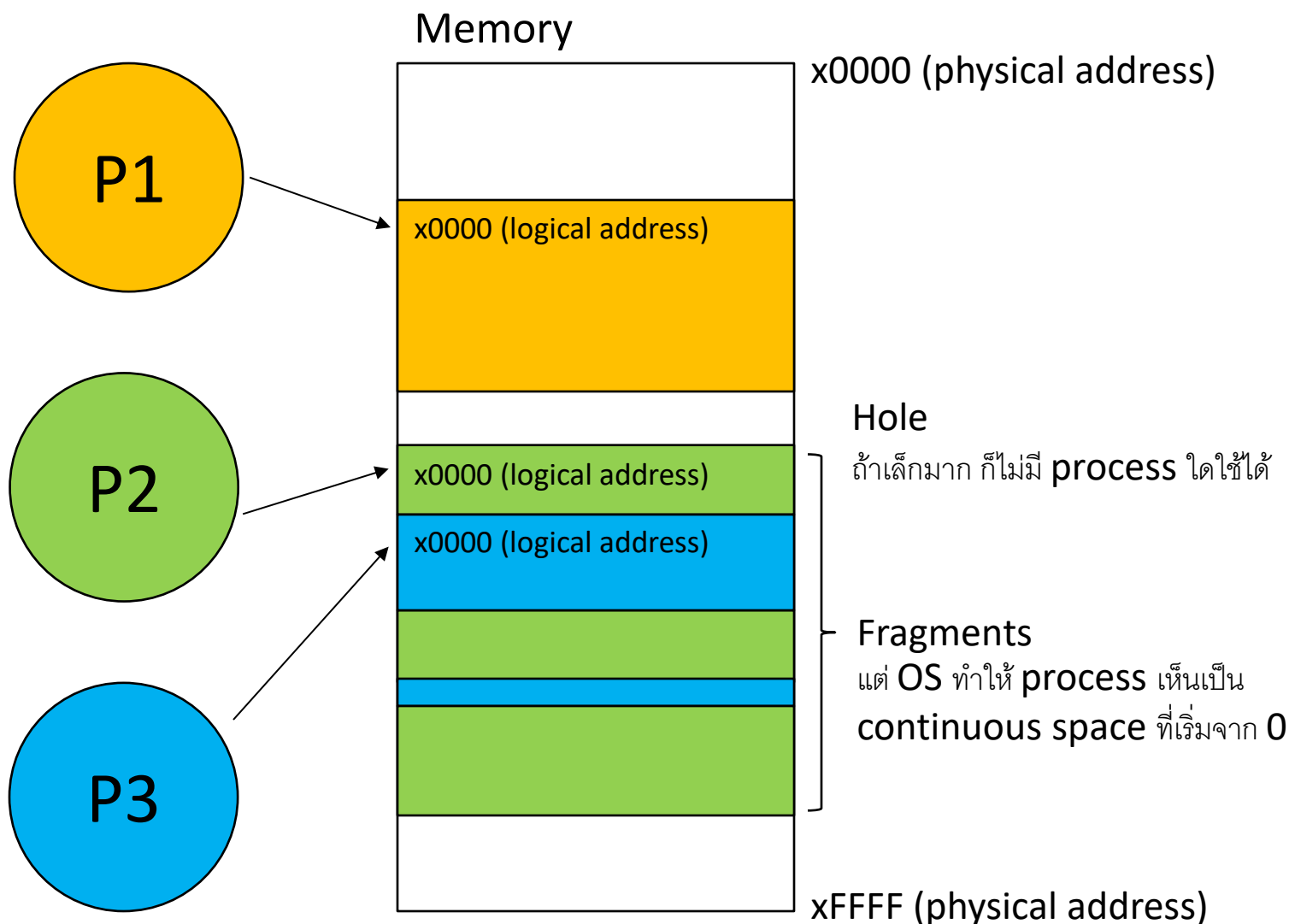


Memory Management



Memory Separation

แบ่ง memory ให้แต่ละ process

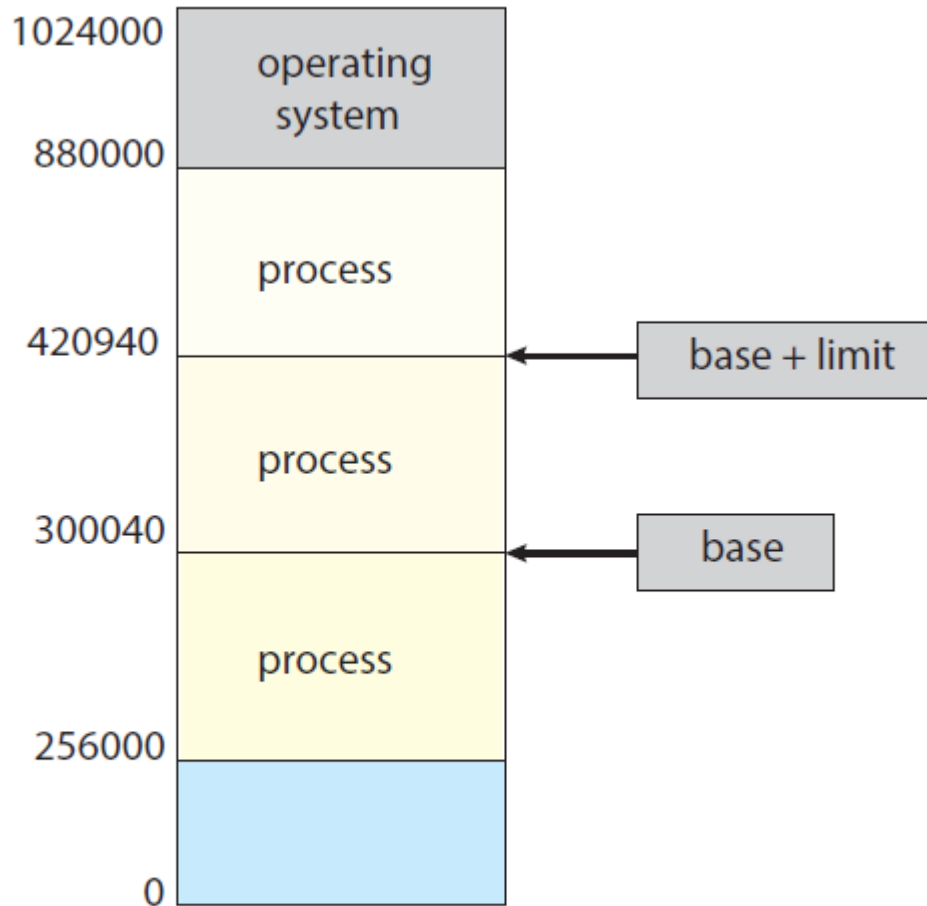


Figure 9.1 A base and a limit register define a logical address space.

