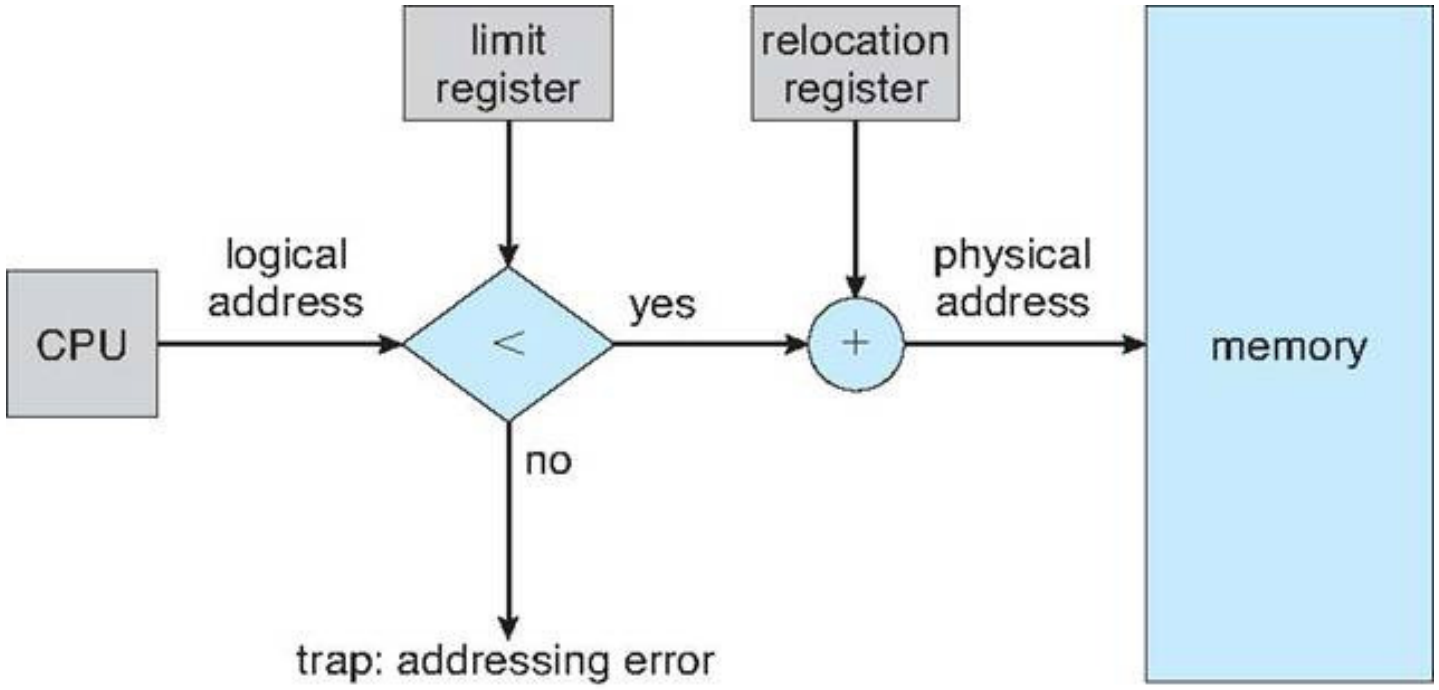
How MMU translates

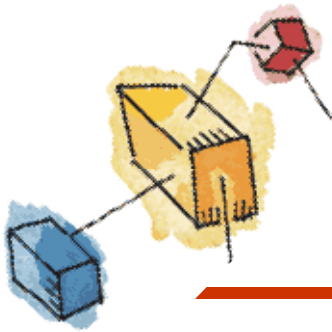


Binding at execution time (when reference is made)

Memory Protection

- **A test for limit can be added.**
- **Limit is the total amount of memory allocated to the process.**





Relocation

- Base register
- Bounds register

These values are set when the process is loaded or when the process is swapped in

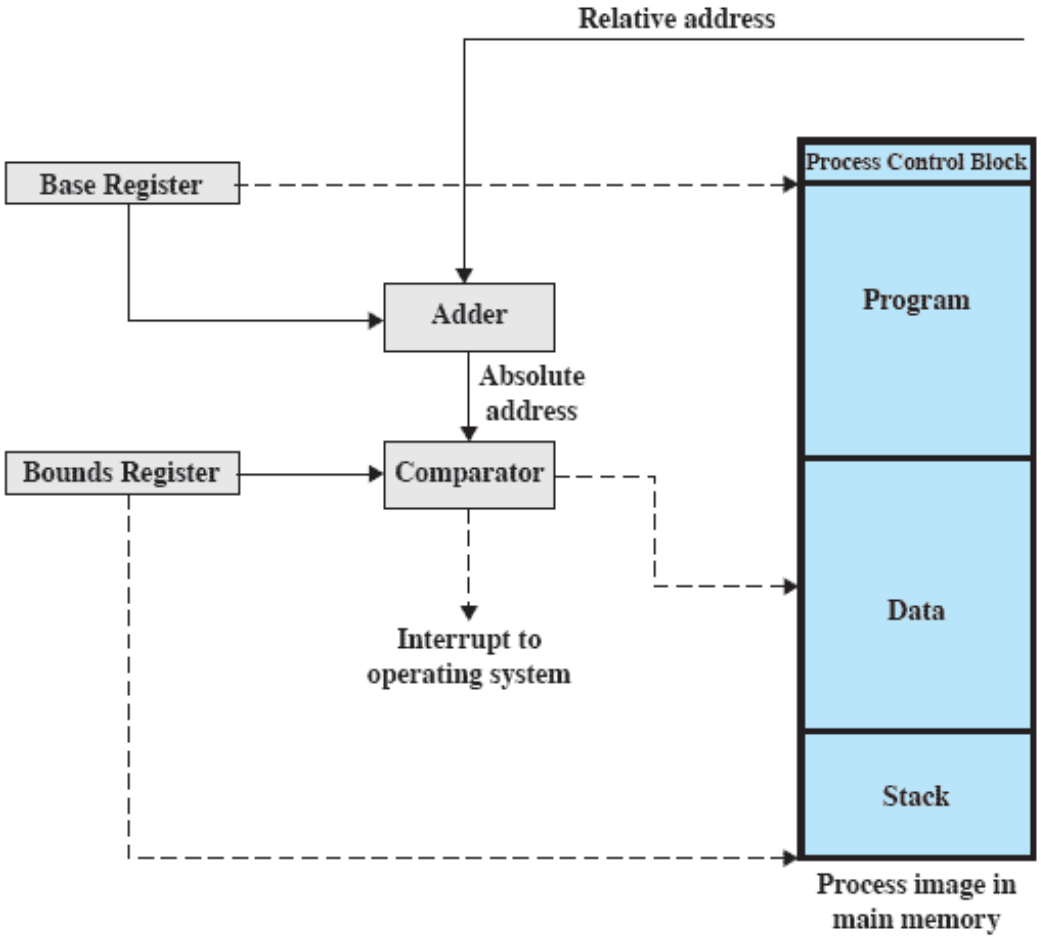
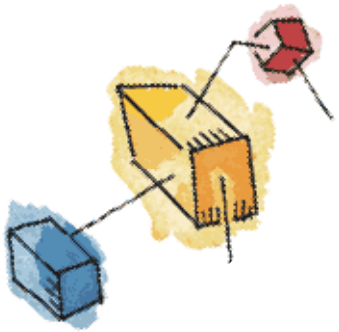


Figure 7.8 Hardware Support for Relocation

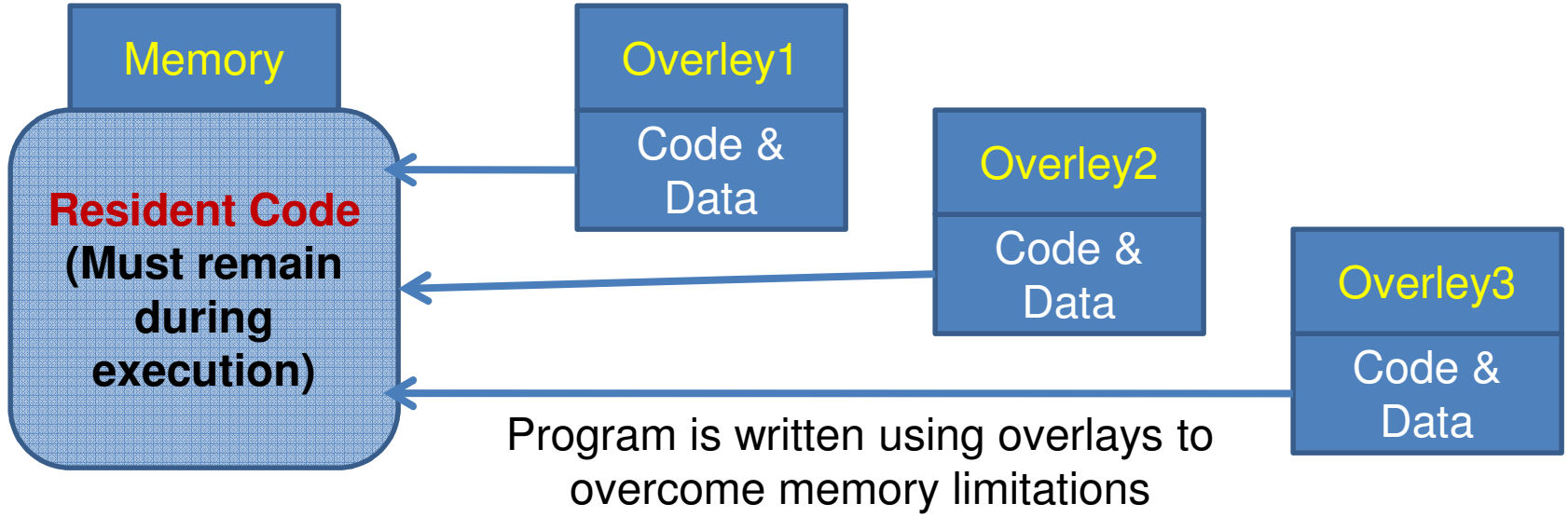


- Sofar,
  - during execution process must be in physical memory.
  - Process size is limited to memory size.

- What if the process is higher than allocated memory size??

Soln 1: Overlays

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



Disadvantages:

- programmer should design their programs to maintain overlay structure
- require complete knowledge of program structure, its code and data structure

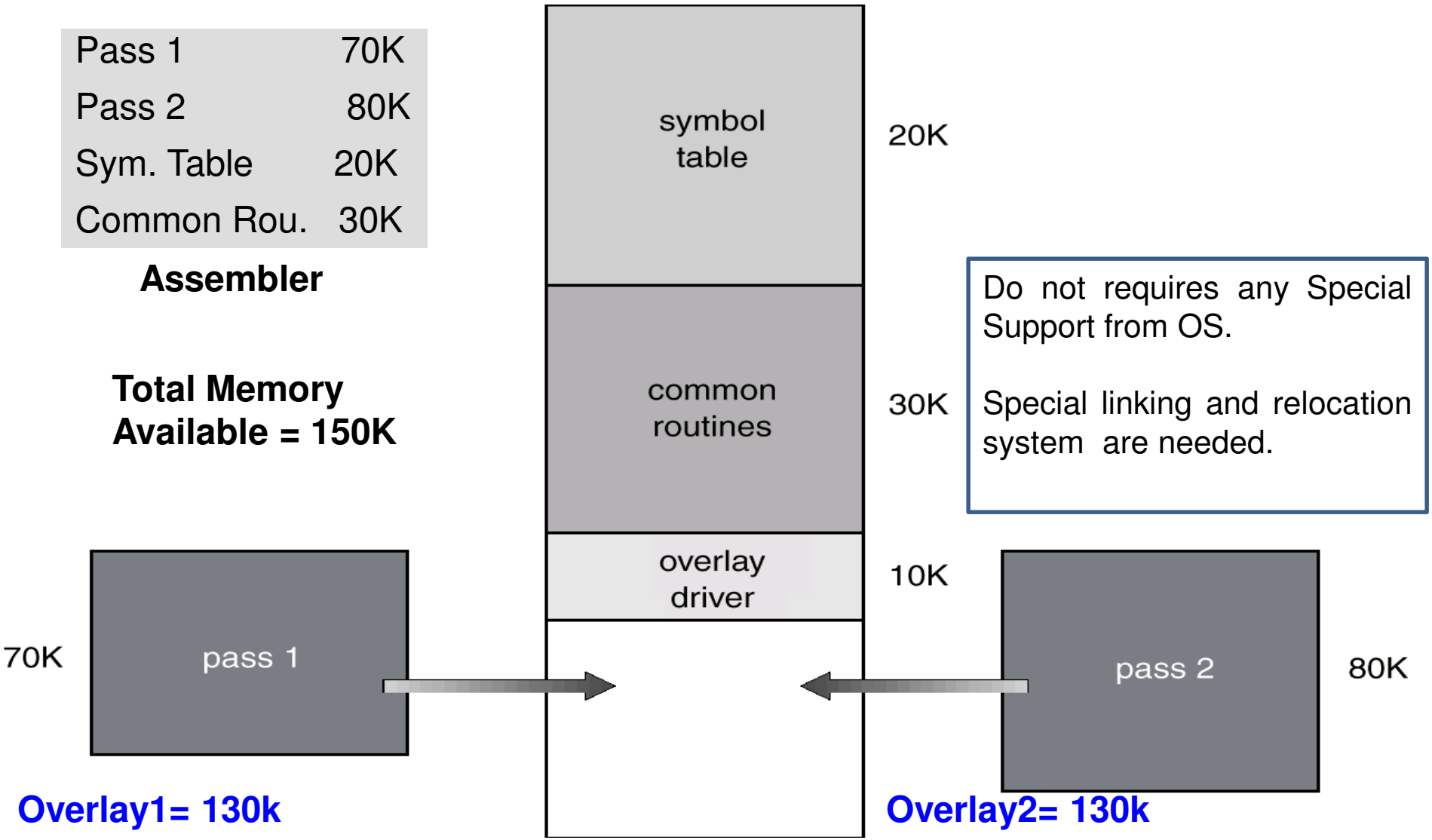
Example : Assembler required memory 200K

Overlay (Cont.)

