CS222
Operating Systems

Lecture 11
Memory Management

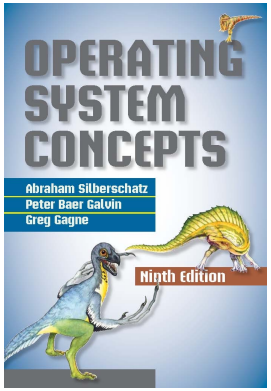
(Section 100001)

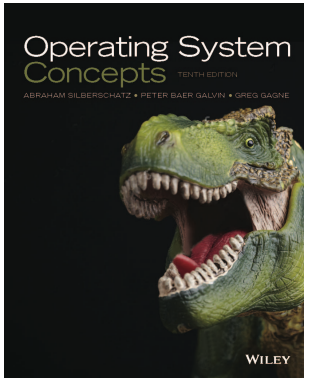
ผศ. ดร. กษิดิศ ชาญเชี่ยว

ckasidit@tu.ac.th

Textbook

- Avi Silberschatz, Peter B. Galvin and Greg Gagne; Operating System Concepts, 9th Edition; John Wiley & Sons, Inc; 2012; ISBN 978-1118063330





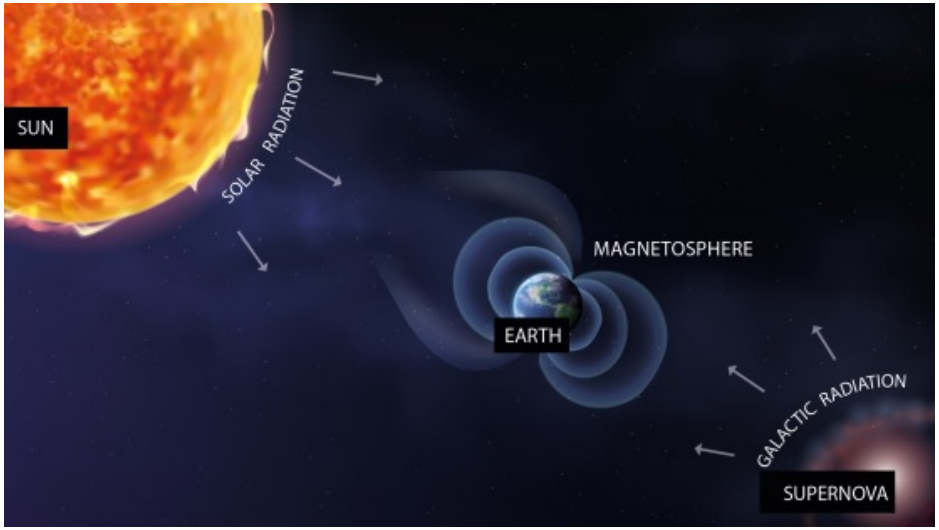
- Chapter 9 (10th edition)

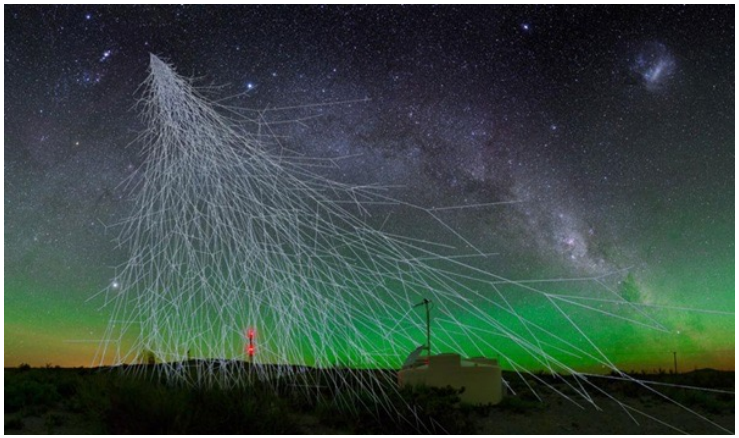
  Memory Management

- Original Slides
- https://www.os-book.com/OS9/slide-dir/index.html
- https://www.os-book.com/OS10/slide-dir/index.html

เกี่ยวกับ cosmic rays

- อนุภาค ที่มาจากภายนอกโลก
- ชนกับชั้นบรรยากาศ แล้วทำให้เกิดอนุภาคอื่นตกมายังพื้นโลก อนุภาคเช่น นิวตรอน อิเลคตรอน ส่งผลต่อระบบ ดิจิตอล
- ตย. คะแนนผลการเลือกตั้งในเบลเยี่ยมเพิ่ม หรือ ผู้เล่นเกมส์ Supermario สามารถกระโดดได้เกินกว่าชั้นที่โปรแกรมกำหนด
- บน Space Shuttle ต้องใช้ คอม 4 เครื่องพร้อมกันและนำผลจากการคำนวณทุกอย่างมา vote เป็นค่าผลลัพธ์
- Astronaut เห็นแสงสว่าง Light Flash ในตา



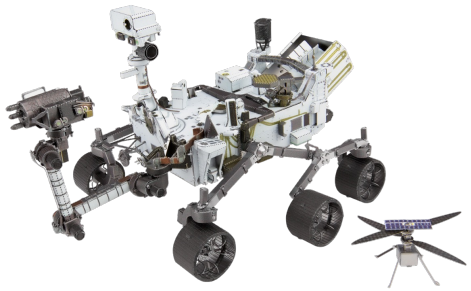


เกี่ยวกับ Radiation กับอุปกรณ์ Electronics

- ใน MARS rover ต้องใช้ Mem น้อย และซีพียูป้องกัน Radiation (รวมทั้งจาก cosmic rays)

Comparison of embedded computer systems on board the Mars rovers

Rover (mission, organization, year)	CPUs	RAM	Flash	EEPROM	Operating system
Sojourner Rover (Pathfinder, NASA, 1997)[1][3][4][5]	2 MHz[6] Intel 80C85	512 KB	176 KB	None	Custom cyclic executive
Pathfinder Lander (NASA, 1997)[1] (Base station for *Sojourner* rover)	20 MHz MFC	128 MB	None	6 MB	VxWorks[7] (multitasking)
Spirit and *Opportunity* (Mars Exploration Rover (MER), NASA, 2004)[1]	20 MHz BAE RAD6000	128 MB	256 MB	3 MB	VxWorks (multitasking)
Curiosity (Mars Science Laboratory (MSL), NASA, 2011)[1][8][9]	200 MHz BAE RAD750	256 MB	2 GB	256 KB	VxWorks (multitasking)
Perseverance (Mars 2020, NASA, 2020)[10]	200 MHz BAE RAD750	256 MB	2 GB	256 KB	VxWorks (multitasking)[11]



- ระดับของ Radiation สูงกว่าบนโลกหลายเท่า
- อุปกรณ์อิเลคโทรนิคจะทนต่อ Radiation ได้พอสมควร ได้ในระดับหนึ่ง

300 MHz Motorola PowerPC 750 processor with off-die L2 cache on the CPU module from a Power Mac G3.

เกี่ยวกับ Memory (terms)