Memory Protection

ป้องกันไม่ให้อ่าน/เขียน **memory** เกินขอบเขตที่จองไว้

ใน **kernel mode** เท่านั้น ที่จะแก้ไขค่า **base** และ **limit** ได้

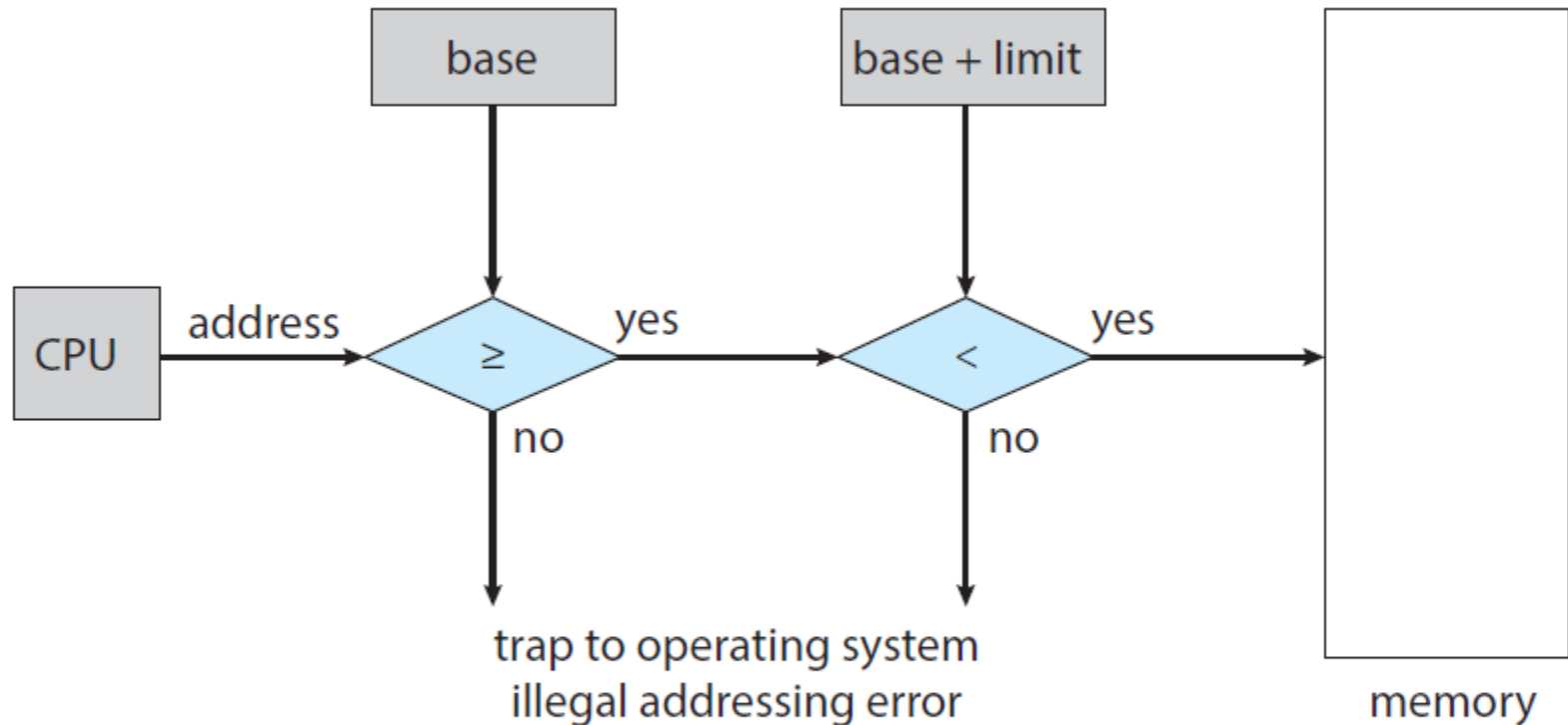


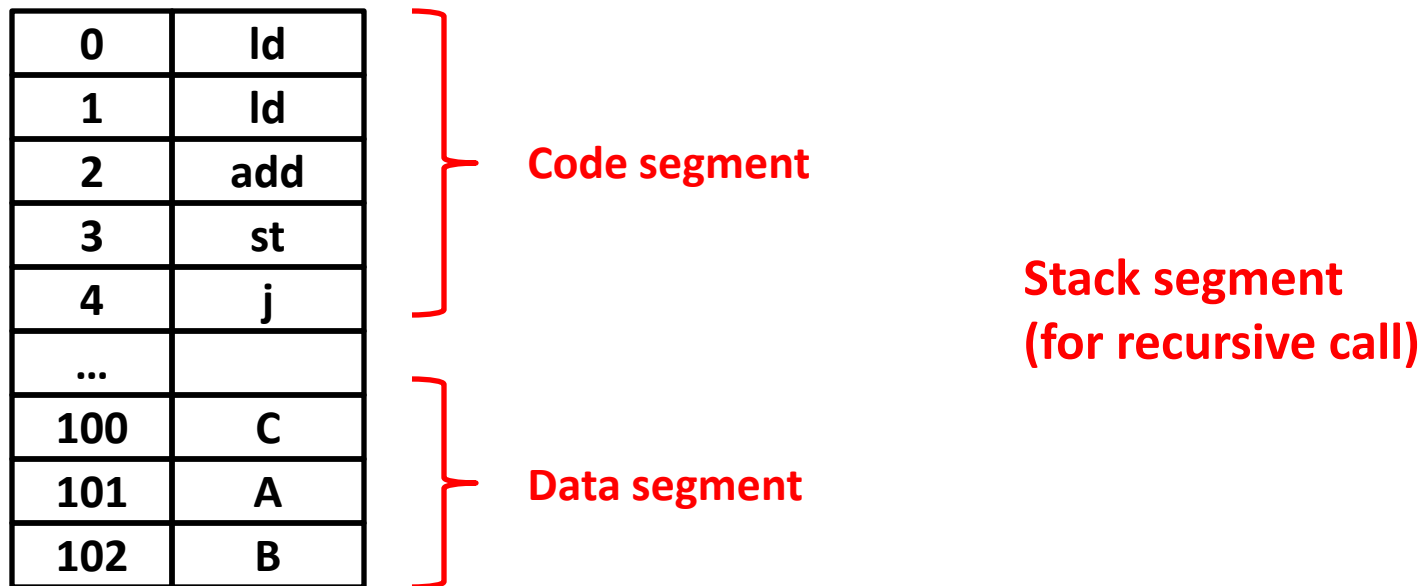
Figure 9.2 Hardware address protection with base and limit registers.

มี **separation & protection** แล้ว แต่ยังต้องโปรแกรมด้วย **physical address**

```
ld    r1    101           // data segment (variable A)
ld    r2    102           // data segment (variable B)
add   r0    r1    r2
st    r0    100           // data segment (variable C = A + B)
J     50                // code segment
```

This program assumes that base address is zero.

If not, the program may encounter illegal memory access.



Logical Address vs. Physical Address

Logical address must begin from zero.

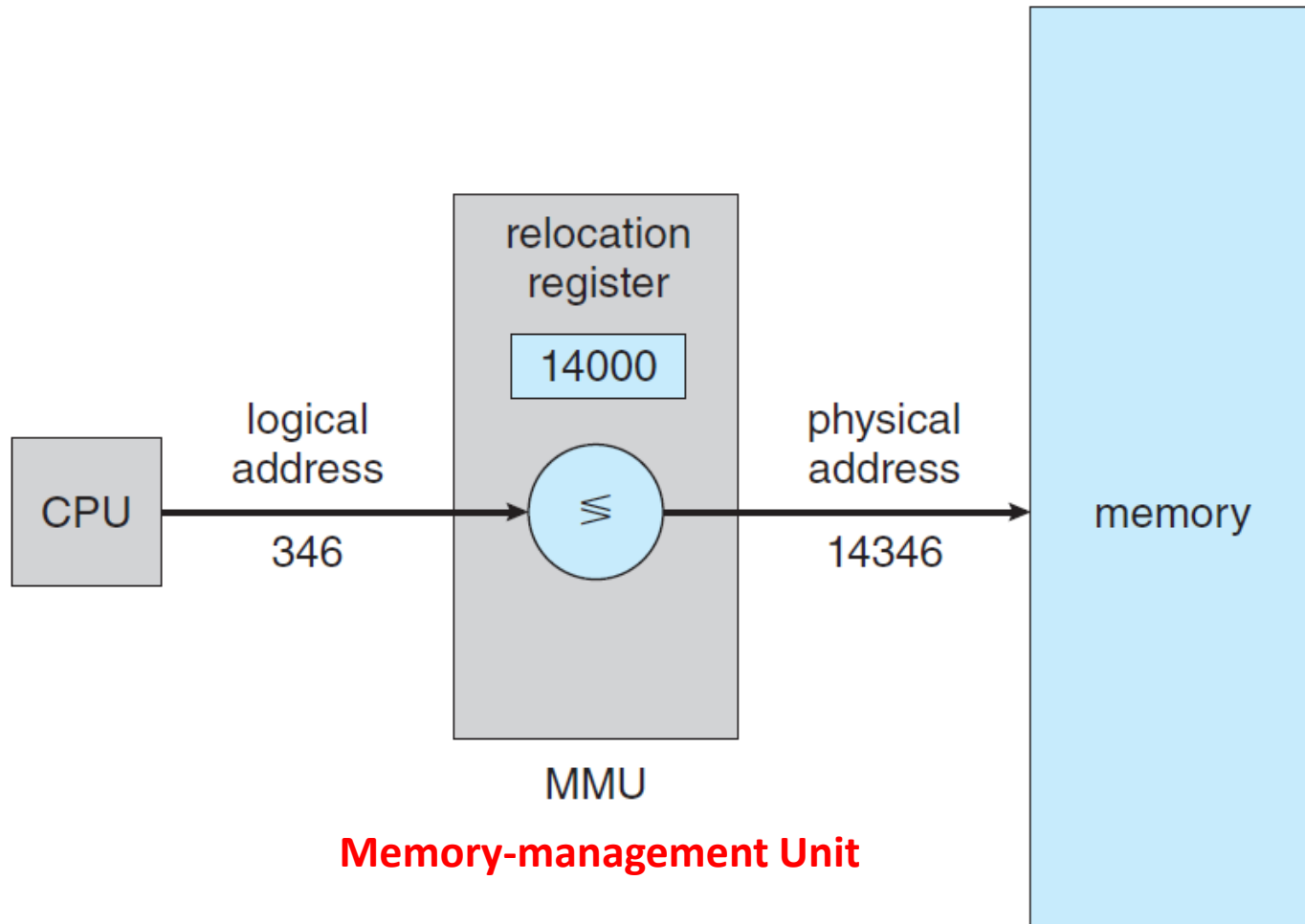


Figure 9.5 Dynamic relocation using a relocation register.

Address Binding

- Compile time

```
ld    r1    101
ld    r2    102
add   r0    r1    r2
st    r0    100
J     50
```

- Load time

```
ld    r1    
ld    r2    
add   r0    r1    r2
st    r0    
J     
```

เว้น address ไว้ก่อน
เมื่อรู้ว่าจะโหลดลงที่ใด
ค่อยเติม addr ให้ถูกต้อง

เกิดจาก dynamic loading
หรือ swapping

- Execution time

ใช้ logical address เริ่มต้นจาก 0
ต้องมี relocation reg.

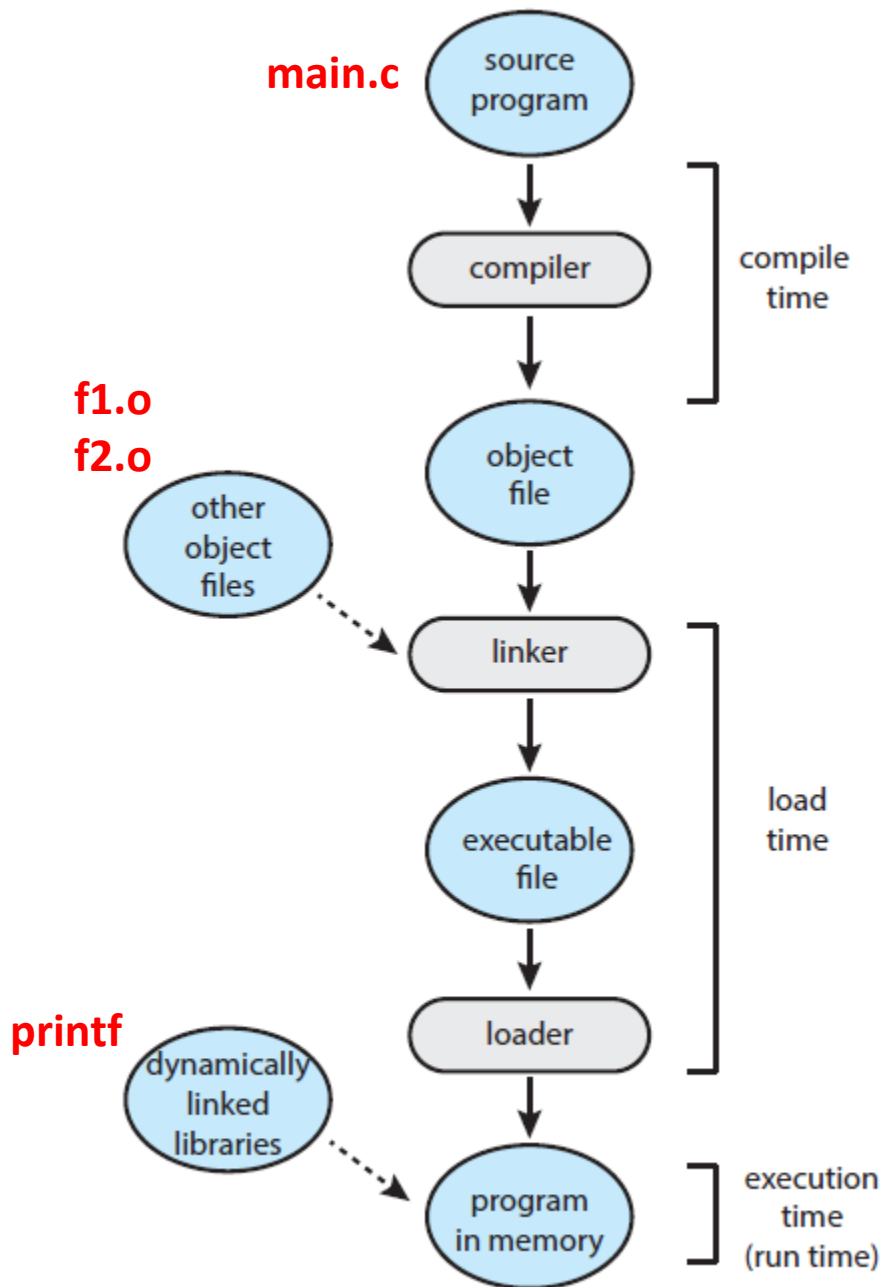
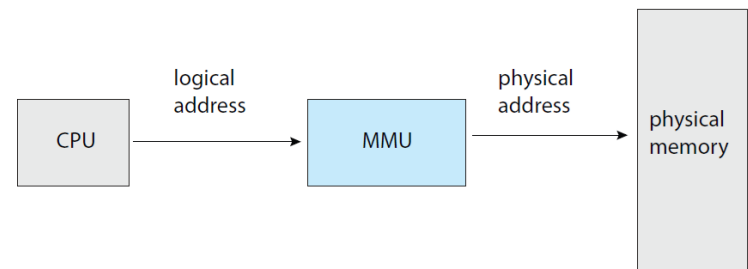