Pass 1	70K
Pass 2	80K
Sym. Table	20K
Common Rou.	30K

Assembler

Total Memory Available = 150K



Do not requires any Special Support from OS.

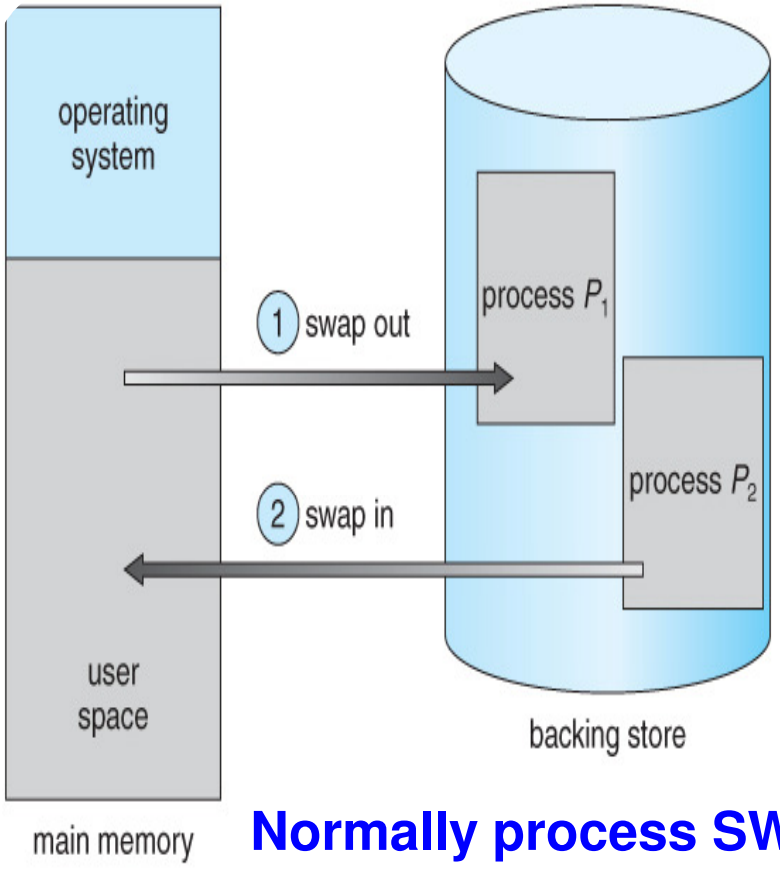
Special linking and relocation system are needed.

Overlay1= 130k

Overlay2= 130k

Soln 2: SWAP

Other Solutions:
Paging, Segmentation, Virtual memory
will explain later



Process can be swapped temporarily out of memory to a backing store and then brought back into memory for continue execution.

RR CPU Scheduling Algo (Quantum expires).

Called Roll In-Roll Out in priority-based scheduling.

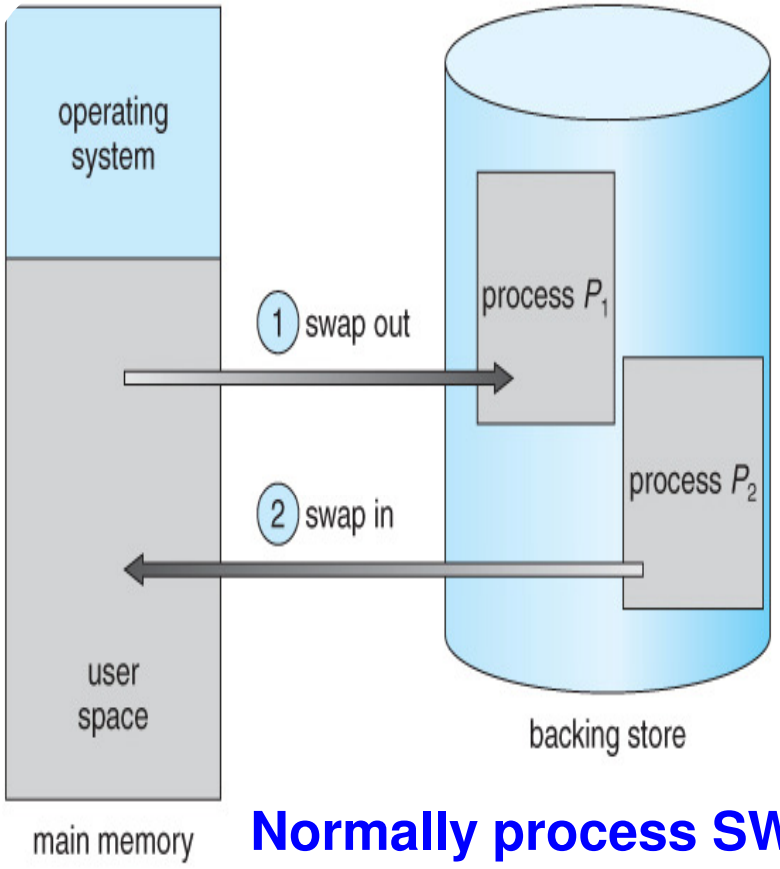
Normally process SWAP out will be swap back to same memory

- **Address Binding is done at assembly/load time**
  - **-Process can't be moved @ diffⁿ location**

- **Address Binding is done in execution time**
  - **-Swap process into diffⁿ location**

Soln 2: SWAP

Other Solutions:
Paging, Segmentation, Virtual memory
will explain later



Process can be swapped temporarily out of memory to a backing store and then brought back into memory for continue execution.

RR CPU Scheduling Algo (Quantum expires).

Called Roll In-Roll Out in priority-based scheduling.

Normally process SWAP out will be swap back to same memory

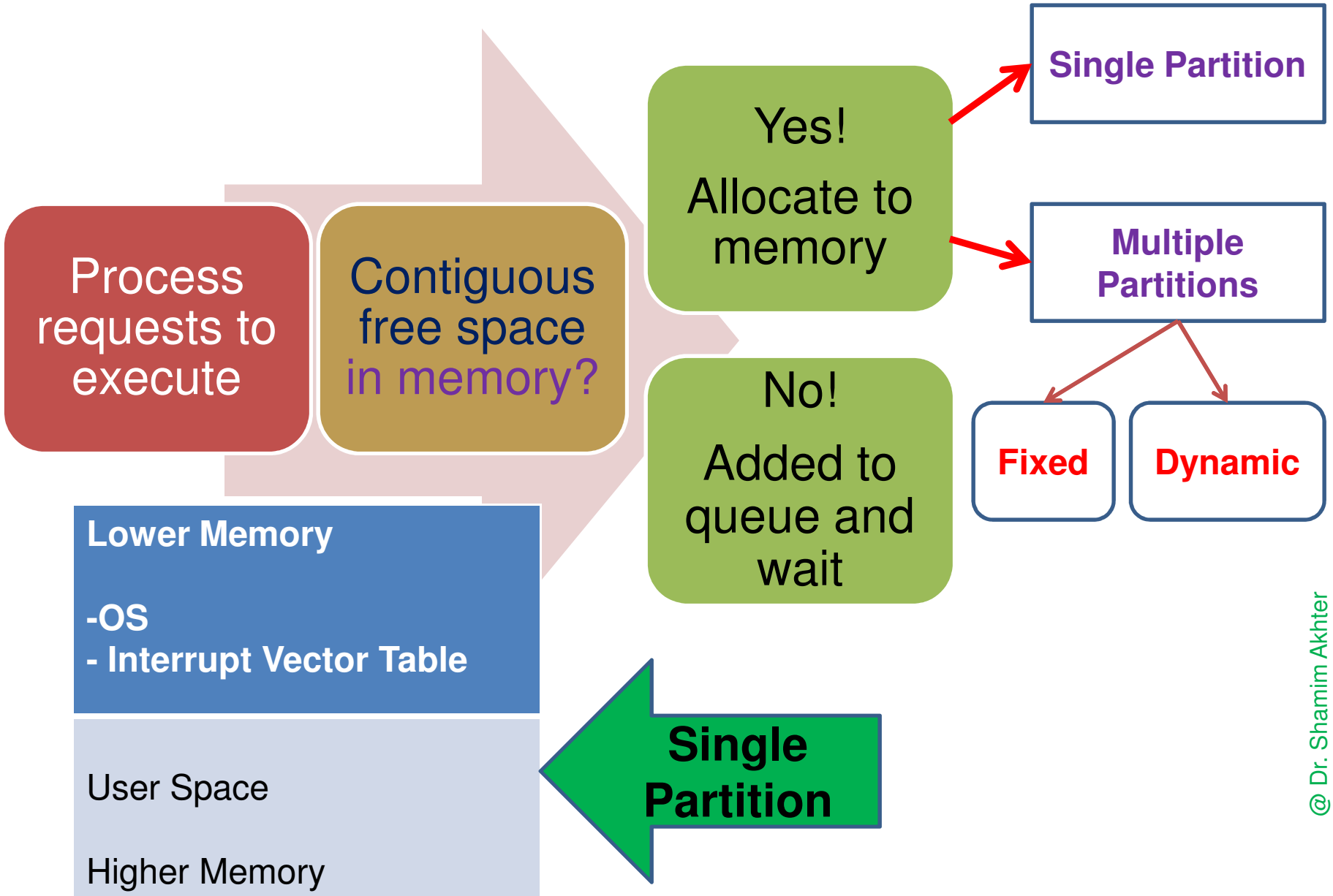
- **Address Binding is done at assembly/load time**
  - **-Process can't be moved @ diffⁿ location**

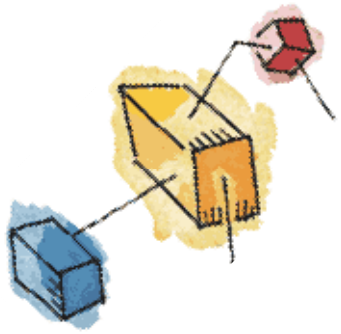
- **Address Binding is done in execution time**
  - **-Swap process into diffⁿ location**

Main memory consists
OS and various user processes.

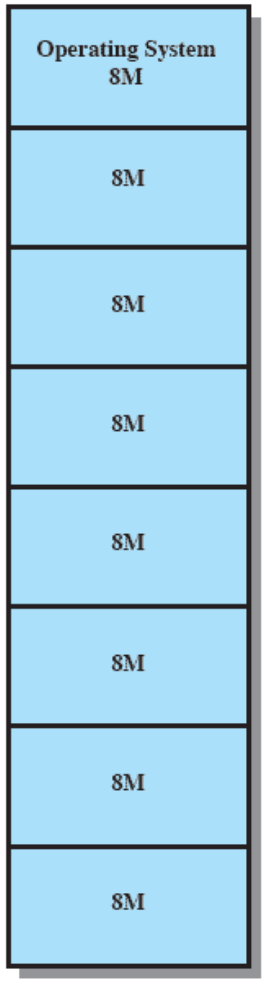
Need efficient way to allocate processes into memory.