- **แบ่งเป็นสองแบบได้แก่ ECC Memory และ Non-ECC Memory**
- **ECC Memory หรือ** Error Correction Code (ECC) เป็นอุปกรณ์ฮาร์ดแวร์หน่วยความจำที่สามารถตรวจจับและแก้ไขข้อผิดพลาด หากเกิด BIT FLIP จำนวน 1 bit ได้
  - มักใช้บน Server ที่รันเป็นเวลานาน หรือ Mission Critical Computer ที่มีโอกาสที่จะถูกชนโดย ฝนอนุภาคที่เกิดจาก การที่ cosmic rays ชนชั้นบรรยากาศโลก
- **Non-ECC Memory** คืออุปกรณ์หน่วยความจำธรรมดา ที่ไม่สามารถตรวจจับความผิดพลาดได้ มักใช้บน Desktop หรือ Notebook
- **Soft error** คือความผิดพลาดที่เกิดจากข้อมูลผิด อาจเกิดเพราะอปกรผิดพลาด หรือ ไม่ได้เกิดจากสิ่งอื่นเช่น cosmic rays
- ECC ***ไม่สามารถป้องกัน***การโจมตีแบบ Row Hammer ที่ใช้วิธีการเขียน DRAM ในรูปแบบหนึ่งซ้ำแล้วซ้ำอีกและส่งผลให้ข้อมูลของ RAM ที่อยู่ใกล้เคียงตำแหน่งที่เกิดการเขียนบ่อยเปลี่ยนได้ ***ถ้ามี bitflips มากว่า 1 bit***

ความรู้ทั่วไป

อ้างอิง: ภาพและข้อมูลจาก Wikipedia

https://www.metalearth.com/mars-rover-perseverance-ingenuity-helicopter

https://astronomy.com/news/2021/01/the-history-of-cosmic-rays-is-buried-beneath-our-feet

https://www.youtube.com/watch?v=AaZ_RSt0KP8

https://www.cashify.in/best-processor-for-mobile-phone-ranking-list

https://www.iaea.org/newscenter/news/cosmic-radiation-why-we-should-not-be-worried

https://docs.kernel.org/scheduler/sched-energy.html

https://docs.kernel.org/scheduler/sched-capacity.html

Chapter 9: Memory Management

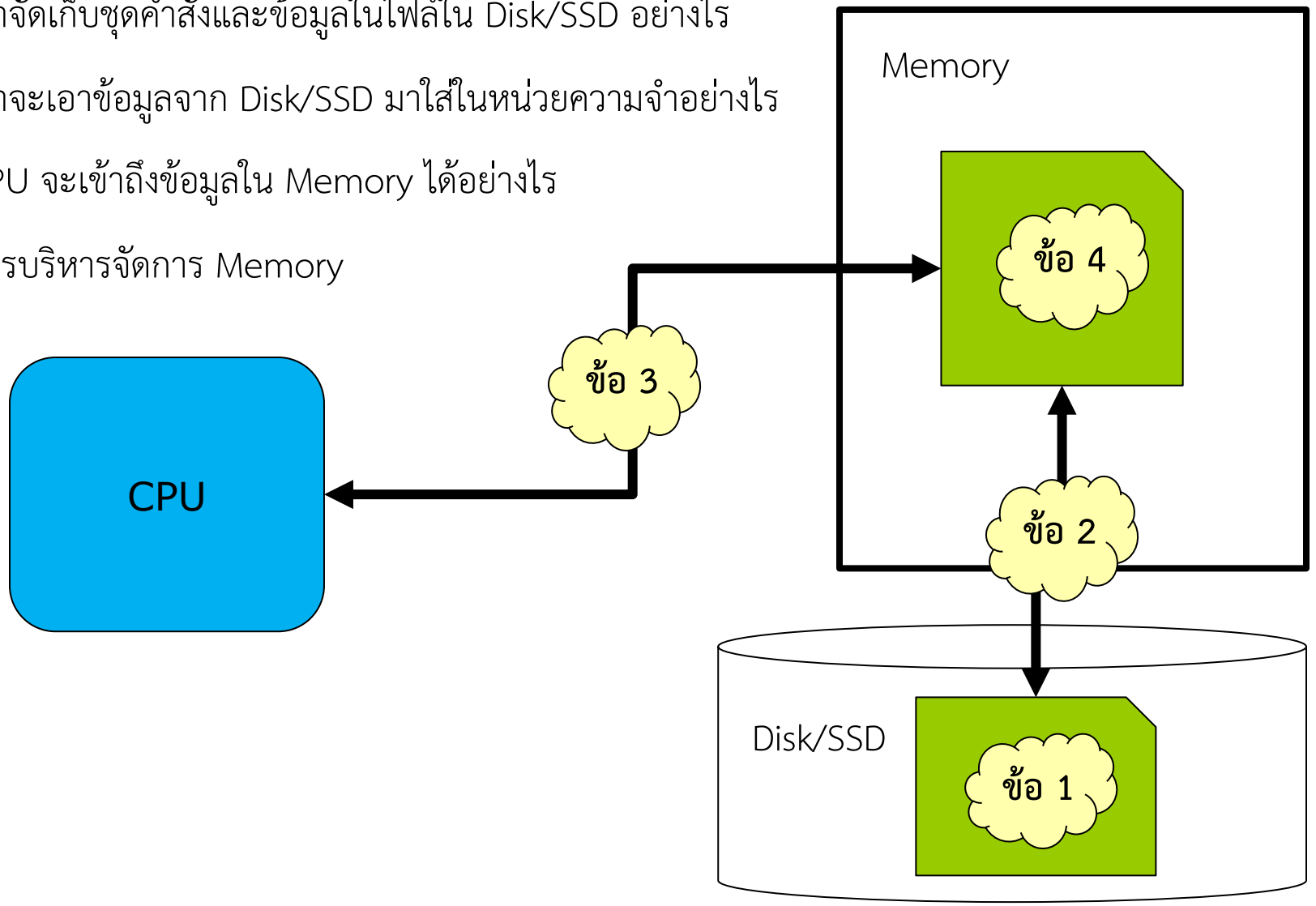
- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,

โจทย์ที่เราต้องการศึกษา

1. เราจัดเก็บชุดคำสั่งและข้อมูลในไฟล์ใน Disk/SSD อย่างไร
2. เราจะเอาข้อมูลจาก Disk/SSD มาใส่ในหน่วยความจำอย่างไร
3. CPU จะเข้าถึงข้อมูลใน Memory ได้อย่างไร
4. การบริหารจัดการ Memory



การจัดการ Memory แบบพื้นฐาน

Background

- **Program must be brought (from disk) into memory and placed within a process for it to be run**
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests

Background

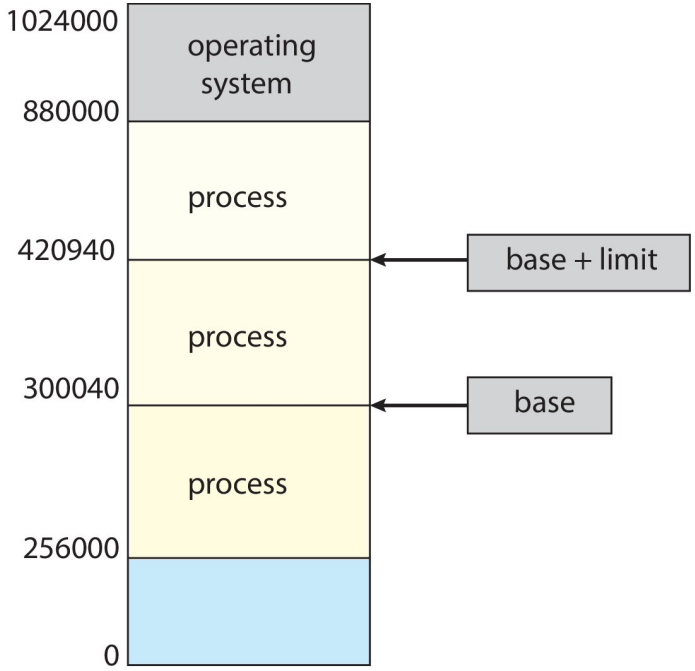
- Register access is done in one CPU clock (or less)
- แต่ Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
  - การบริหารจัดการ cache ทำโดย hardware
- **Protection of memory required to ensure correct operation**
  - ทุก Process มี Separate Memory Space
  - hardware ป้องกันการเข้าถึงข้อมูลของ Process หนึ่ง โดย Process อื่น