Figure 9.3 Multistep processing of a user program.

Dynamic Loading

1. Load main function into memory.
2. Load other functions on demand.

This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

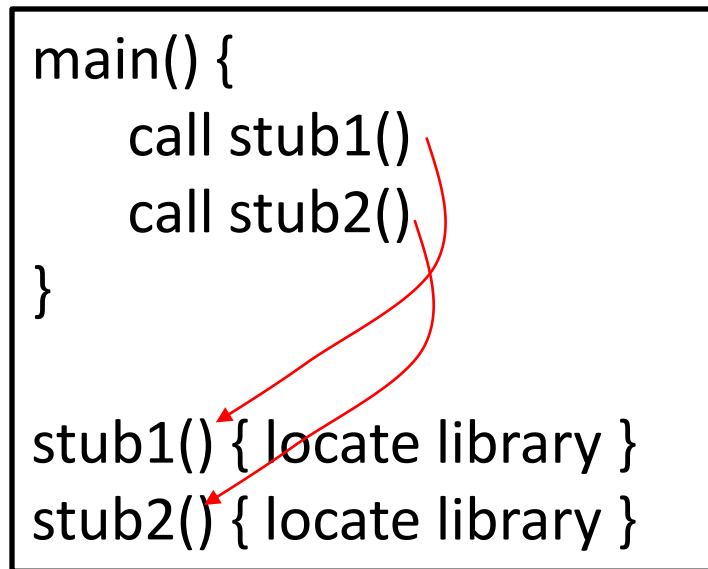
There may be multiple copies of the same routine in memory.

Dynamic Linking and Shared Libraries

1. Static linking $\text{exe} = \text{main.o} + \text{f1.o} + \text{f2.o}$
2. Dynamic linking $\text{exe} = \text{main.o} + \text{stub1} + \text{stub2}$

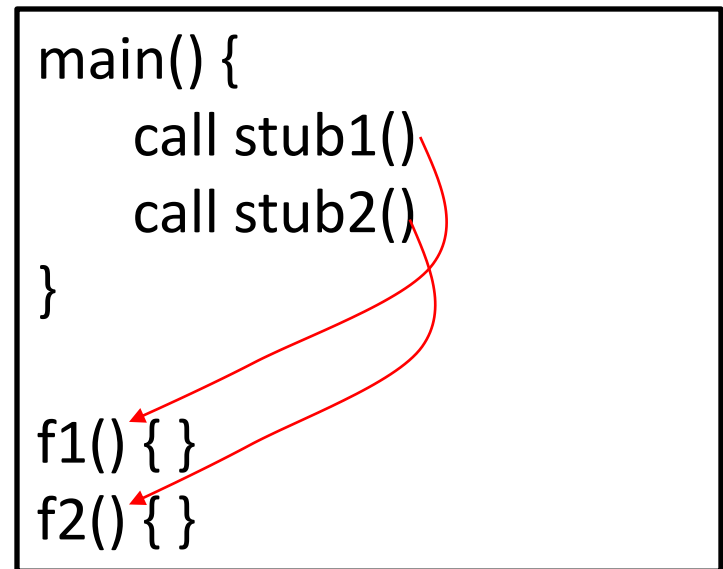
Before calling

```
main() {  
    call stub1()  
    call stub2()  
}  
  
stub1() { locate library }  
stub2() { locate library }
```

A diagram showing the state of the program before calling the stubs. The main function calls stub1() and stub2(). Both stubs are currently configured to 'locate library'. Red arrows point from the 'call stub1()' and 'call stub2()' lines in the main function to the stub1() and stub2() definitions respectively.

After calling

```
main() {  
    call stub1()  
    call stub2()  
}  
  
f1() { }  
f2() { }
```

A diagram showing the state of the program after calling the stubs. The main function calls stub1() and stub2(). The stubs have been replaced by the actual functions f1() and f2(). Red arrows point from the 'call stub1()' and 'call stub2()' lines in the main function to the f1() and f2() definitions respectively.

There is only single copy of f1() and f2() in memory.