Contiguous Memory Allocation

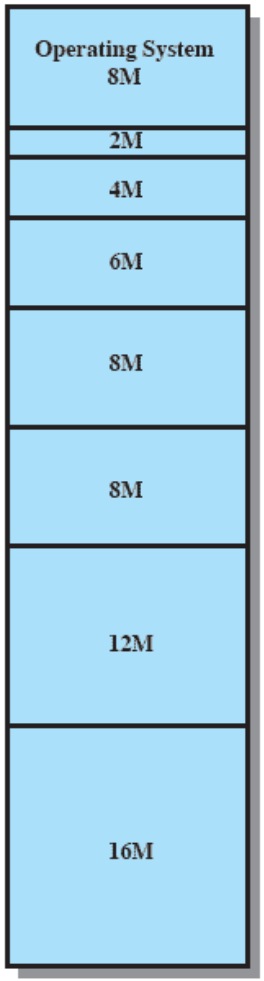




Multiple Partitions
Fixed Partitioning

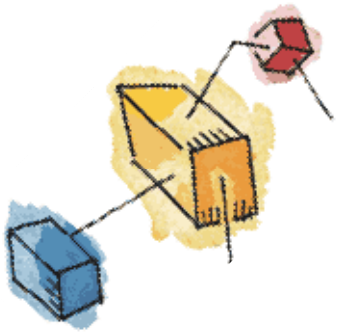


(a) Equal-size partitions

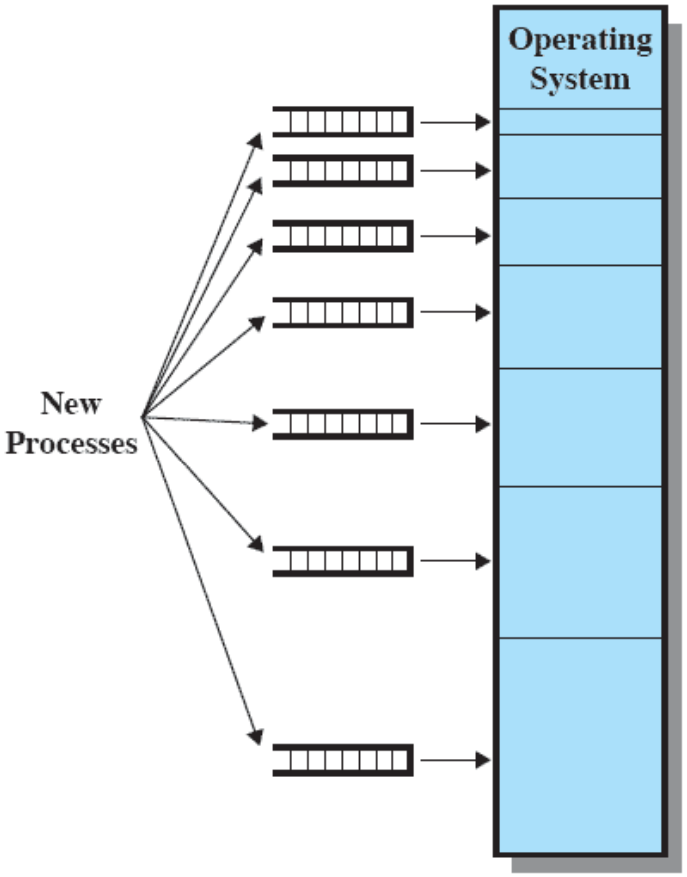


(b) Unequal-size partitions

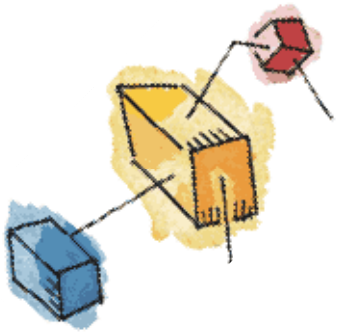
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory



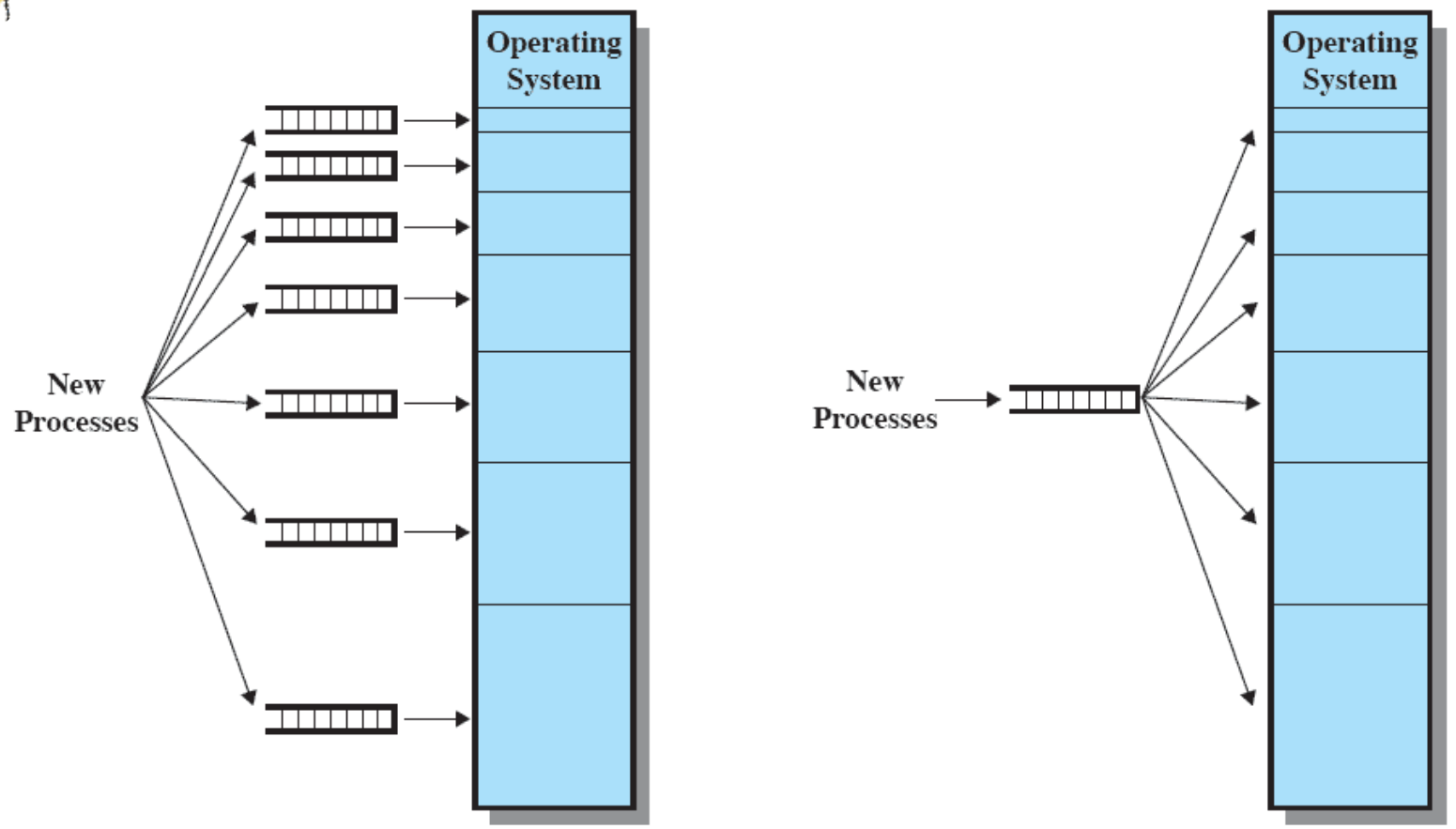
Fixed Partitioning



(a) One process queue per partition



Fixed Partitioning



(a) One process queue per partition

(b) Single queue

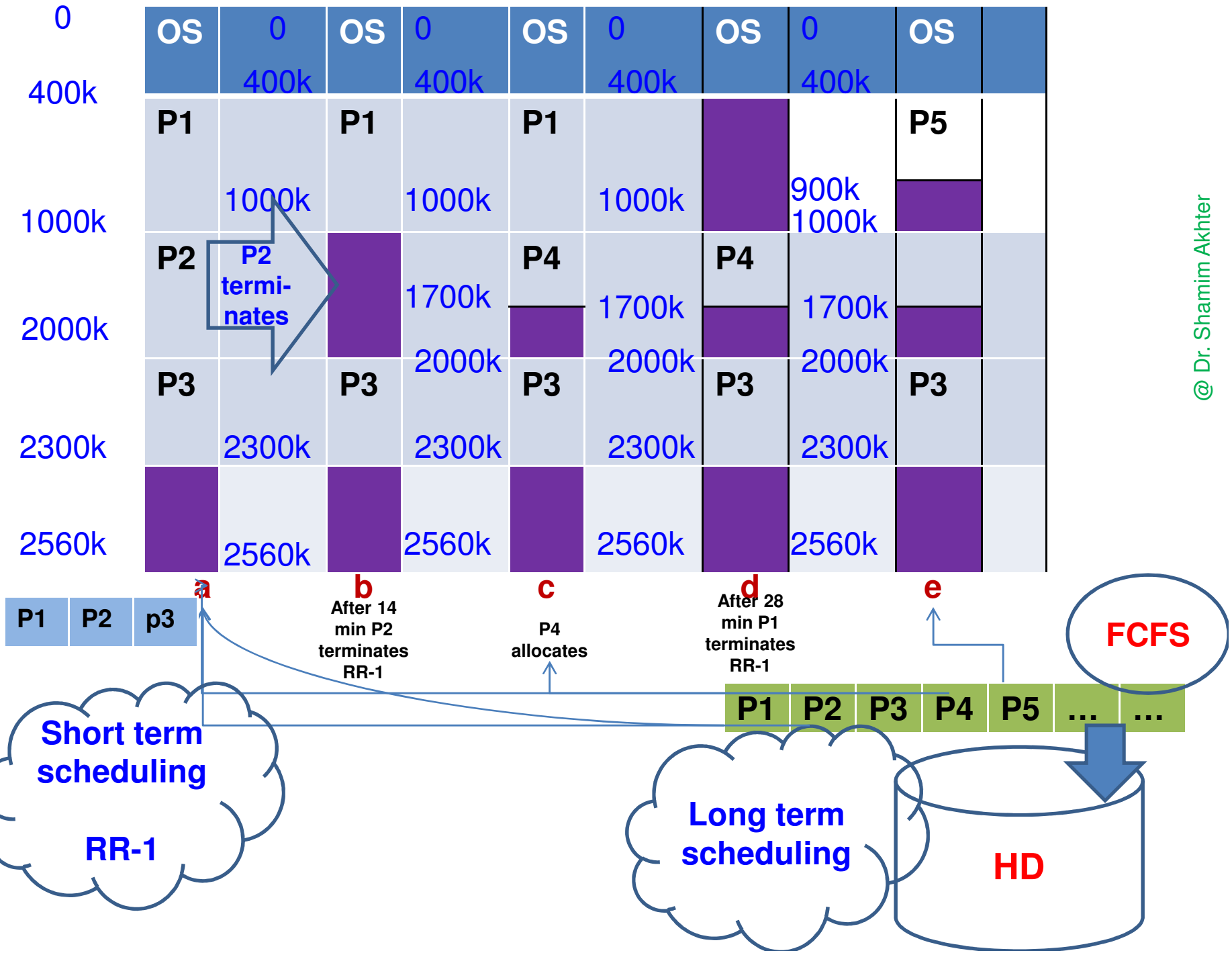
Figure 7.3 Memory Assignment for Fixed Partitioning