Protection

- Protection of memory required to ensure correct operation
  - ทุก Process มี Separate Memory Space
  - hardware ป้องกันการเข้าถึงข้อมูลของ Process หนึ่ง โดย Process อื่น

base = 300040

limit = 120900

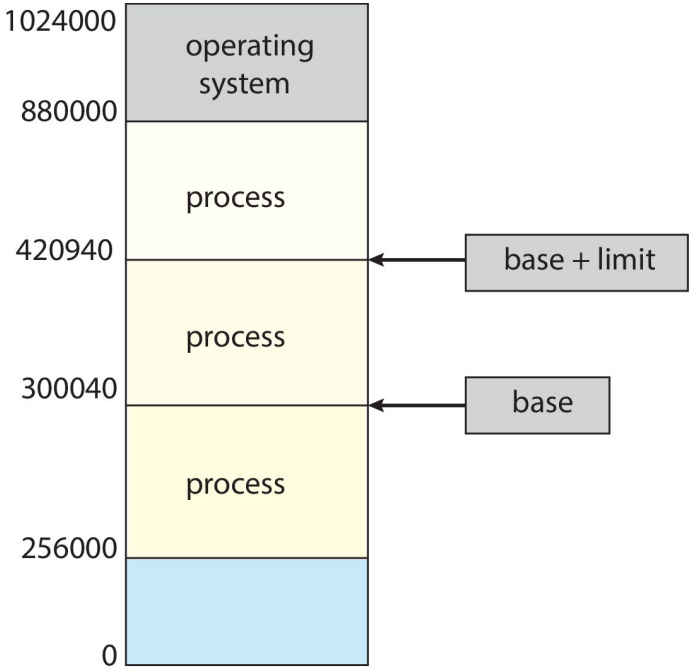


Protection

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process

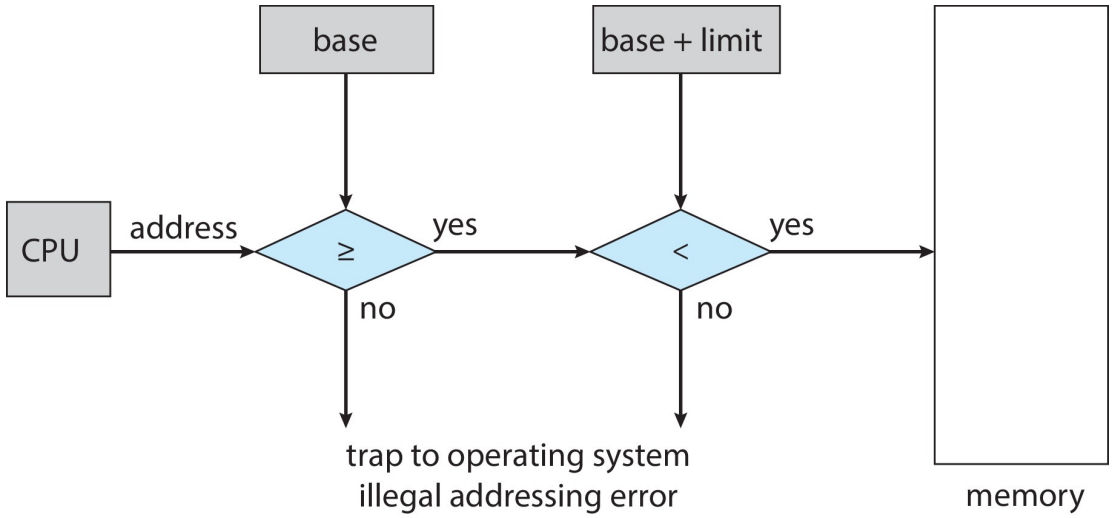
base = 300040

limit = 120900



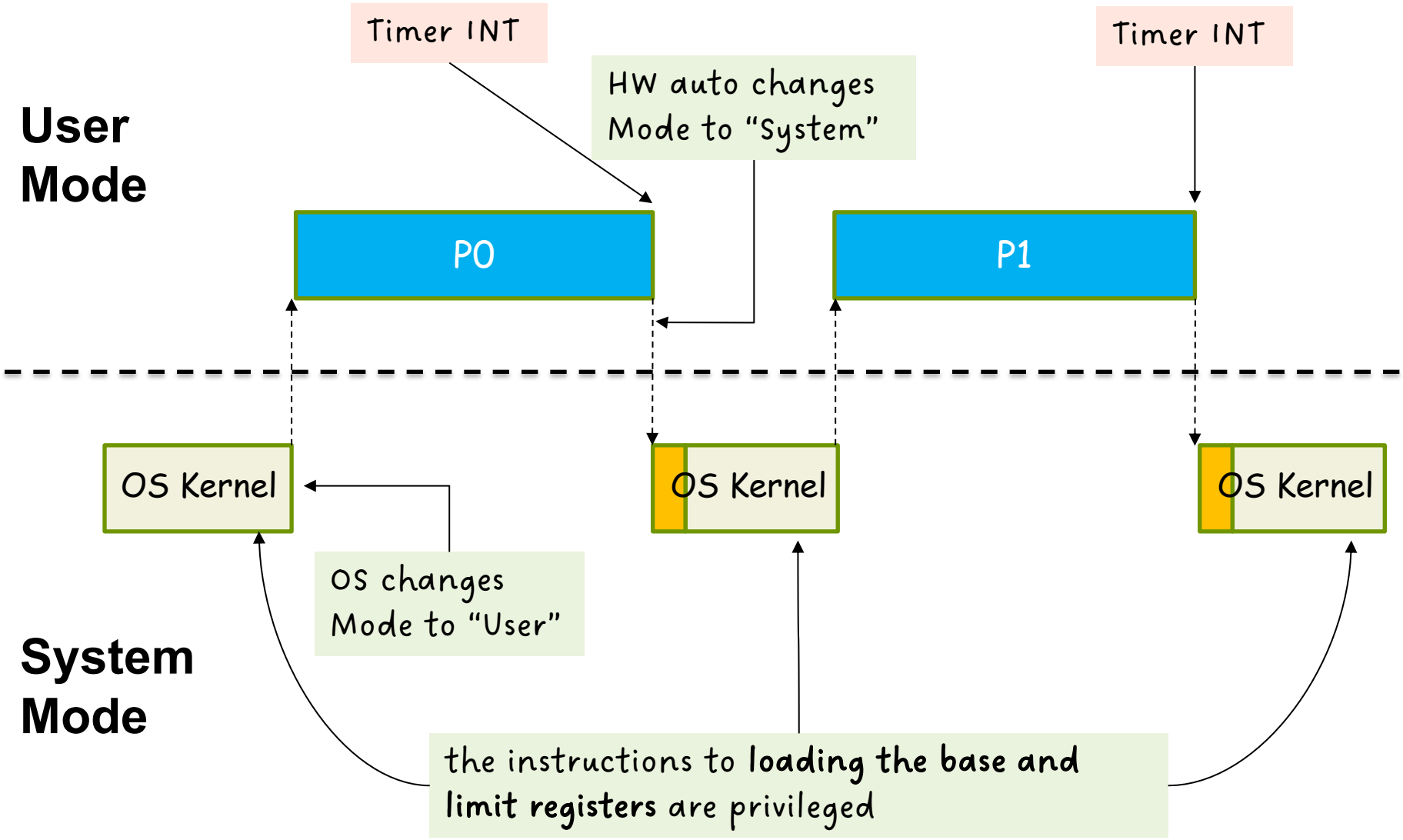
Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- the instructions to loading the base and limit registers are privileged
  - CPU ต้องอยู่ใน System Mode เท่านั้น ถึงจะรันคำสั่ง Privileged Instruction ได้
  - CPU อยู่ใน System Mode เมื่อ OS Kernel ประมวลผล