Swapping

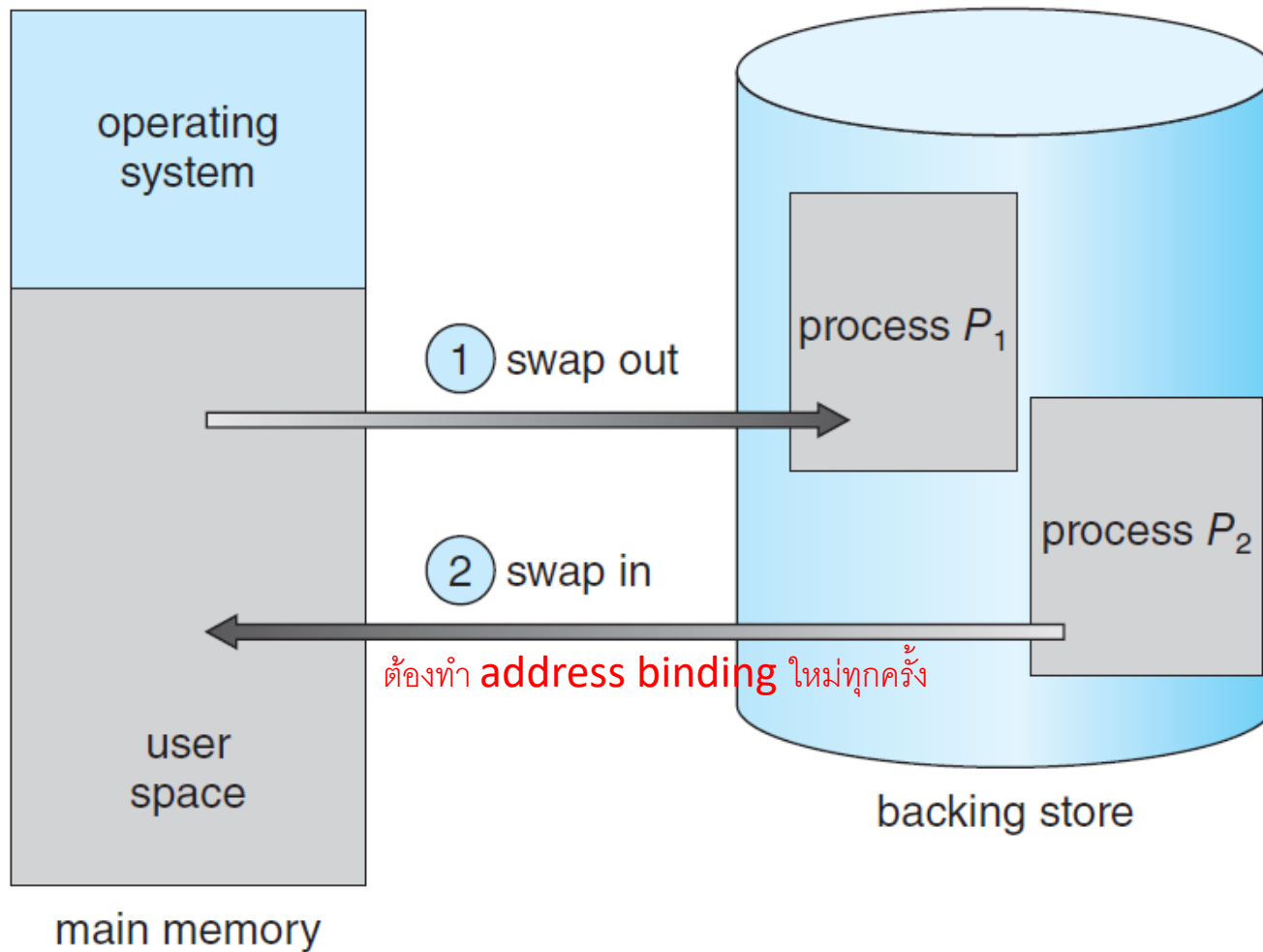


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Memory Mapping & Protection

ยังเป็น Contiguous Memory Allocation

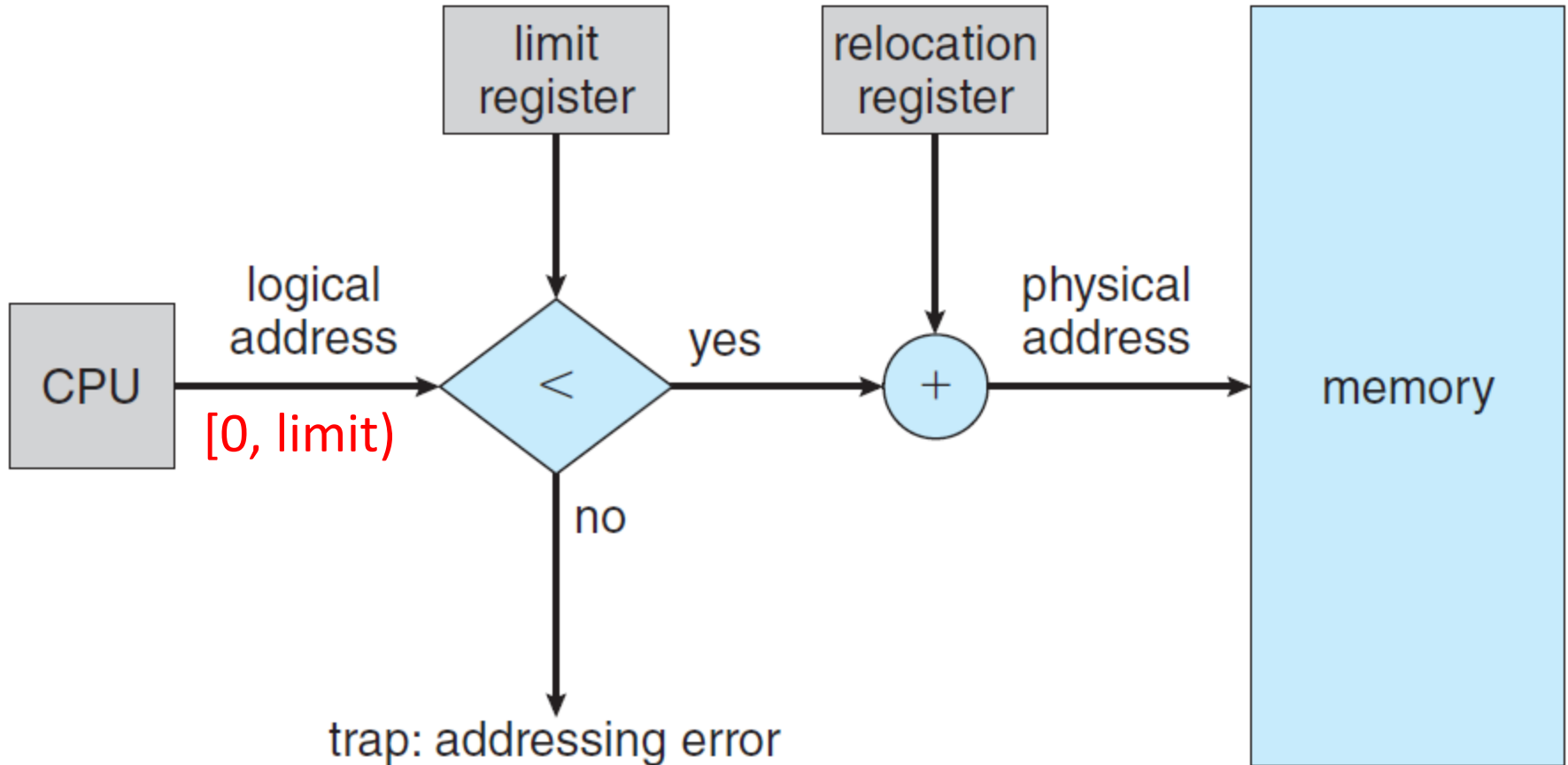


Figure 9.6 Hardware support for relocation and limit registers.

Memory Allocation

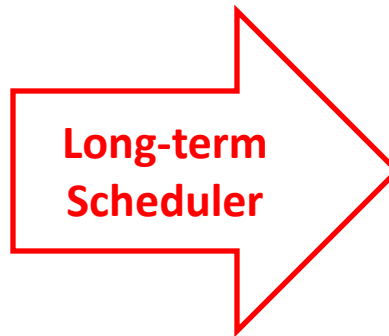
Input queue
(waiting processes)

Process 5

Process 6

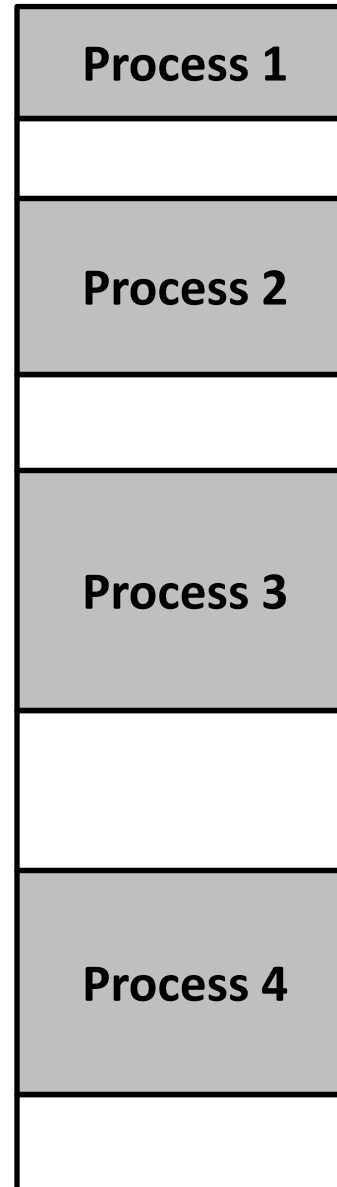
Process 7

Process 8



First Fit
Best Fit
Worst Fit

Memory



← Hole

Fragmentation

External fragmentation

Holes in memory.

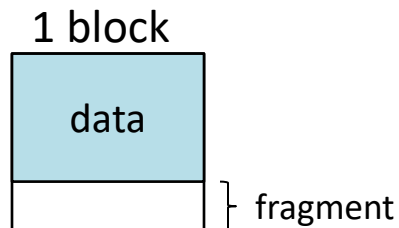
แก้ด้วยวิธี 1 ทำ compaction

แก้ด้วยวิธี 2 ใช้ non-contiguous memory

50-percent rule (first fit), given n blocks allocated, $0.5n$ will be lost.
One-third of memory may be unusable.

Internal fragmentation

เกิดจากการจอง memory เป็น block แล้วใช้ไม่เต็ม block



Paging

Frame fixed-size block on physical memory

Page fixed-size block on logical memory

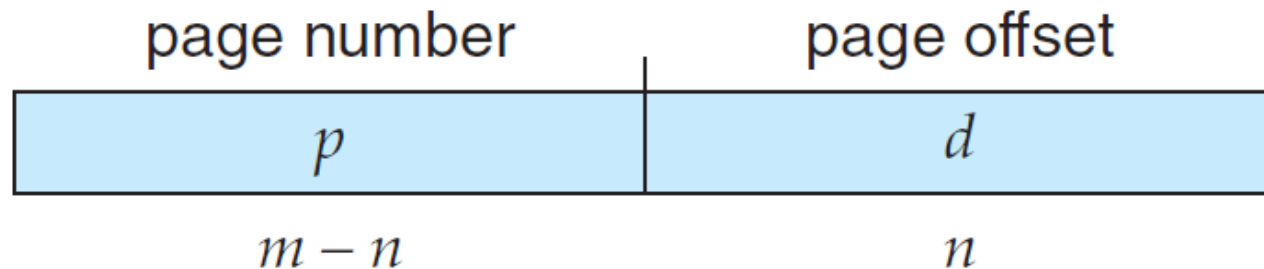
Page number (p) left side of logical address

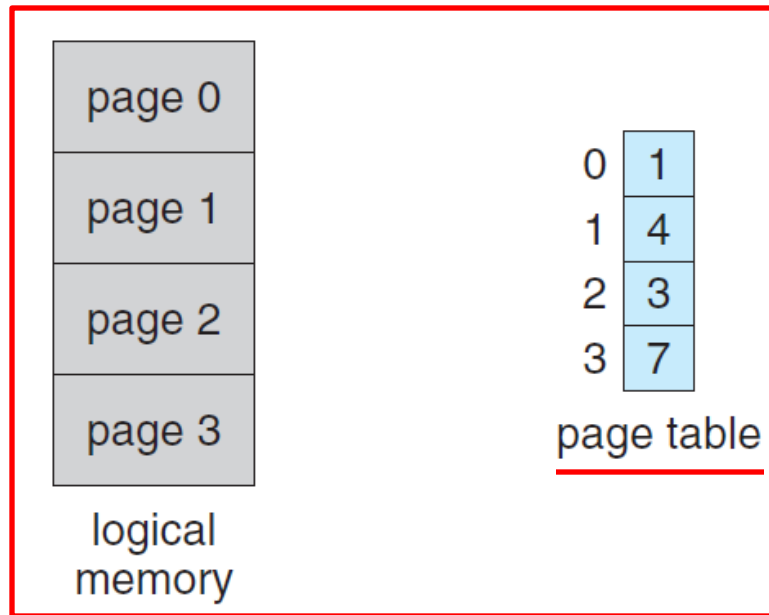
Page offset (d) right side of logical address

Page size = frame size Windows 10 supports page sizes of **4KB** (typical) and 2MB.
Linux also supports two page sizes.

m number of bits (logical address space)

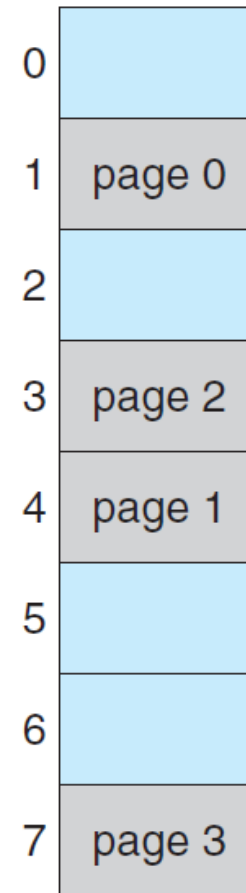
n number of bits (page size)





1 process

frame
number



Non-contiguous

physical
memory

Figure 9.9 Paging model of logical and physical memory.