Example: Multiple Partitions Dynamic Partitions

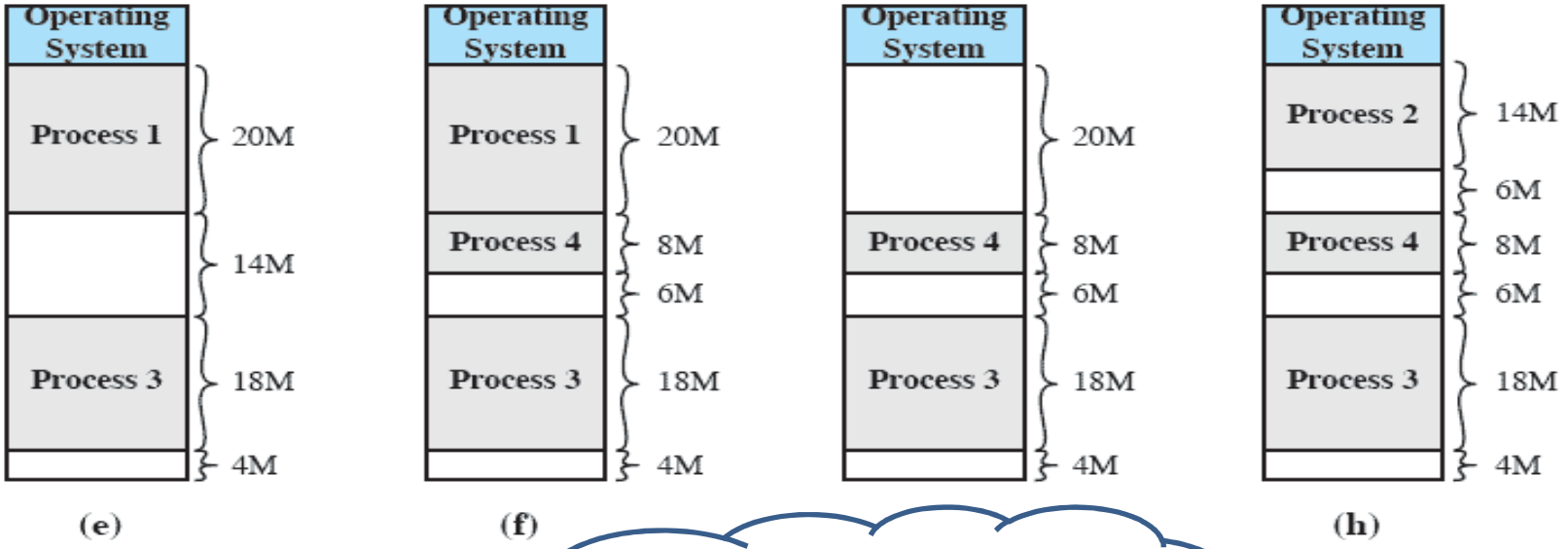
Job Queue

Process	Memory	Burst Time
P1	600k	10
P2	1000k	5
P3	300k	20
P4	700k	8
P5	500k	15

FCFS Scheduling Algorithm


0
400k
1000k
2000k
2300k
2560k
OS
0
400k
P1
P2
P3
P4
P5
1000k
2000k
2300k
2560k
900k
1000k
1700k
P2 termi-nates
a
b
c
d
e
After 14 min P2 terminates RR-1
P4 allocates
After 28 min P1 terminates RR-1
P1
P2
p3
P1
P2
P3
P4
P5
…
…
FCFS
Short term scheduling
RR-1
Long term scheduling
HD
@ Dr. Shamim Akhter

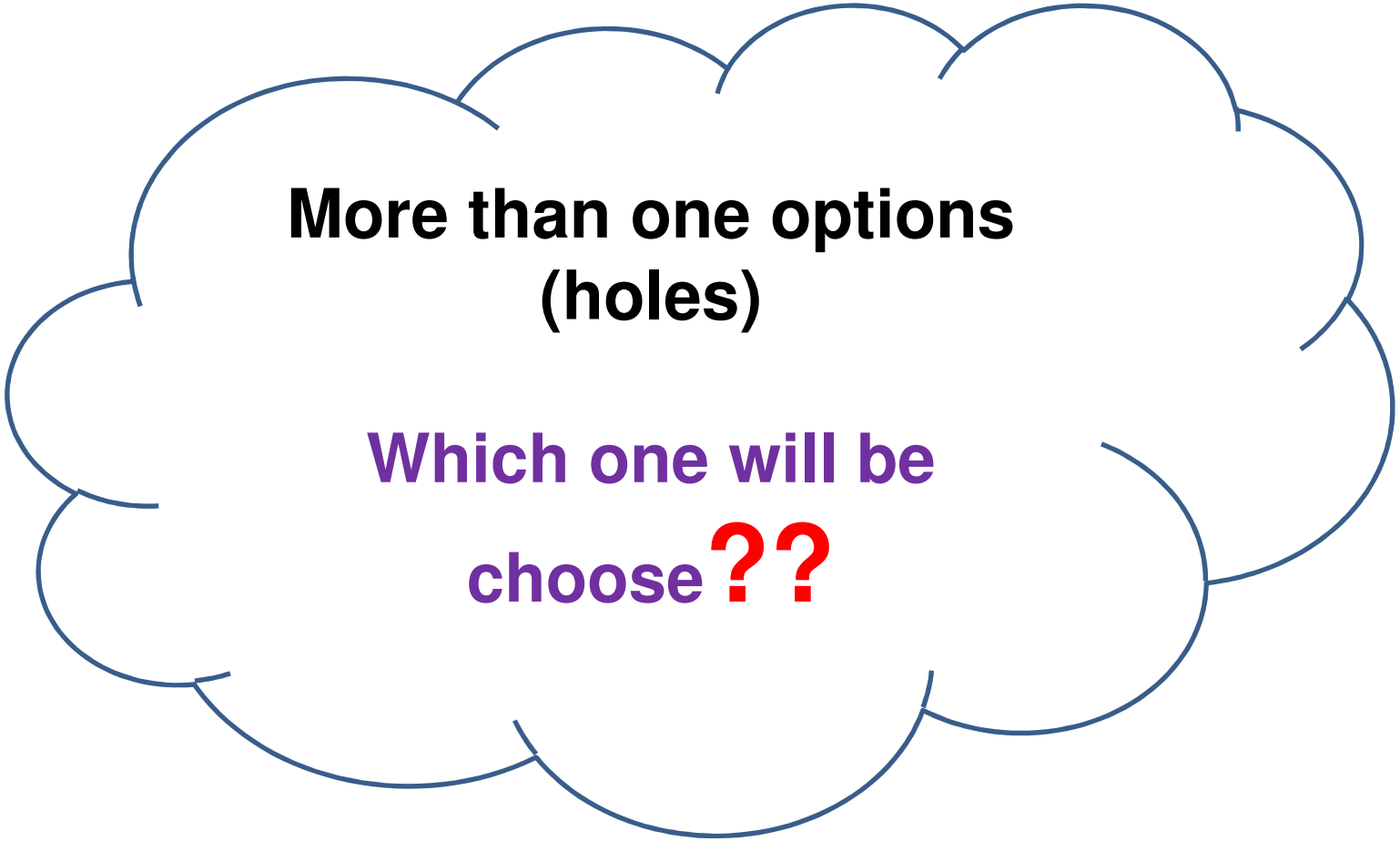
Effects of Dynamic Partitioning

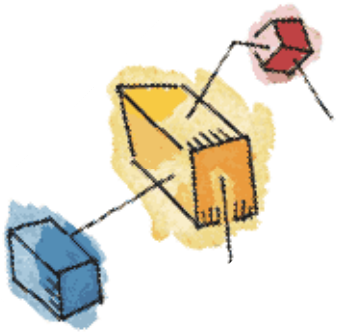


Compaction
Merged two holes while they nearby

Figure 7.4 Th tioning

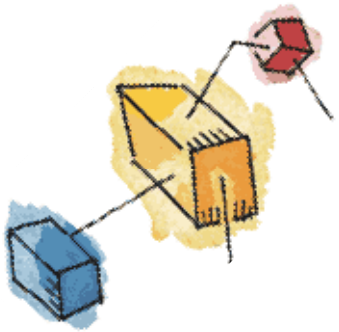
Must use compaction to shift processes so they are contiguous and all free memory is in one block





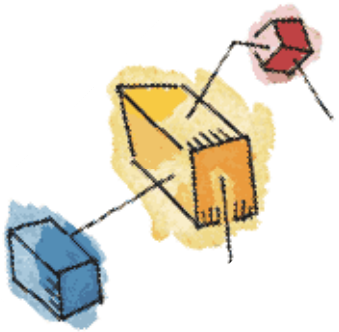
Dynamic Partitioning

- OS must decide which free block to allocate
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since the smallest block is found for process, the fragmentation of the smallest size is left
  - Memory compaction must be done more often



Dynamic Partitioning

- First-fit algorithm
  - Scans memory
    - form the beginning OR form the previous first fit
    - and chooses the first available block that is large enough
  - Fastest
  - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block



Dynamic Partitioning

- Worst-fit
  - Scans memory for the largest hole.
  - Produces largest leftover hole.

- Simulation reflects:
  - First fit and best fit are better(time, storage) than worst fit

Allocation Schemes

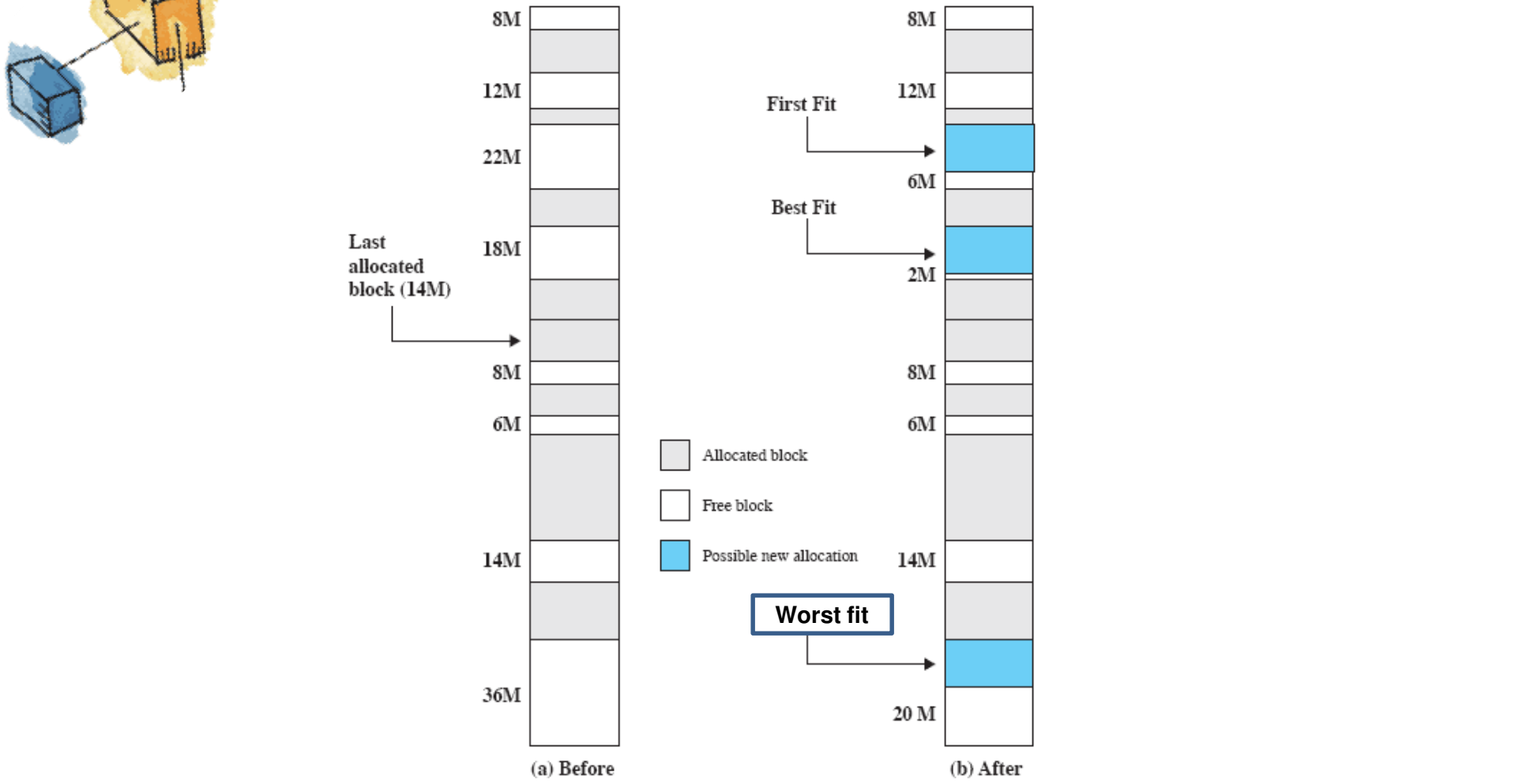
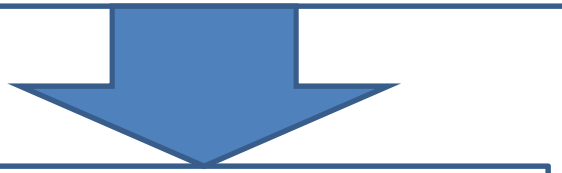
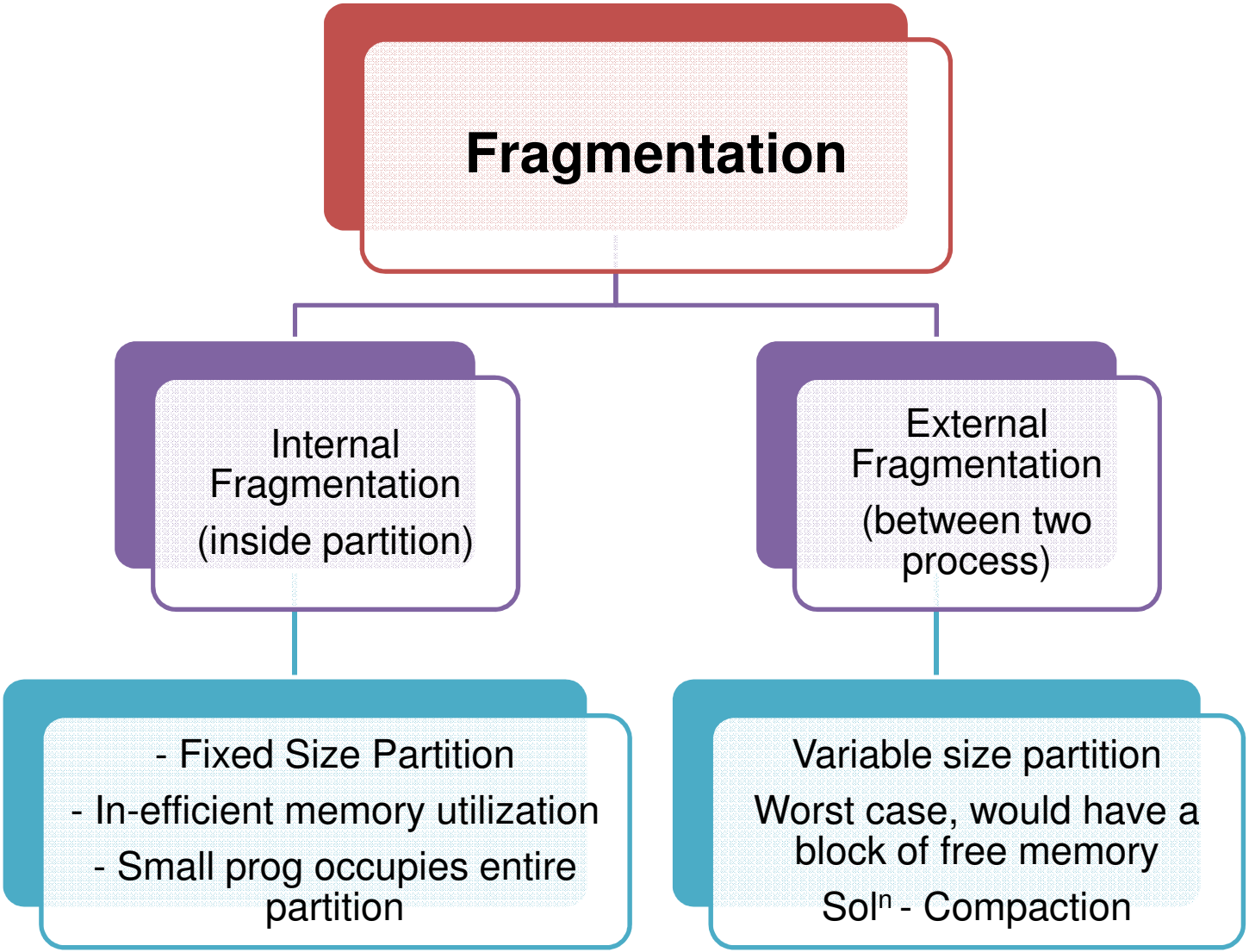