CPU timeline

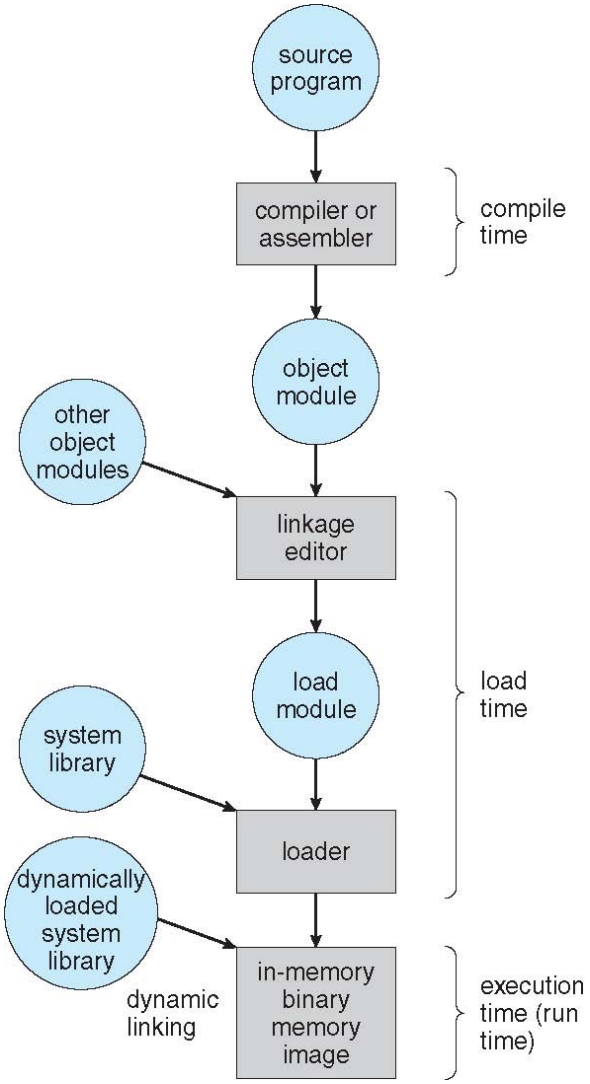


Address Binding

ระดับชั้นของ นามธรรม (abstraction)

- CPU ประมวลผลชุดคำสั่งแบบ binary code หรือชุดคำสั่งภาษาเครื่อง
- การออกคำสั่งด้วย binary code ทำได้ยากจึงเกิดการพัฒนาภาษาโปรแกรมขึ้น
- ภาษา Assembly ใช้สัญลักษณ์ขั้นพื้นฐาน
  - เราใช้ Assembler แปลงโปรแกรม Assembly ให้เป็น binary code
  - one statement ต่อ 1 machine instruction
- จากโปรแกรม C, C++, Java และ python เป็น High-level language (HLL)
  - เราสามารถใช้ Compiler แปลงโปรแกรมให้เป็น assembly code
  - และใช้ assembler แปลงโปรแกรมให้เป็น binary code
  - 1 statement มักจะถูกแปลงเป็น หลาย machine instruction
- นามธรรมของส่วนประกอบของโปรแกรมมีหลายระดับ

Multistep Processing of a User Program



ระดับชั้นของ นามธรรม (abstraction)

- ในภาษา มีการให้ตัวตนกับสรรพสิ่ง ถ้าสิ่งนั้นมีตัวตน ก็ต้องมีนาม (สัญลักษณ์)
- การให้ชื่อเกิดขึ้นในต่างระดับของนามธรรม

ตัวอย่าง	สัญลักษณ์ภาษาชั้นสูง	สัญลักษณ์ภาษา assembly (ตย. RISC-V, MIPS)
1	int x, y, z;	ตำแหน่งหน่วยความจำ (address) ที่ 01014 (x ใช้พื้นที่จากนั้น 4 bytes) ตำแหน่งหน่วยความจำ (address) ที่ 01018 ตำแหน่งหน่วยความจำ (address) ที่ 01022
2	int f1(void){...}	ตำแหน่งหน่วยความจำ (address) ที่ 01234
3	z = x+y;	lw t0 0(01014) lw t1 0(01018) add t2 t0 t1 sw t2 0(01022)
4	f1();	jal 01234

Address Binding

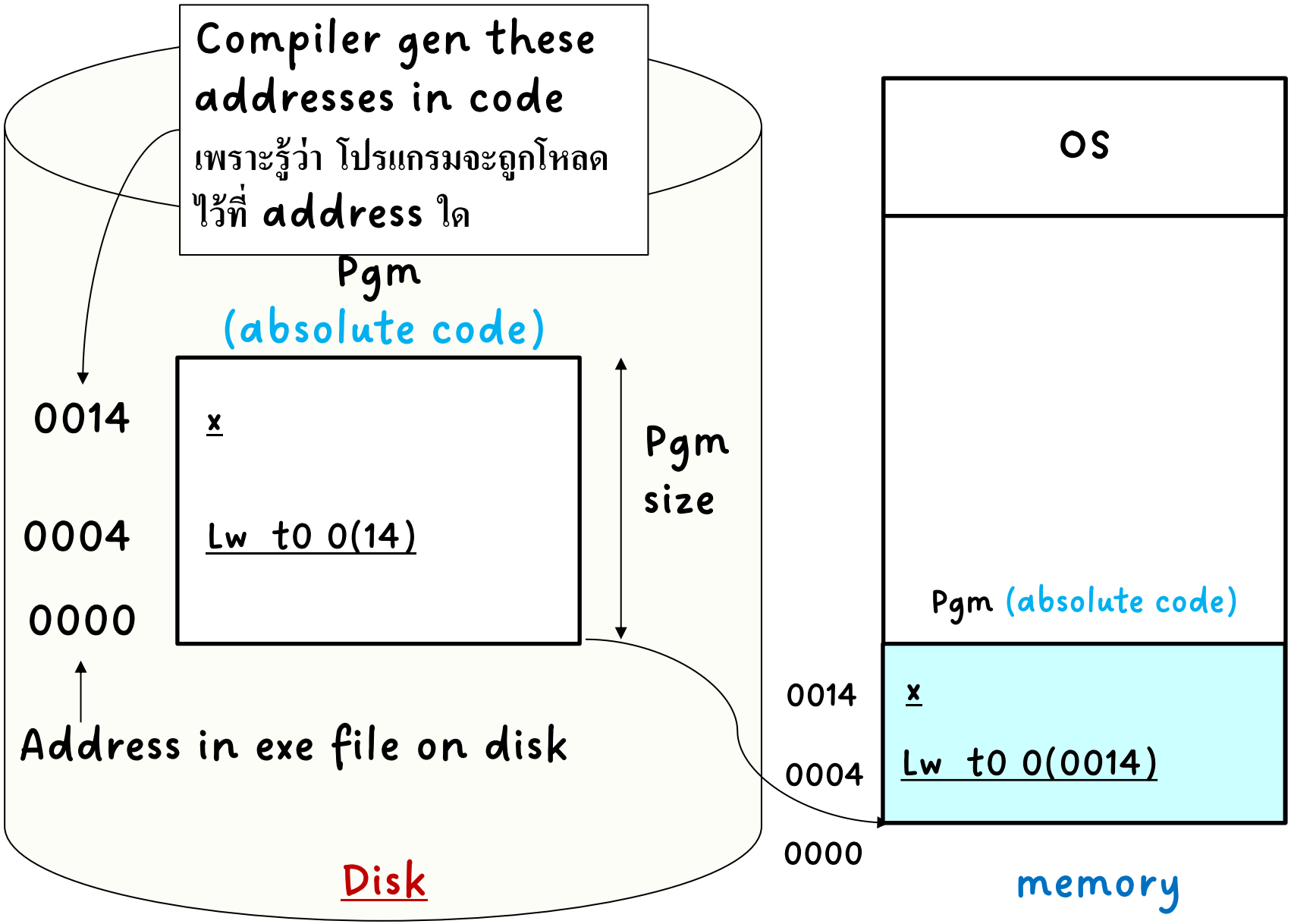
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will **bind** relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- ในการรัน Process เนื้อหาของโปรแกรมจะต้องอยู่ใน main memory
- binding จาก Address in exe file on disk → Address in memory
- ทำได้สามตอน: ตอนคอมไฟล์โปรแกรม ตอนโหลดโปรแกรม และตอน รันโปรแกรม
- *หมายความว่า คุณจะรู้ค่า physical mem addr ของ ชื่อของตัวแปร หรือคำสั่งใน โปรแกรมของคุณตอนไหน* ตอนคอมไฟล์ ตอนโหลด หรือตอนรันโปรแกรม