Page table is in memory, and it is indexed by PTBR (page-table base register) ที่จริงแล้ว page table ก็อยู่ใน memory นั่นเอง ทำให้ทุกครั้งที่โปรแกรมจะ read/write memory ก็ต้องอ่าน page table ก่อน ทำให้ต้องอ่าน memory 2 รอบ

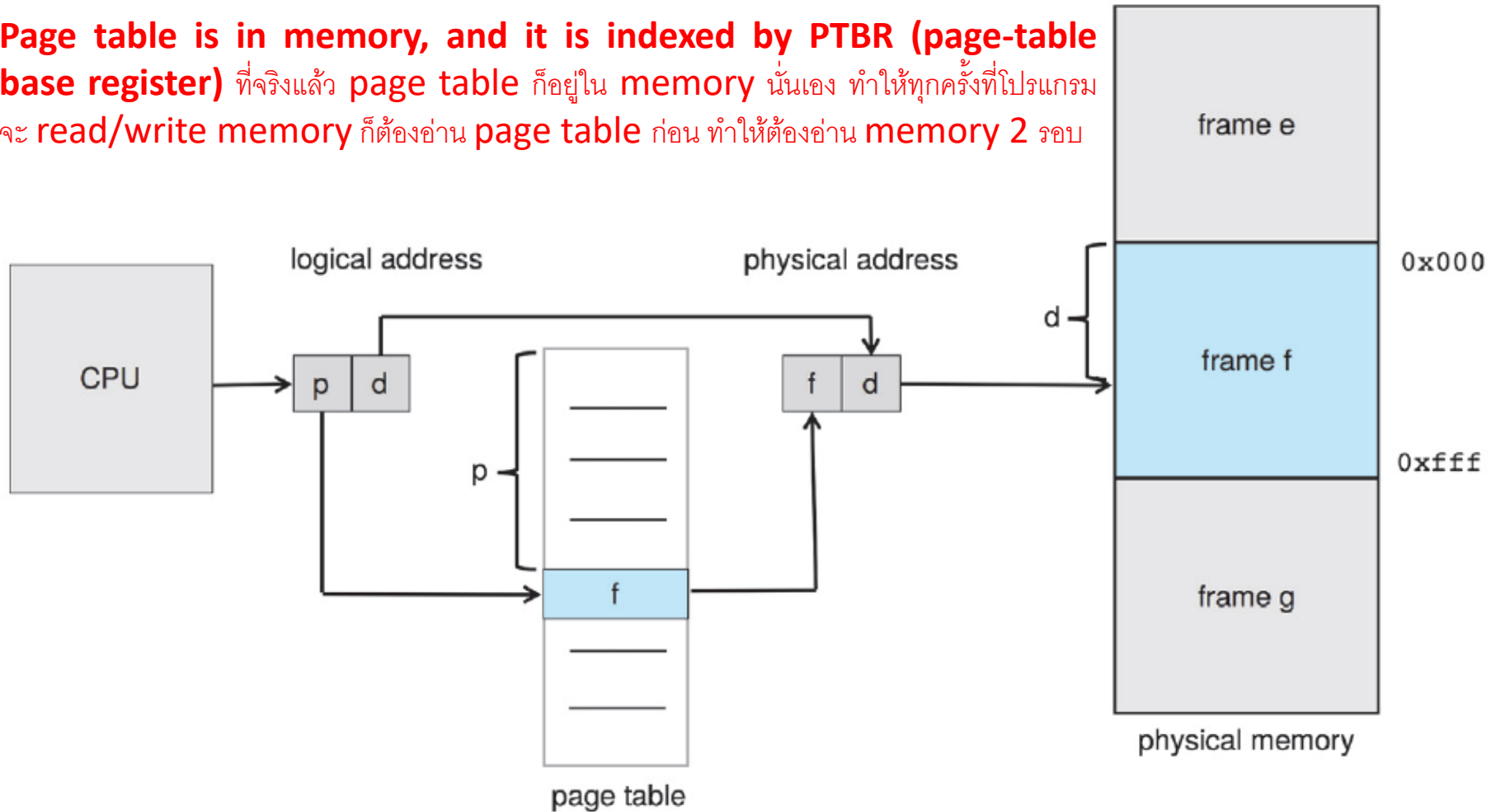


Figure 9.8 Paging hardware.

$m = 4, n = 2$

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

} Page size = 2^n

0	5
1	6
2	1
3	2

page table

မီ memory 2^m နံရံ

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory မီ memory 32 နံရံ

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

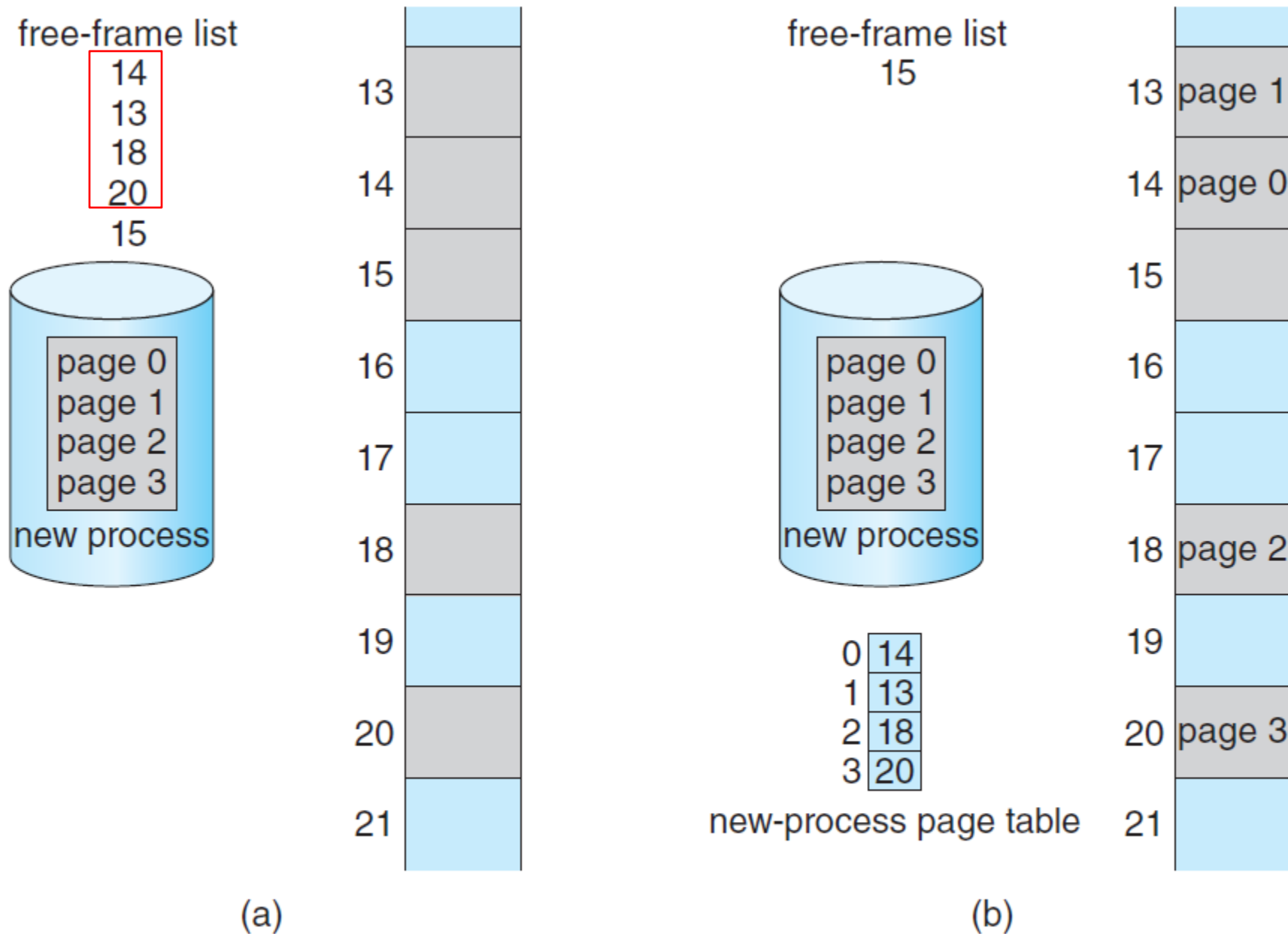


Figure 9.11 Free frames (a) before allocation and (b) after allocation.

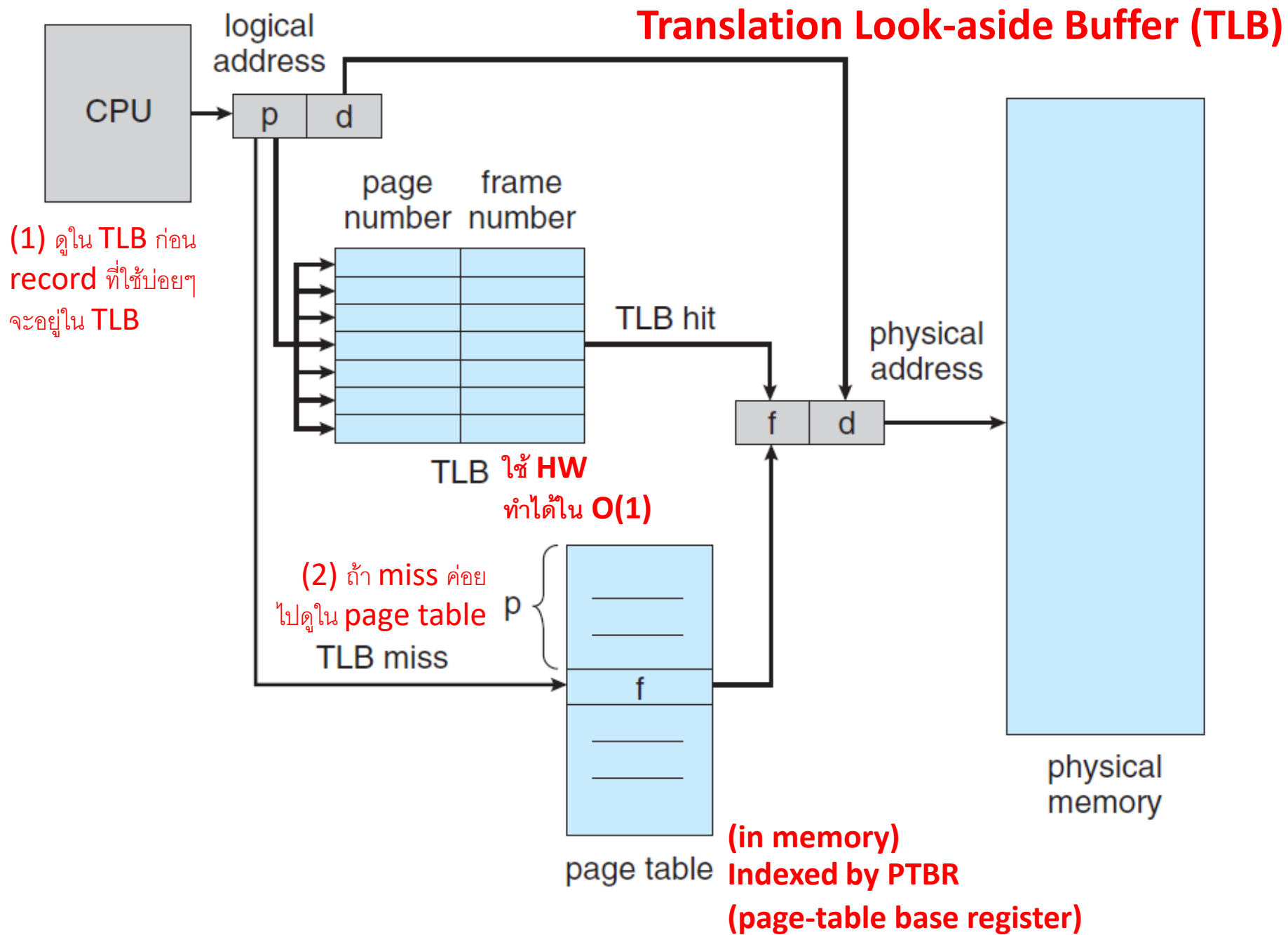


Figure 9.12 Paging hardware with TLB.