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

As processes are loaded & removed from memory, the free space is broken into little pieces.



FRAGMENTATION



If relocation is static and is done at assembly/load time-> compaction can't be done

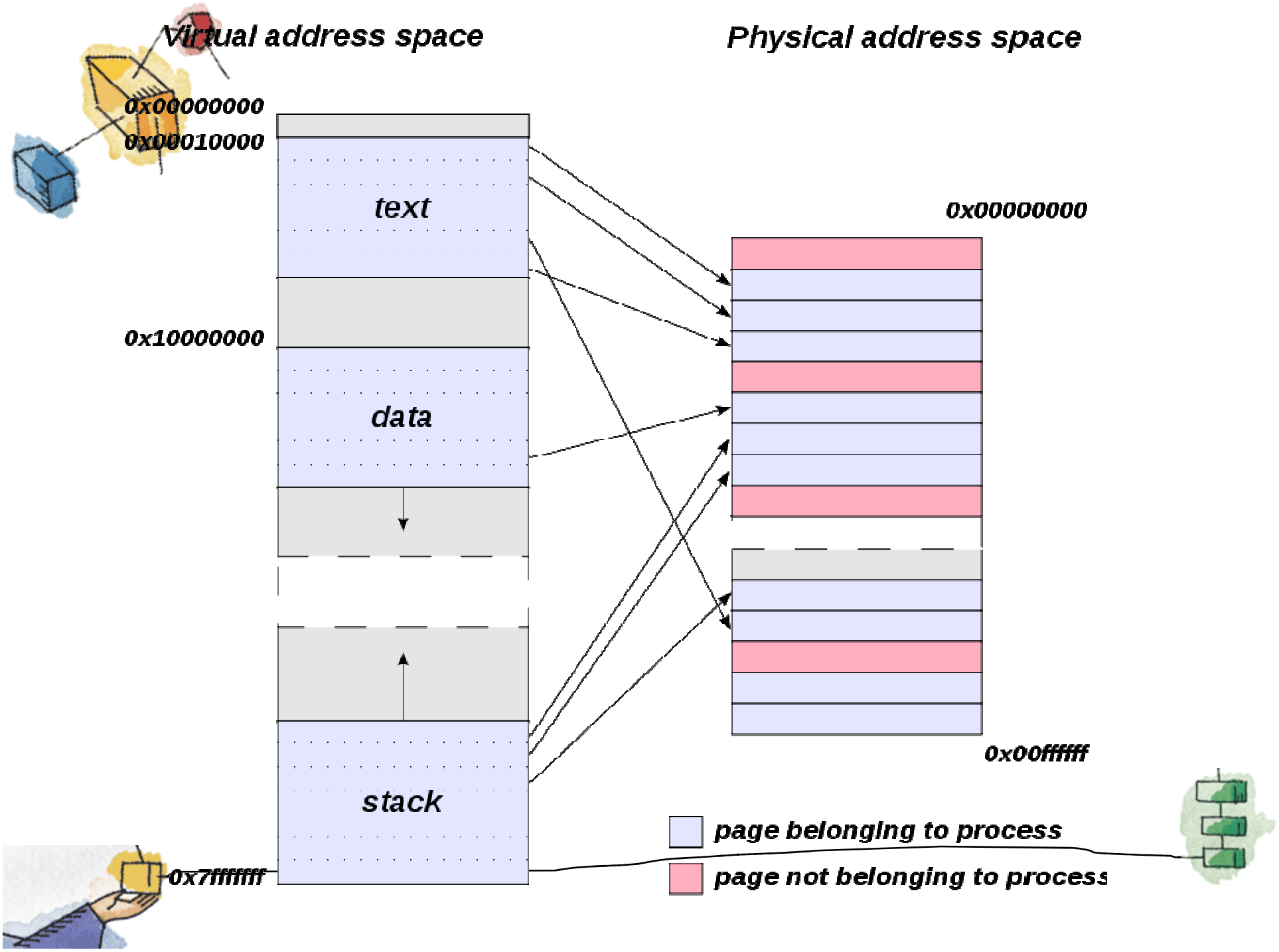
If relocation is dynamic and is done at execution time-> compaction possible

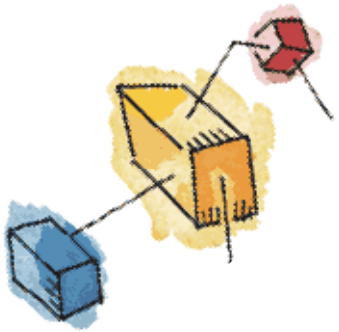
External Fragmentation Solution

Compaction is Costly

Selecting Optimal compaction strategy is difficult.

Solution ??

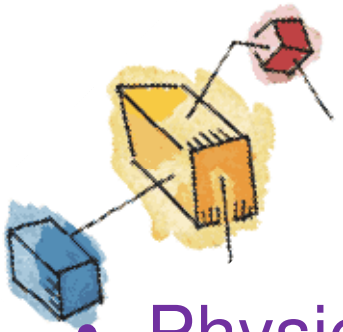

Virtual address space
Physical address space
0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff
0x00000000
0x00ffffff
page belonging to process
page not belonging to process



Sol^{n}3 : Paging

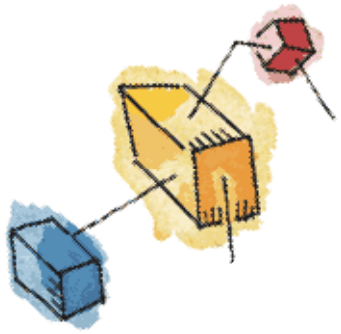
- Sol^n: for external fragmentation problem
  - REDUCE to waste memory space

- Permits logical address of space of a process to be non-contiguous.



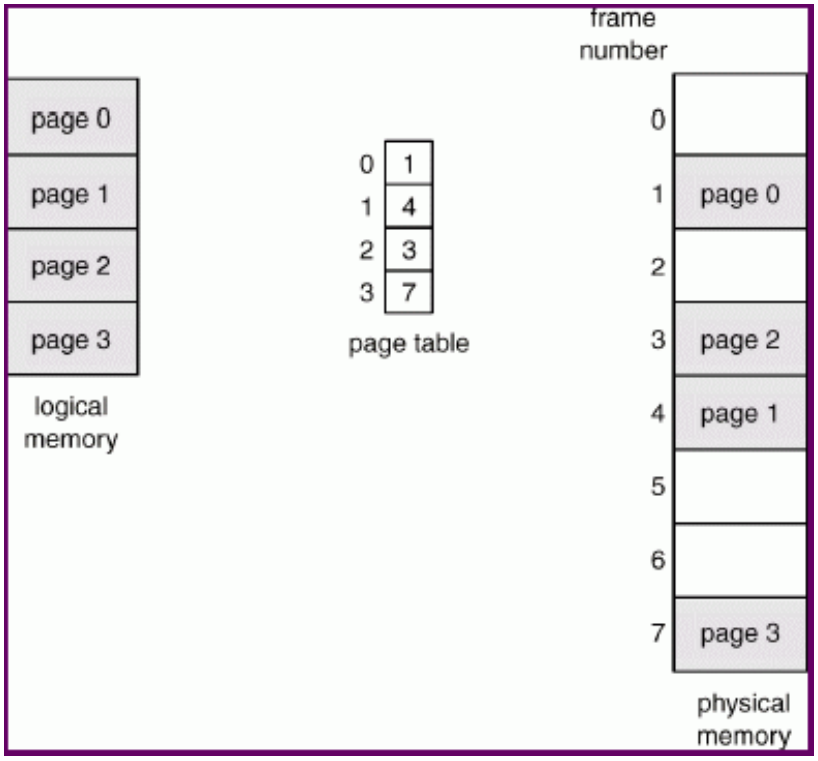
Paging Schemes

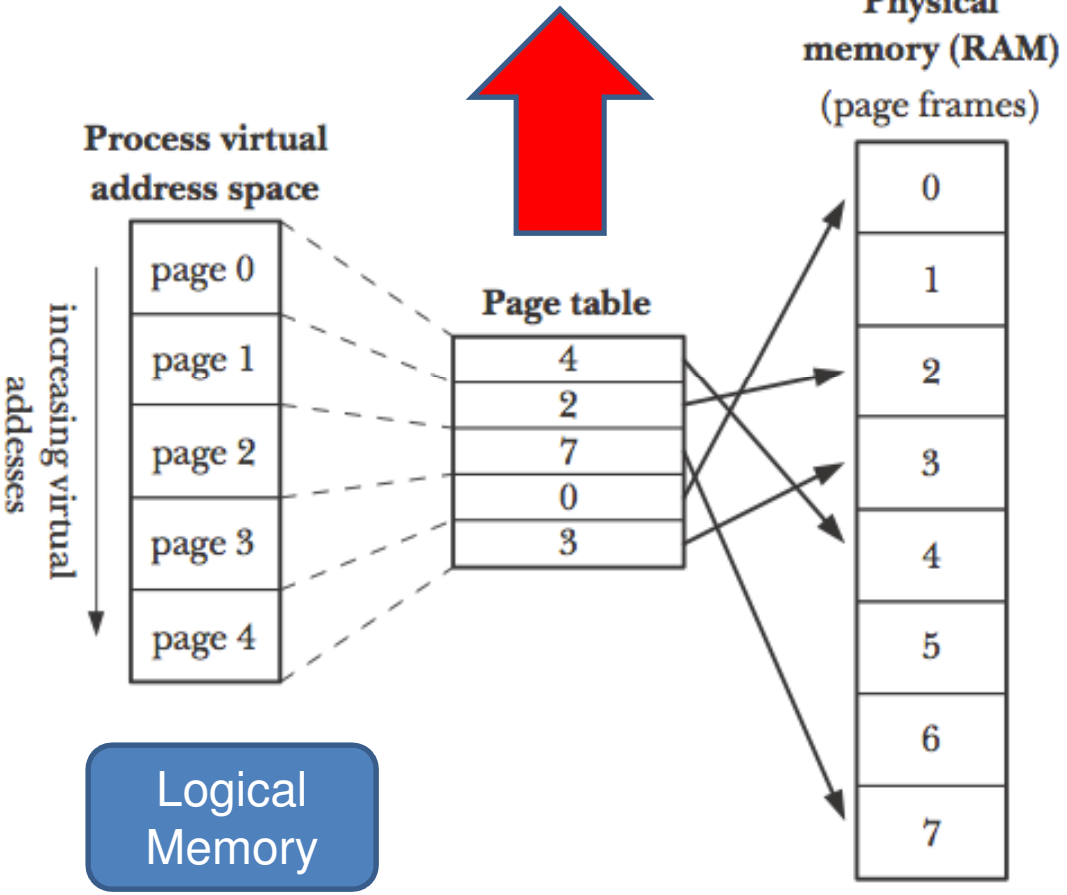
- Physical memory
  - is broken into fixed-sized blocks called frames.
- Logical memory
  - is also broken into blocks of same sized called pages.
  - Page size define by the hardware. Typically 512 – 8192 bytes (power of 2)
  - **getconf PAGE_SIZE** UNIX command to get page size
- Operating system maintains
  - a page table for each process