- Address binding of instructions and data to 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
  - ต้องรู้ memory address ตั้งแต่ตคอมไฟล์โปรแกรม
  - ใช้ในโปรแกรม MSDOS ในอดีต

ตัวอย่าง compile time binding



Binding of Instructions and Data to Memory

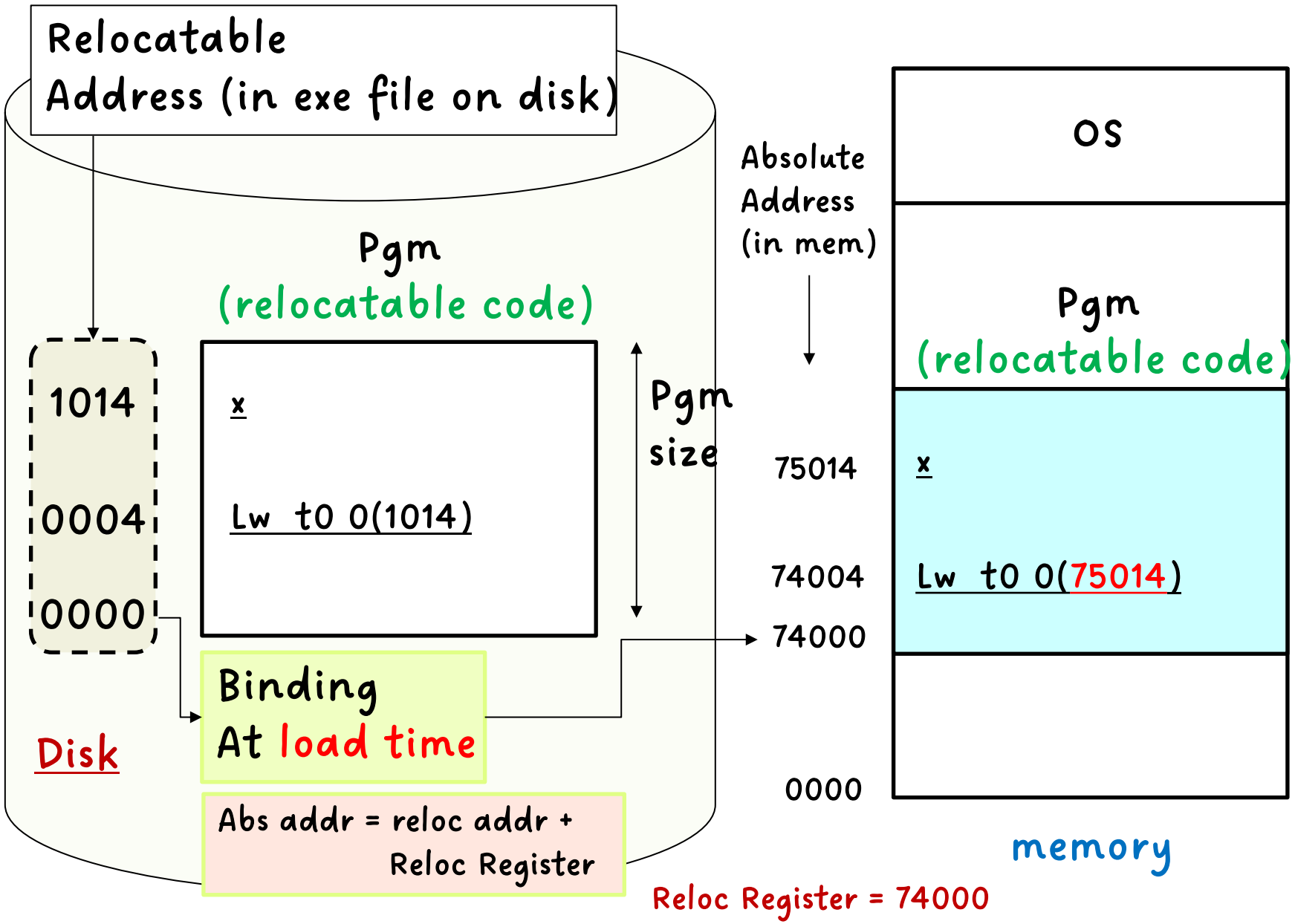
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - การกำหนดค่า address จริง จะเกิดตอน โหลดโปรแกรมเข้าสู่ memory
  - ถ้า starting address เปลี่ยนก็ต้อง reload โปรแกรมไหม่ ให้ค่า address ของส่วนประกอบของโปรแกรมเปลี่ยนตาม starting address ที่เปลี่ยนไป