Protection

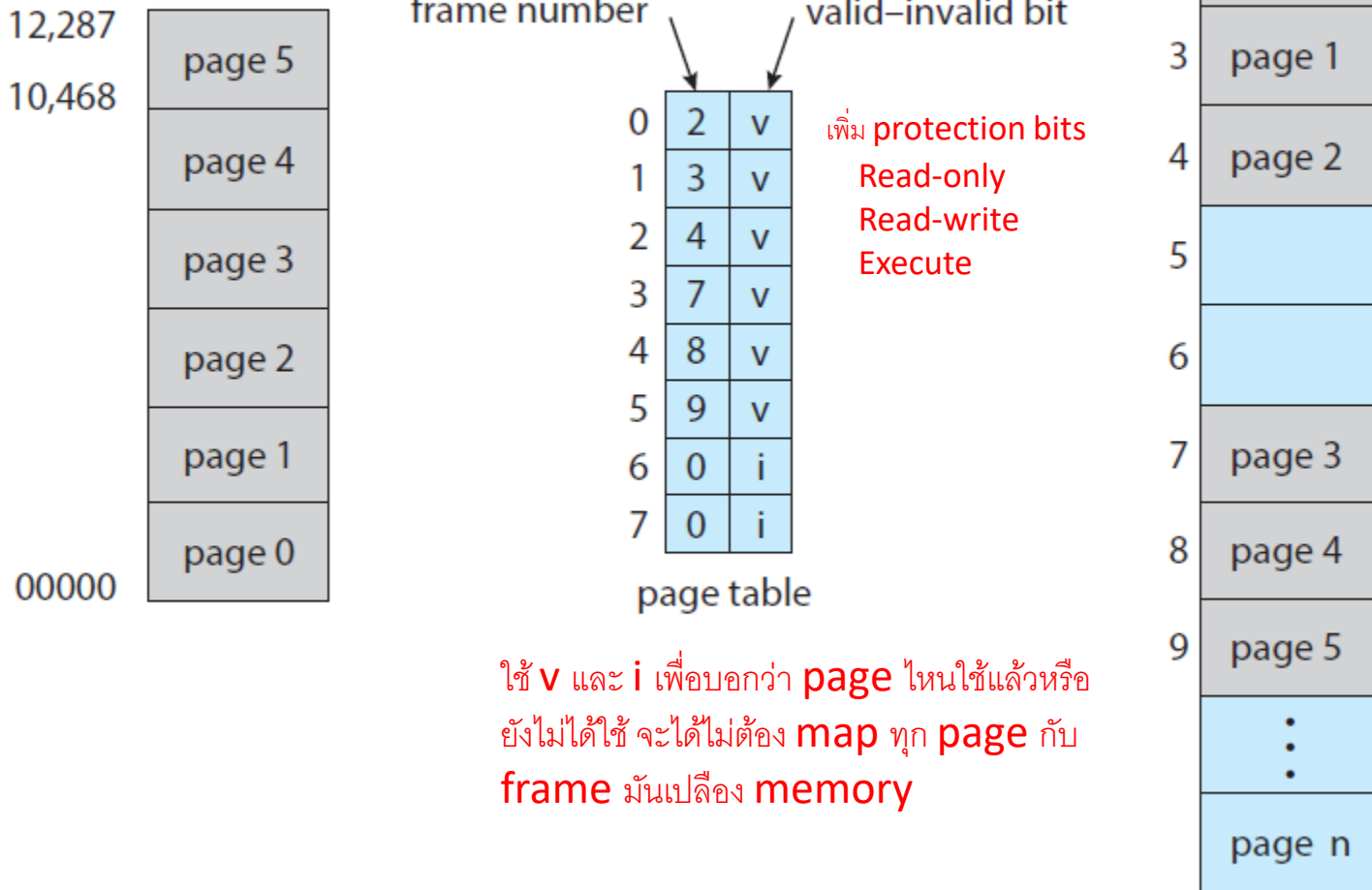


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

If the code is **reentrant code**, however, it can be shared.

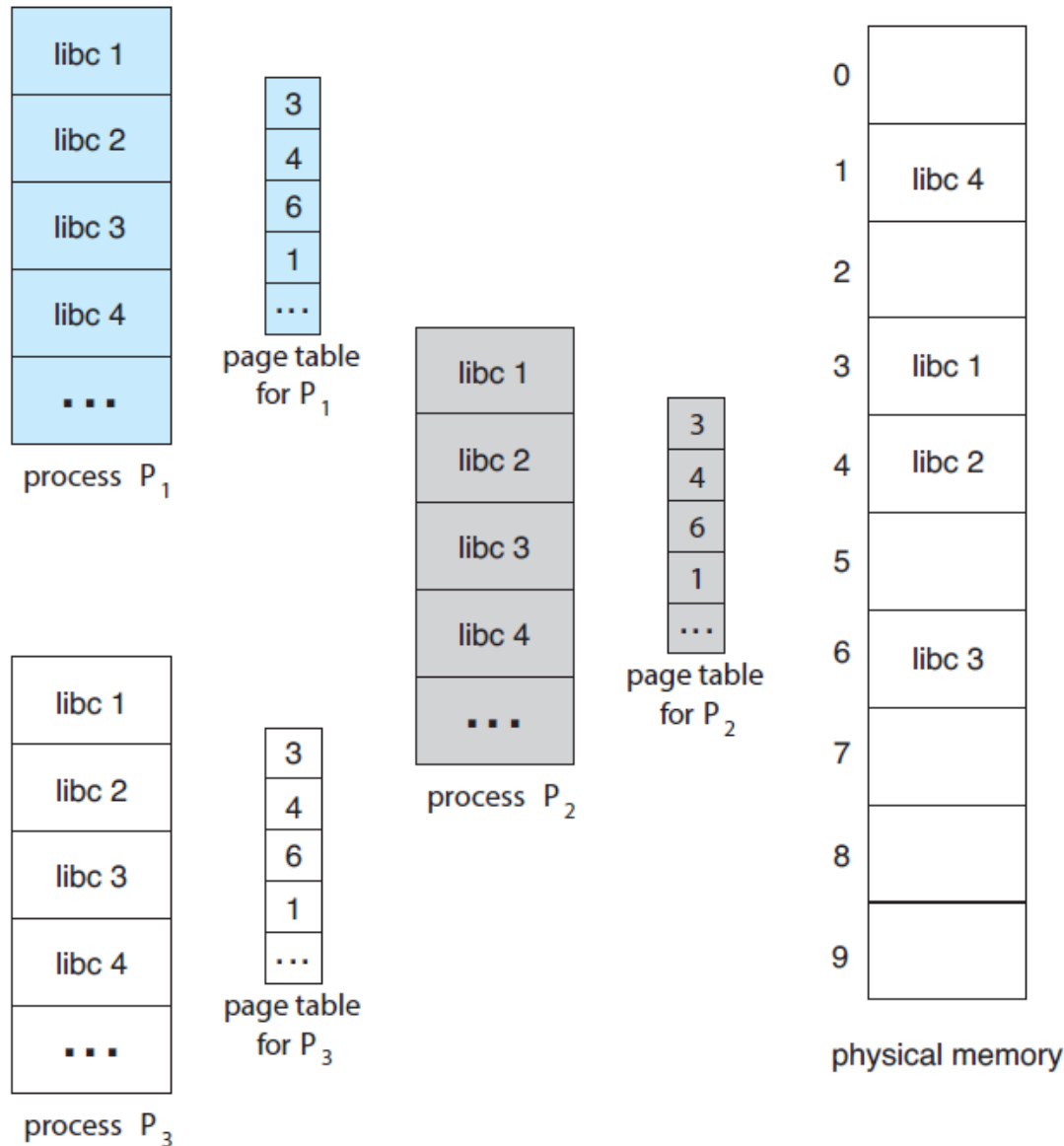


Figure 9.14 Sharing of standard C library in a paging environment.

Hierarchical Paging (32-bit)

page number		page offset
p_1	p_2	d
10	10	12

Create inner page tables on demand.

1 page = 2^{12} entries

No hierarchical paging

page table = $2^{20} = 1$ million entries

A new process costs 1M.

Hierarchical paging

new process = $2^{10} + 2^{10} = 2K$ entries

A new process costs 2K.