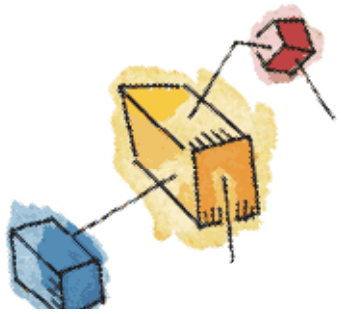
Paging- dynamic relocation

Base (relocation register)
One for each frame

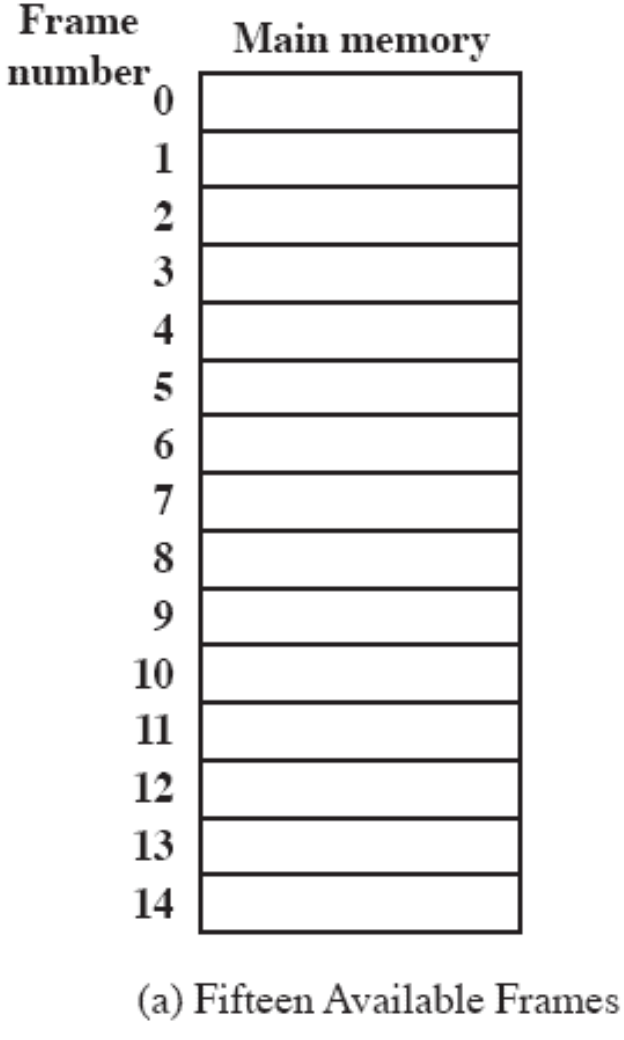




Logical Memory



Process and Frames



(a) Fifteen Available Frames

(b) Load Process A

(c) Load Process B

Process and Frames

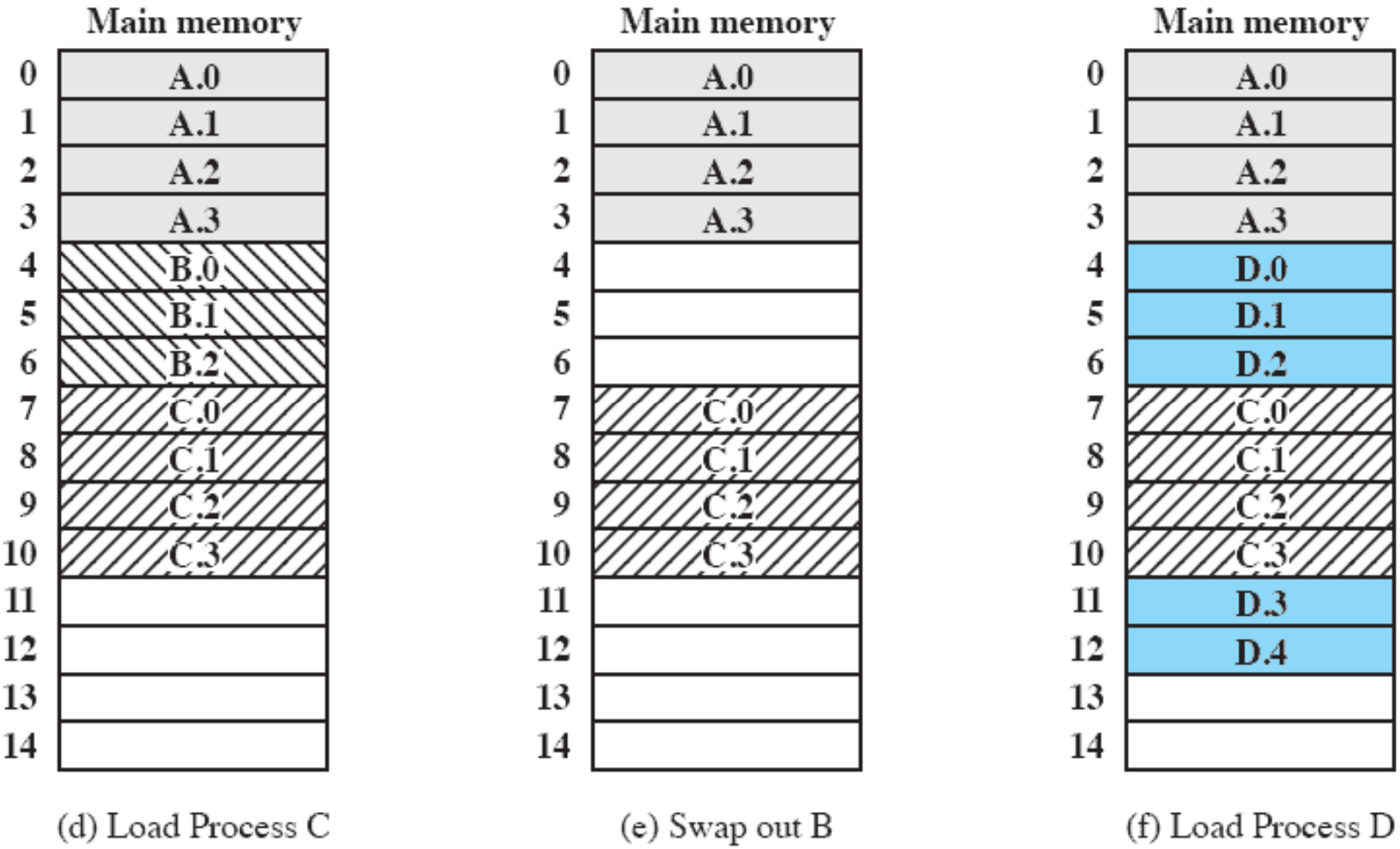


Figure 7.9 Assignment of Process Pages to Free Frames

Logical Addresses

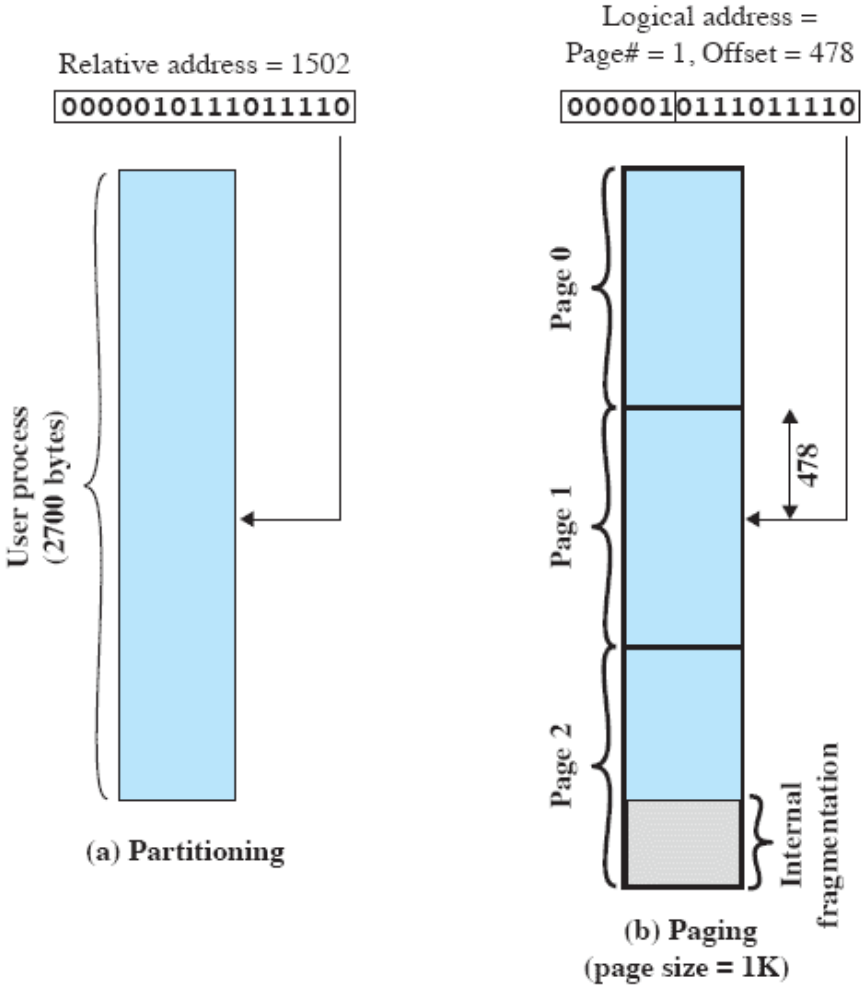
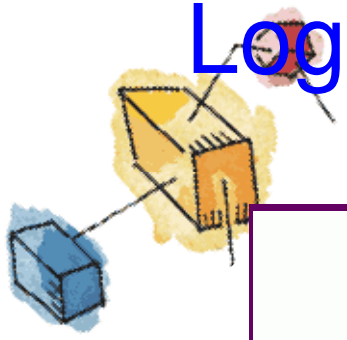


Figure 7.11 Logical Addresses

Logical address to Physical Address
Direct Mapping



Page Size = 4
Frame Size = 4

CPU

logical address

physical address

p | d

f | d

Page Offset
Displacement within the page

1 | 3

0
1

3

3 | 3

f

page table

0	0	
1	1	
2	2	
3	3	
4	0	A
5	1	B
6	2	C
7	3	D

Page 0

Page 1

0	0	
1	1	
2	2	
3	3	
4	0	
5	1	
6	2	
7	3	
8	0	
9	1	
10	2	
11	3	
12	0	A
13	1	B
14	2	C
15	3	D

Frame 0

Frame 1

Frame 2

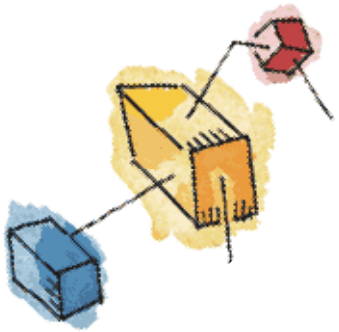
Frame 3

f

p=7/4 | **d**=7-(p*4)