relocatable address

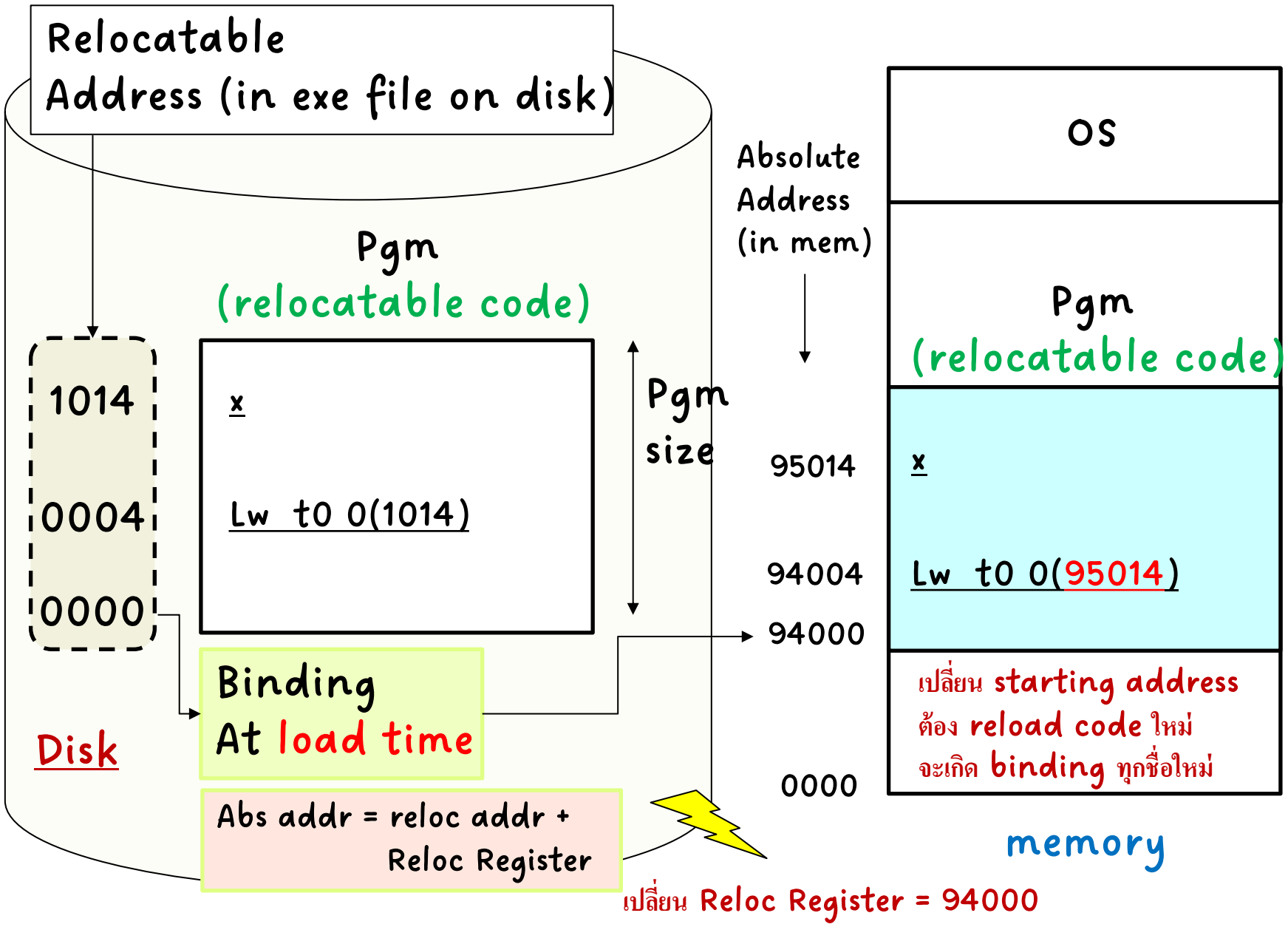
- ชื่อของส่วนประกอบของโปรแกรม ในระดับ Assembly และ Machine Code ล้วนแต่เป็น relocatable address
- compiler จะแปลงทุกชื่อในภาษาชั้นสูงให้เป็น “relocatable address” เหล่านี้ในไฟล์ executable
- เมื่อโปรแกรมประมวลผล โปรแกรมจะต้องถูกโหลดเข้ามาใน Hardware ที่ใช้เก็บข้อมูลแบบ array อุปกรณ์หนึ่งที่เรียกว่า Main Memory M[n] ที่ประกอบไปด้วย i คือ index และ M[i] คือค่าจำนวน 1 bytes จำนวน n ค่า

- การแปลง “relocatable address” ให้เป็นค่า i คือ Address Binding

ตัวอย่าง load time binding



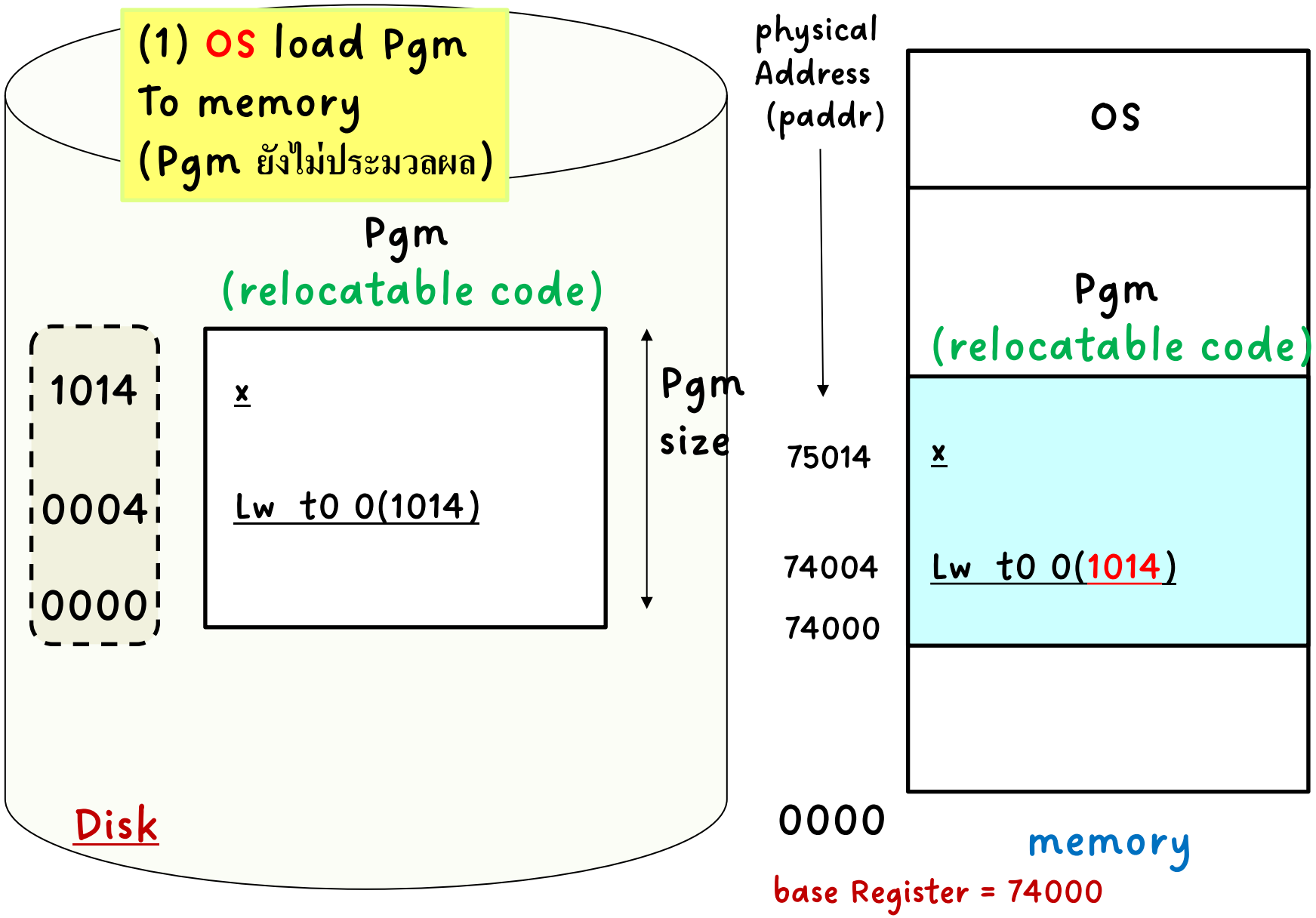
ตัวอย่าง load time binding (reload)