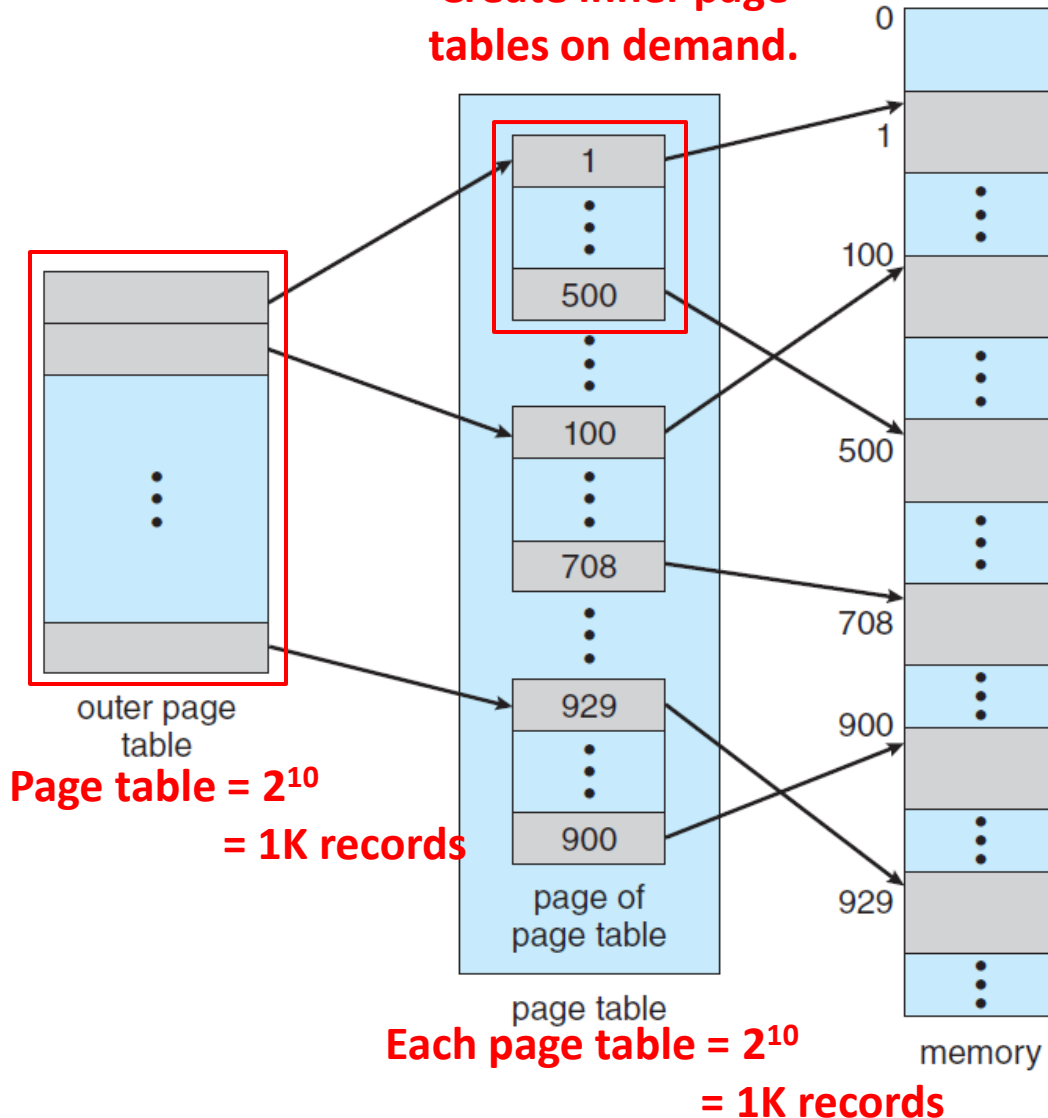
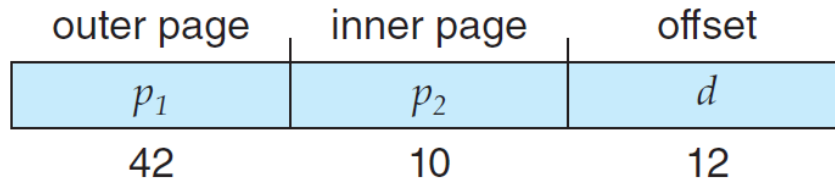
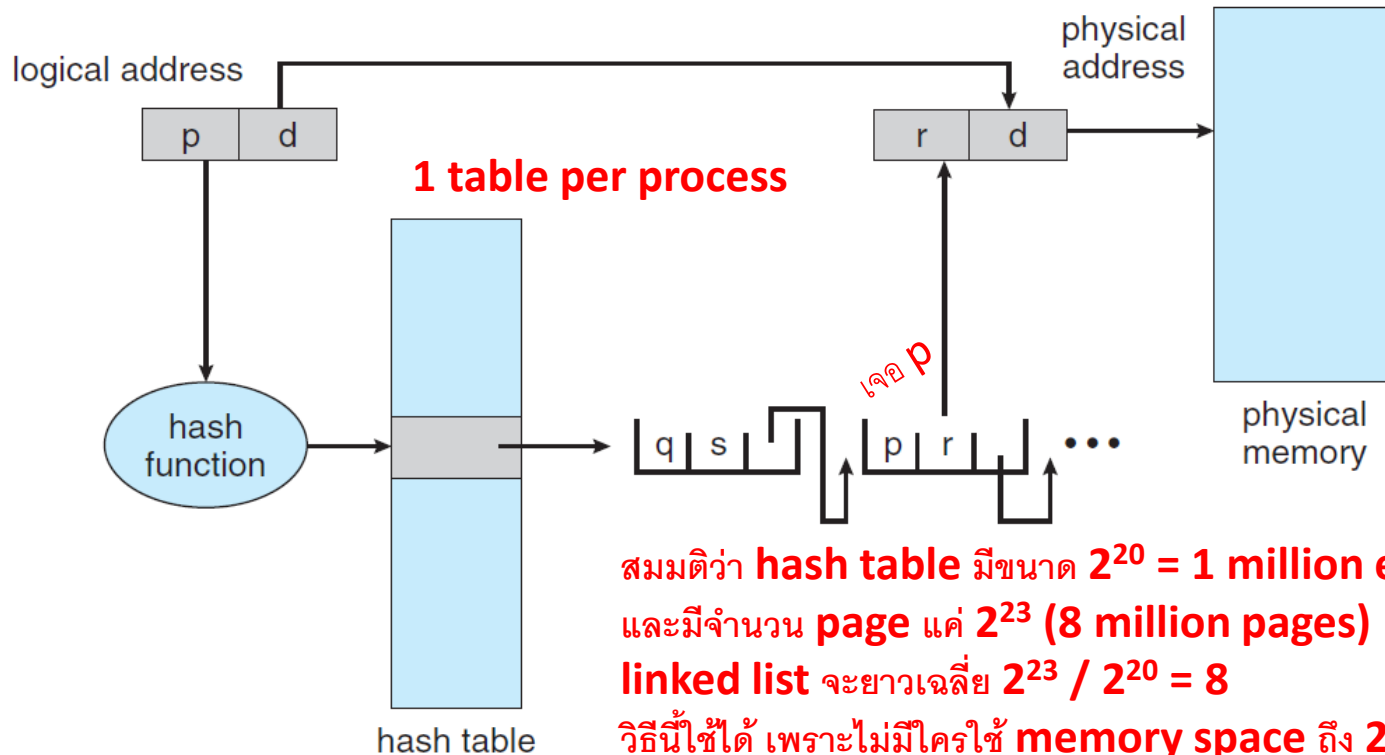


Figure 9.15 A two-level page-table scheme.

Hashed Page Tables (64-bit)



ทำแบบเดิมไม่ได้แล้ว เพราะ 2^{42} ก็ยังใหญ่เกินไป
หรือแบ่งครึ่ง $2^{26} = 64\text{M entries}$ ก็ยังใหญ่อยู่ดี



สมมติว่า **hash table** มีขนาด $2^{20} = 1 \text{ million entries}$

และมีจำนวน **page** แค่ 2^{23} (8 million pages)

linked list จะยาวเฉลี่ย $2^{23} / 2^{20} = 8$

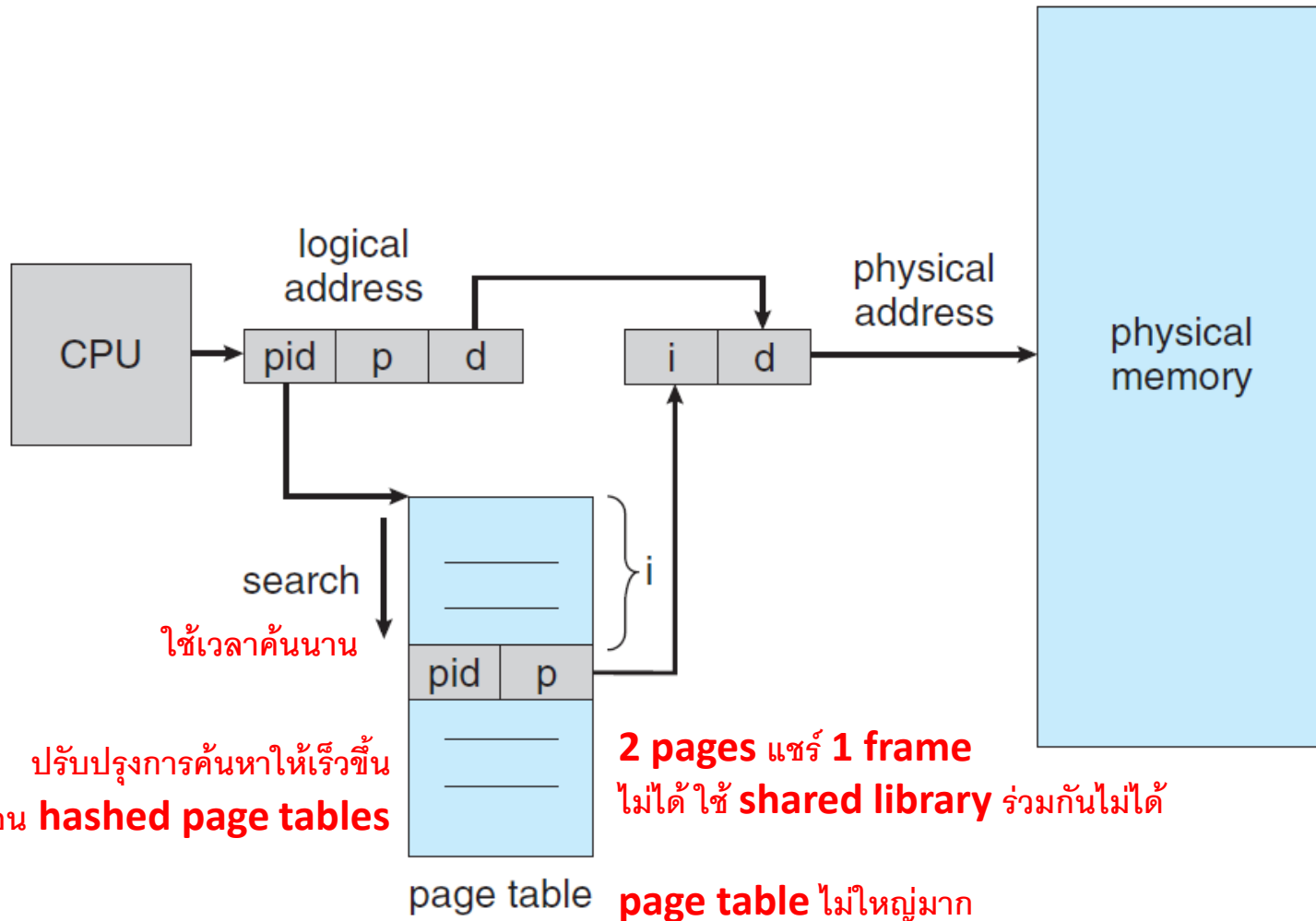
วิธีนี้ได้ เพราะไม่มีใครใช้ **memory space** ถึง 2^{64} จริง ๆ

ถ้าใช้ **clustered page tables** จะทำให้ **linked list** มี **max length** คงที่

Figure 9.17 Hashed page table.