f=3*4 | **d**=3

Logical Address



0		Page 0	2^n
1			
2			
3		Page 1	2^n
4			
5			
6		Page 2	2^n
7			
…			
2^m		Page $2^m/2^n = 2^{m-n}$	2^n

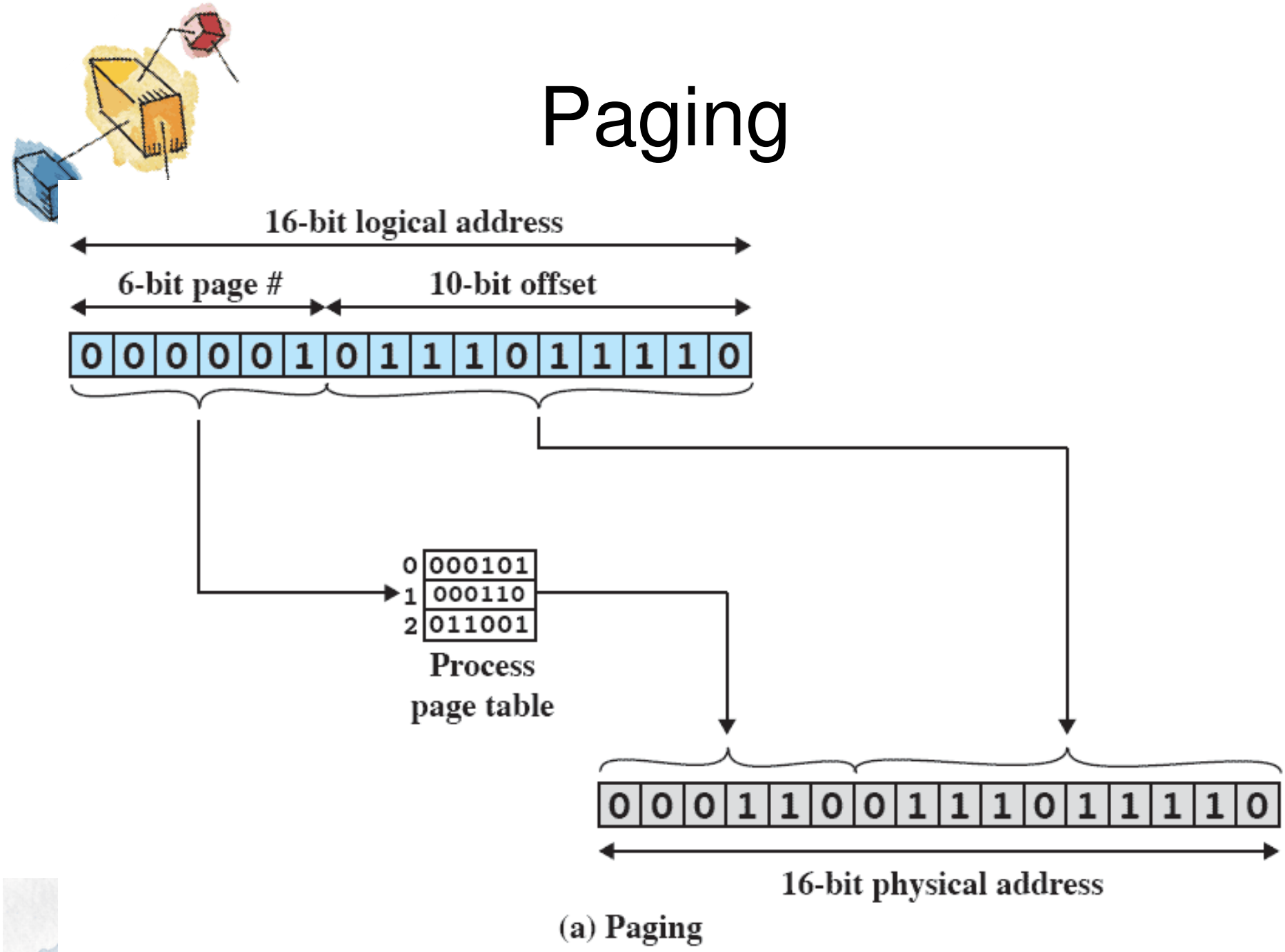
m bits

Page #	Page Offset
p	**D**
m-n bits	**n bits**

0	n-> 2^n
1	n-> 2^n
2	n-> 2^n
…	
2^{m-n}	n-> 2^n

Total Address = 2^m

Paging



(a) Paging

In-Class Exercise

10 pts. Consider a simple paging system with the following parameters: 2^{31} bytes of addressable physical memory; page size of 2^{10} bytes; 2^{26} bytes of logical address space.

(a) How many bits are in a logical address?
Solution: 26

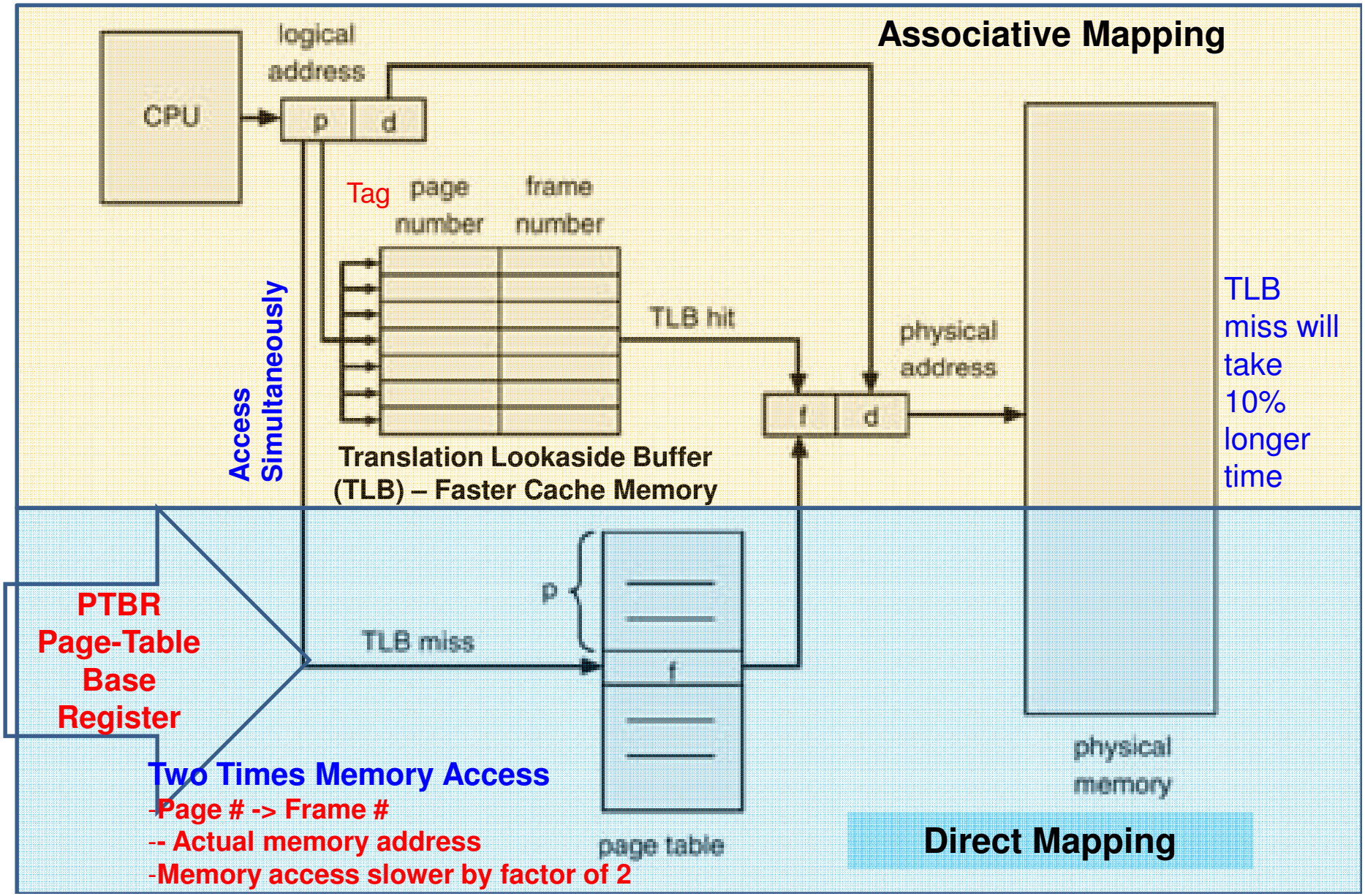
(b) How many bytes in a frame?
Solution: 2^{10}: same as the page size

(c) How many bits in the physical address specify the frame?
Solution: 21: 31 (entire address) - 10 (offset)

(d) How many entries in the page table?
Solution: 2^{16}: the number of pages

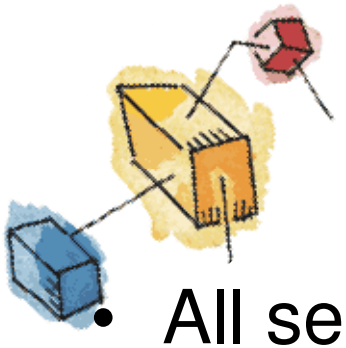
(e) How many bits in each page table entry (assume each page table entry includes a valid/invalid bit).
Solution: 22: 21 (page frame) + 1 (valid)

Paging with Associative & Direct Mapping



Internal Fragmentation

Solution ??



Soln4 : Segmentation

- All segments of all programs do not have to be of the same length
- There is a maximum segment length
- Addressing consist of two parts
  - a segment number and an offset
- Since segments are not equal, segmentation is similar to dynamic partitioning

Logical Addresses

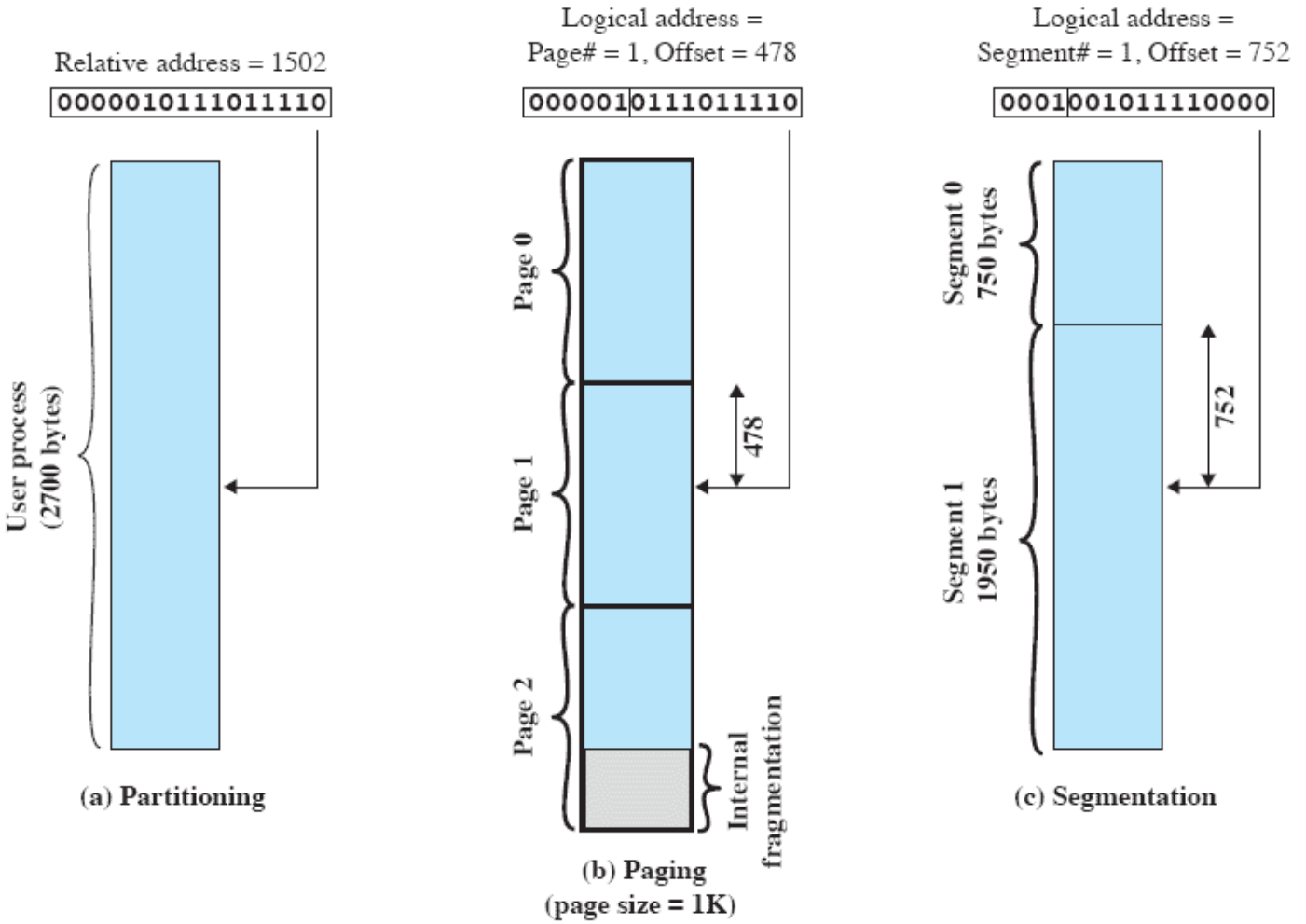
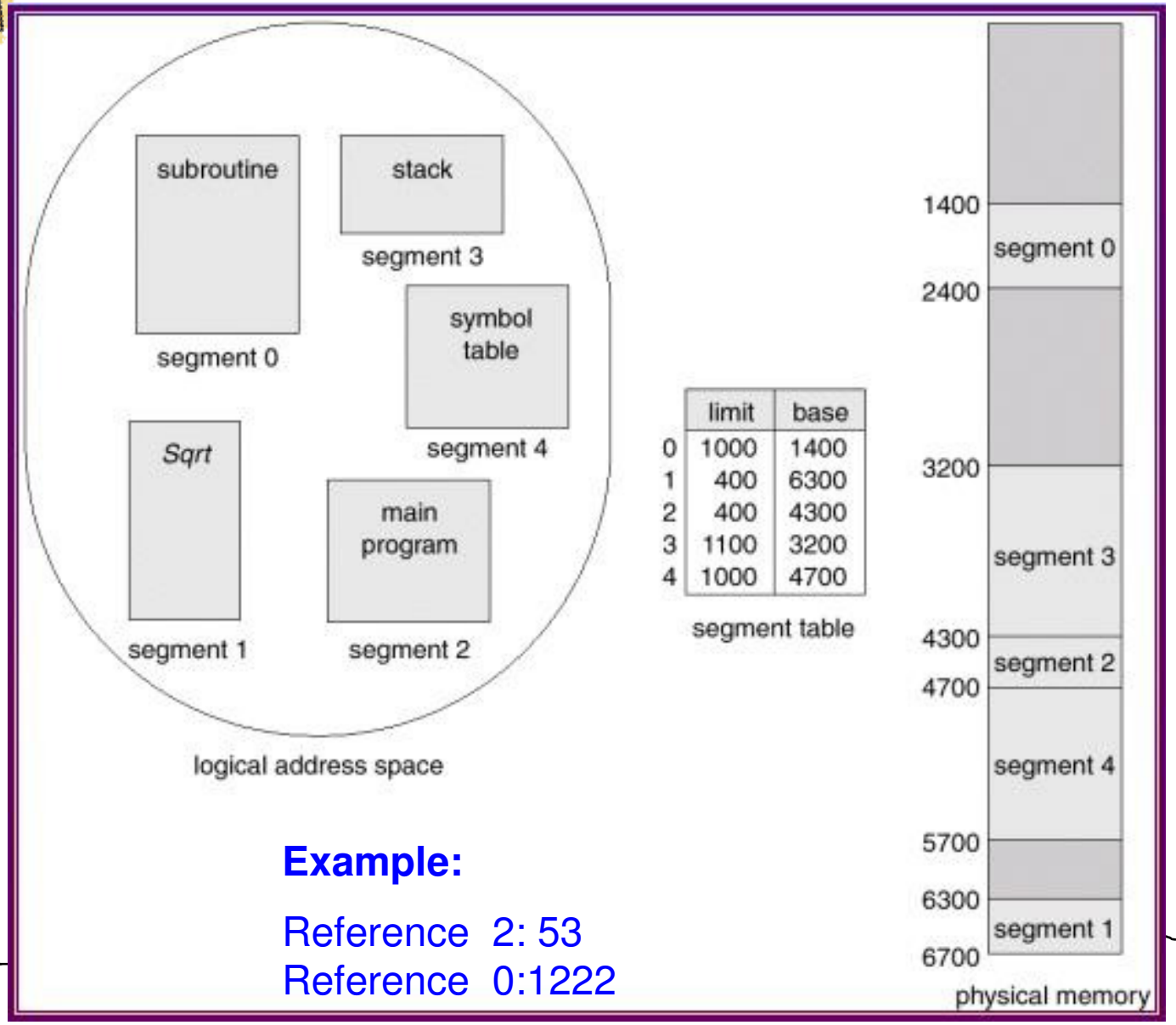