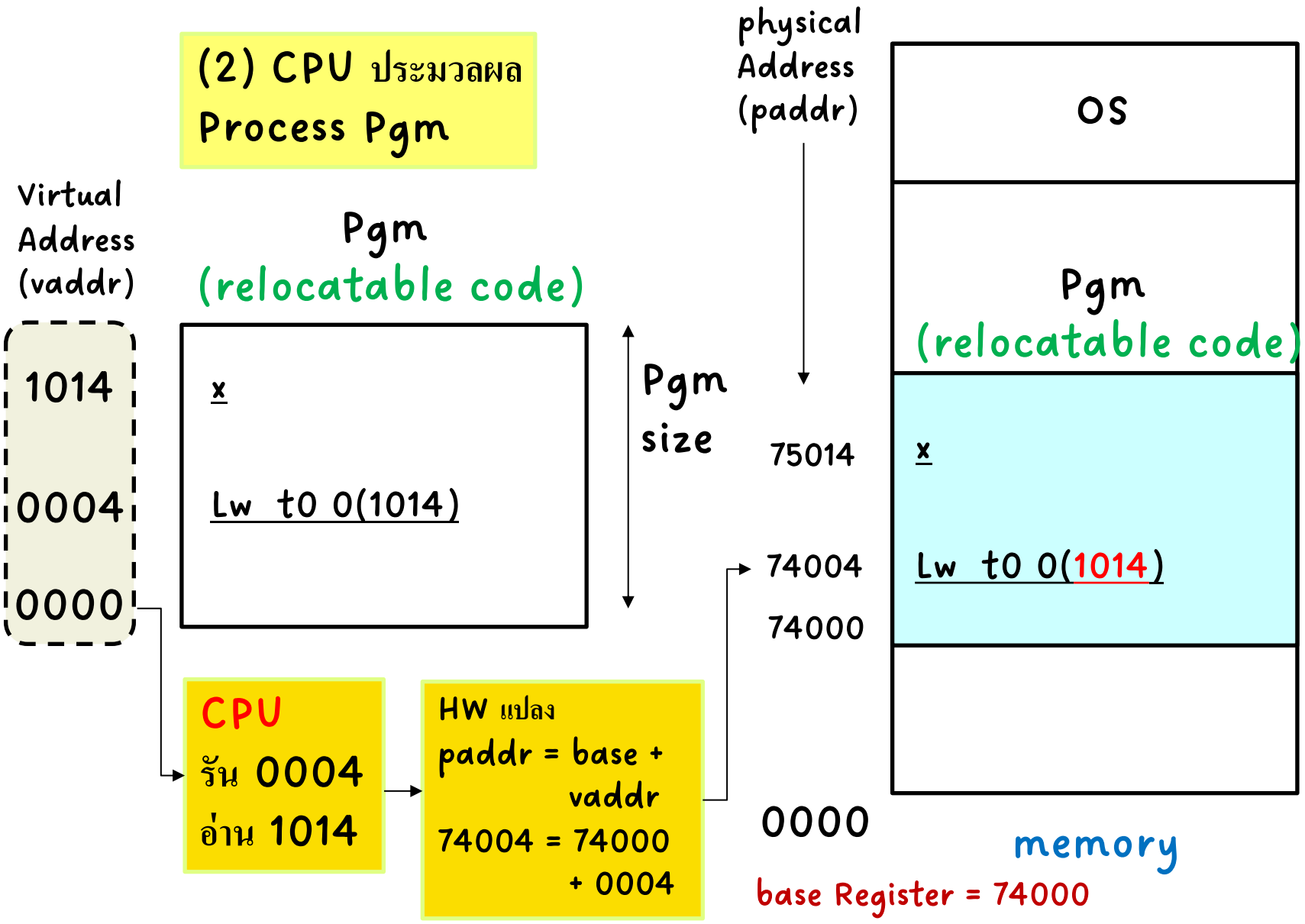
Binding of Instructions and Data to Memory

- Address binding of instructions and data to 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
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)
  - การกำหนดค่า address จริง จะเกิดตอน โปรแกรมรัน แต่ละคำสั่ง
  - ในเรื่องของ “address” Process แต่ละ Process จะมีโลกเสมือน เป็นของตนเอง แยกจาก Process

ตัวอย่าง execution time binding (1)

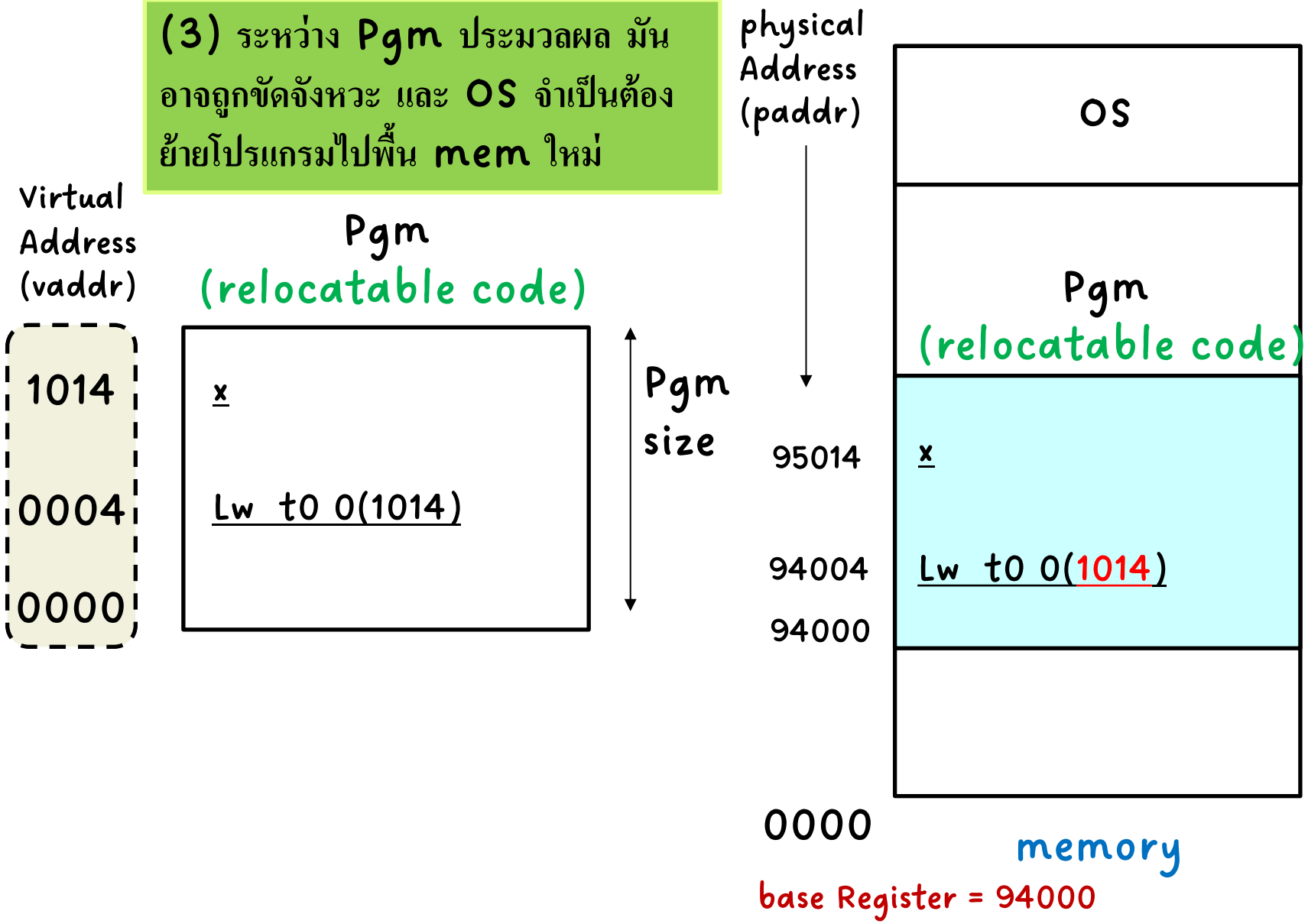


ตัวอย่าง execution time binding (2)