Inverted Page Tables (64-bit)

ทั้งระบบมี **page table** แค่ตารางเดียว ไม่ **per process**



ปรับปรุ้งการค้นหให้เร็วขึ้น
โดยทำเหมือน **hashed page tables**

ใช้เวลาคั่นนาน

2 pages แשר **1 frame**
ไม่ได้ใช้ **shared library** ร่วมกันไม่ได้

page table ไม่ใหญ่มาก
เพราะไม่มีใครใช้ **memory space** ถึง 2^{64} จริง ๆ

Figure 9.18 Inverted page table.

Segmentation

<segment-number, offset>.

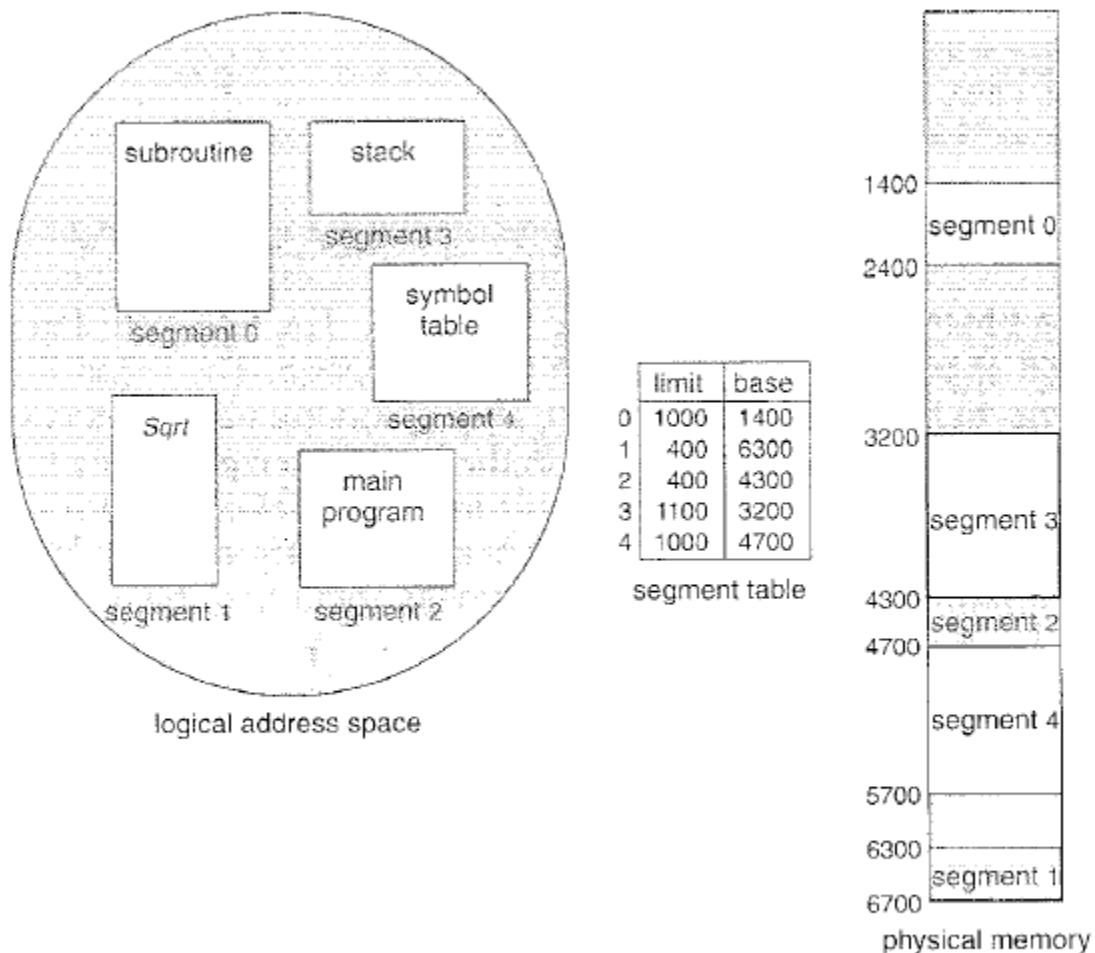


Figure 8.20 Example of segmentation.

รูปนี้มีแคใน **textbook** เล่มเก่า
แต่อธิบายเข้าใจง่ายกว่า

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

Here, *s* designates the segment number, *g* indicates whether the segment is in the GDT or LDT, and *p* deals with protection. The offset is a 32-bit number specifying the location of the byte within the segment in question.

GDT/LDT = Global/Local Descriptor Table (Intel)

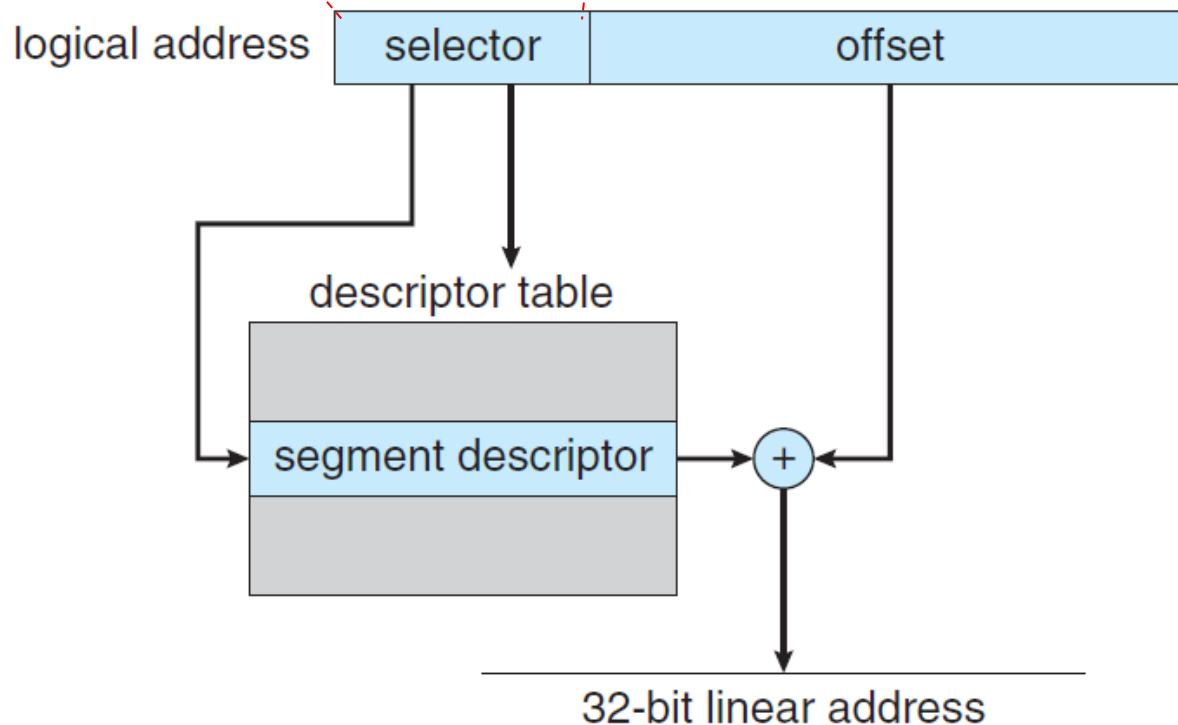


Figure 9.22 IA-32 segmentation.

IA-32 Architecture

32-bit processors generally compatible with the Intel Pentium® II processor

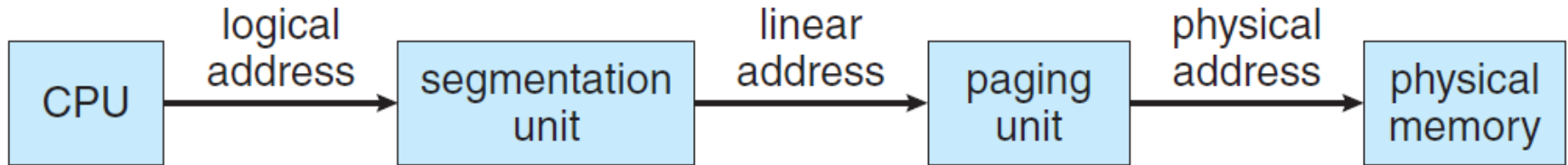


Figure 9.21 Logical to physical address translation in IA-32.

Chapter 9 Main Memory

9.1 Background	349
9.2 Contiguous Memory Allocation	356
9.3 Paging	360
9.4 Structure of the Page Table	371
9.5 Swapping	376

9.6 Example: Intel 32- and 64-bit Architectures	379
9.7 Example: ARMv8 Architecture	383
9.8 Summary	384
Practice Exercises	385
Further Reading	387

ลองพิมพ์คำสั่ง `getconf PAGE_SIZE` บน Linux ค่าที่ได้คืออะไร?

Wired down ใน TLB คืออะไร?

Effective access time ของ memory คืออะไร? คำนวณยังไง?

Reentrant code คืออะไร?