Figure 7.11 Logical Addresses

Example of Segmentation

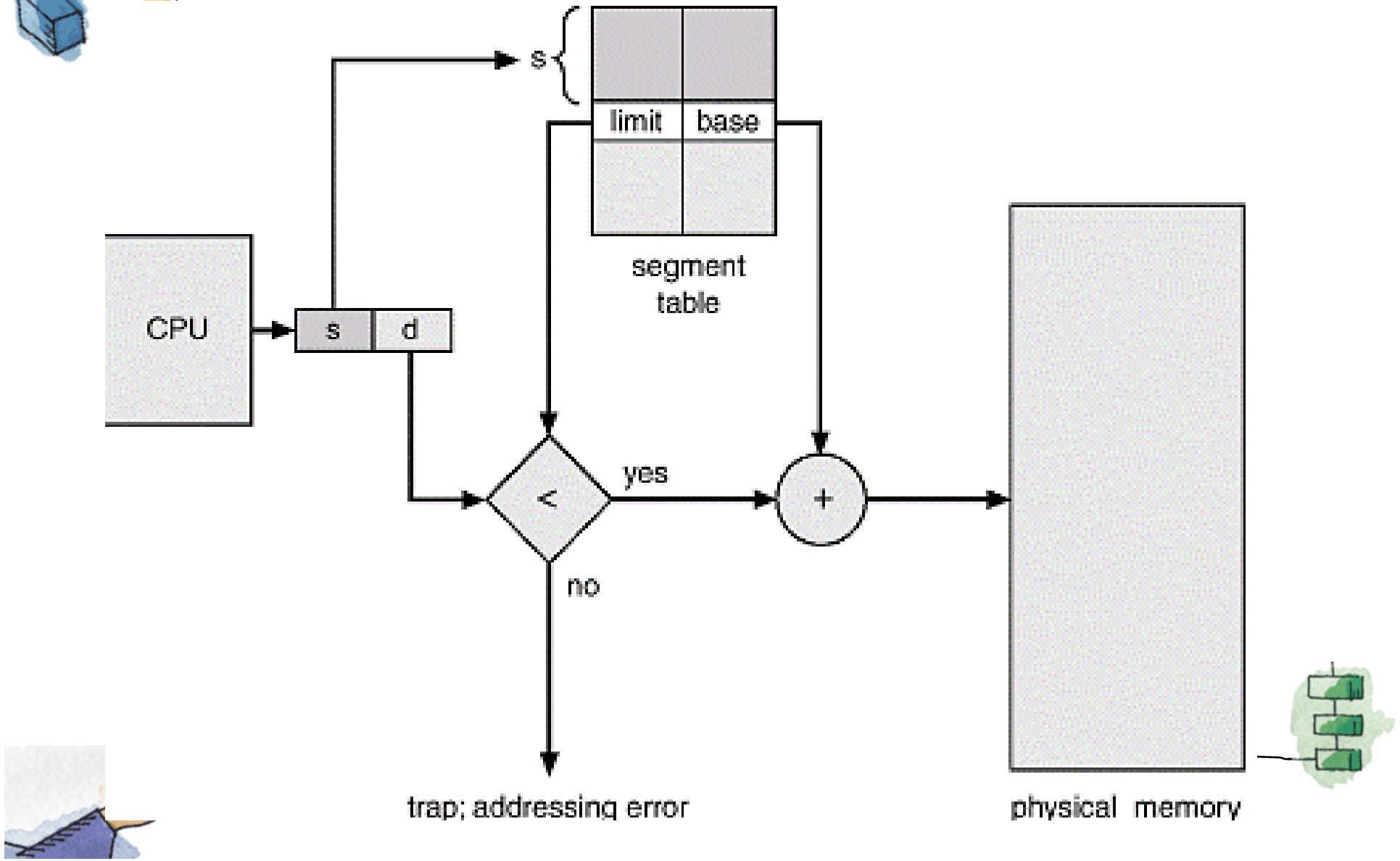


Example:

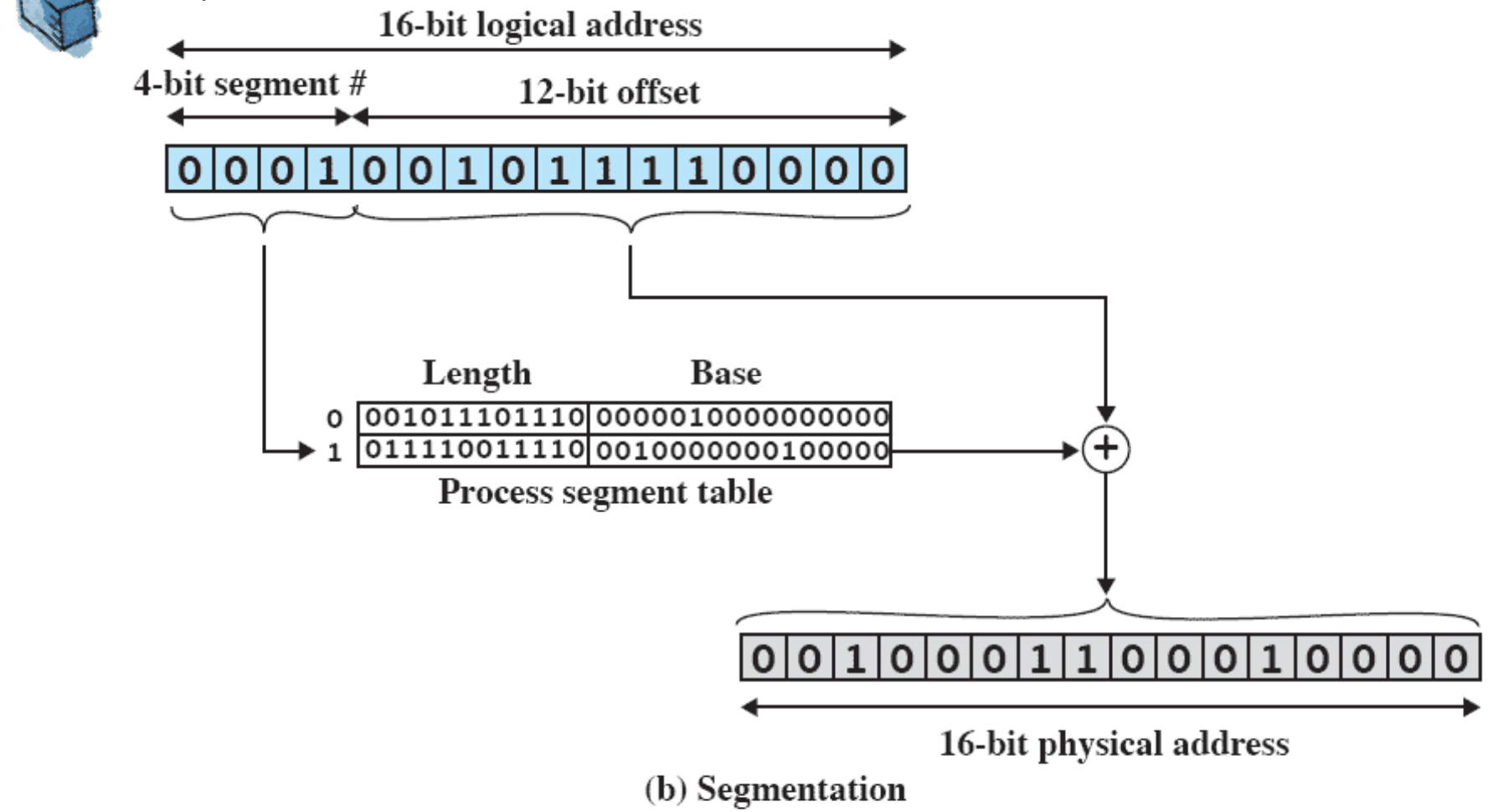
Reference 2: 53

Reference 0:1222

Segmentation Hardware

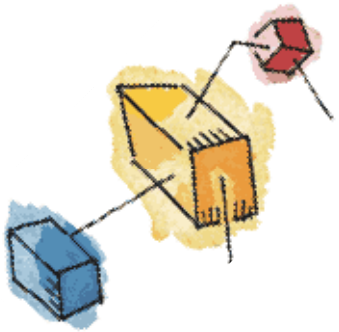


Segmentation



(b) Segmentation

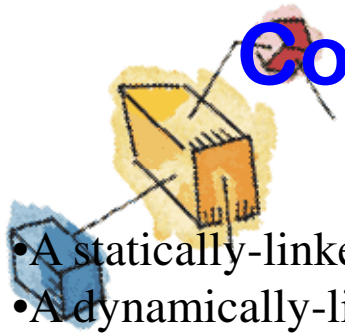
Figure 7.12 Examples of Logical-to-Physical Address Translation



Reading Materials

- Chapter 9:
  - 9.1, 9.2, 9.3, 9.4.1-9.4.3,9.5.1,9.5.2

Compile, assemble, link, load, run



- A statically-linked load module does not require any further linking.
- A dynamically-linked load module may require further linking at run time.

input	suffix	tools	output	time
source program	.c .cpp	compiler	assembly code symbolic addresses	compile time
assembly code	.s	assembler	object module relative addresses	assembly time
object modules	.o	linkage editor (linker)	load module relative addresses	link time
load module, system libraries	.exe .a	static loader	memory segments absolute addresses	load time
load module, system libraries	.exe .so .dll	dynamic linker, dynamic loader	memory segments absolute	execution time, run time