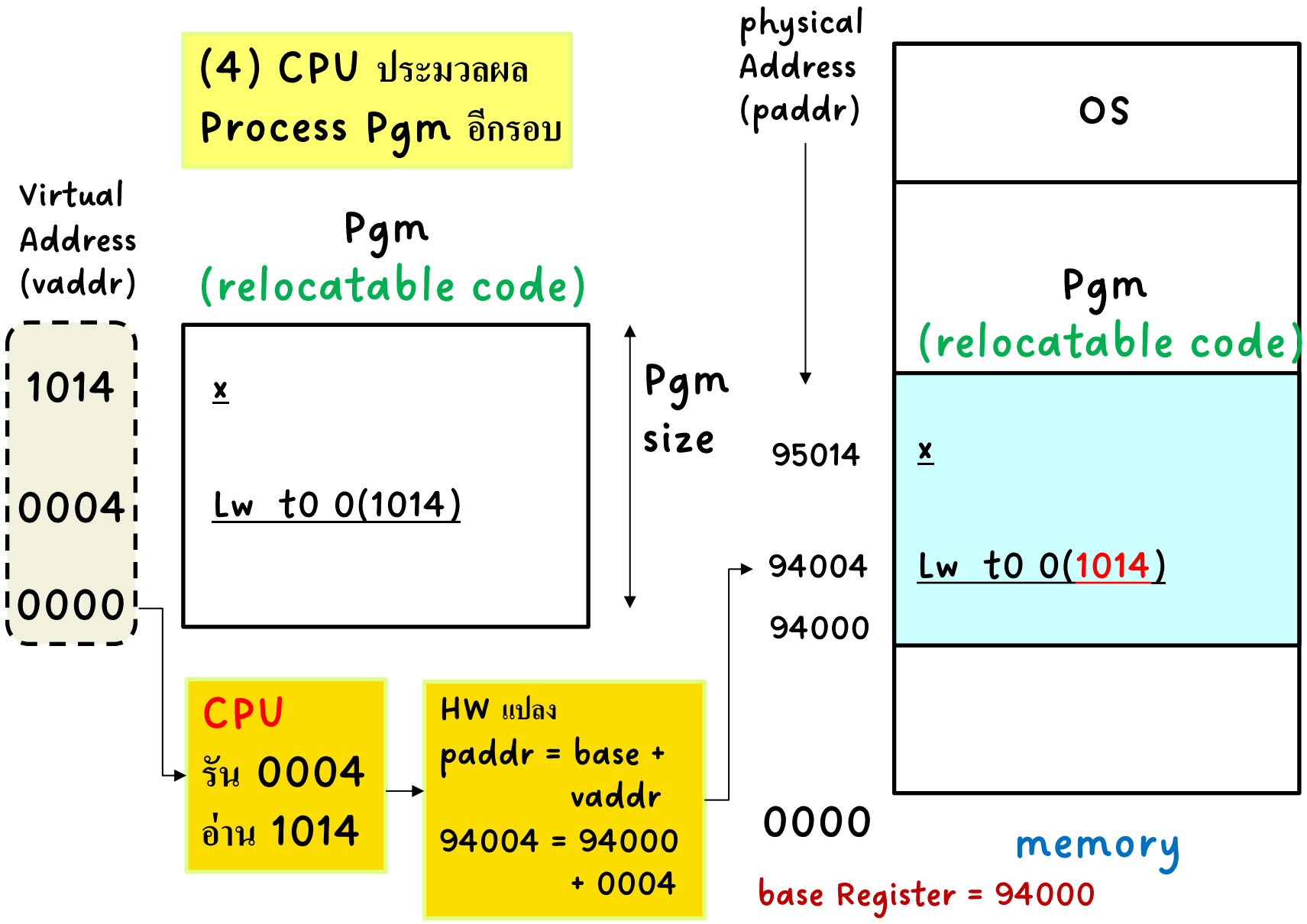


ตัวอย่าง execution time binding (3)

(3) ระหว่าง Pgm ประมวลผล มันอาจถูกขัดจังหวะ และ OS จำเป็นต้องย้ายโปรแกรมไปพื้น mem ใหม่



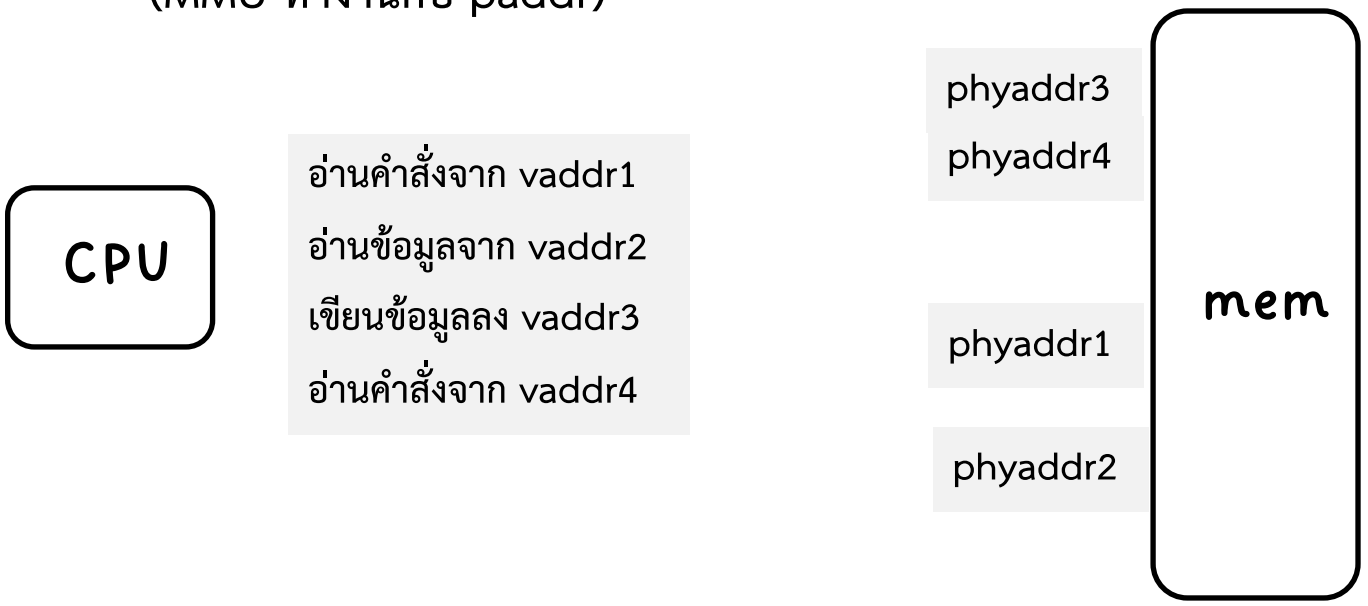
ตัวอย่าง execution time binding (4)



Logical vs. Physical Address Space

- ซีพียู อ่านคำสั่งและข้อมูลจาก Memory เข้ามารัน
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address** (ซีพียูทำงานกับ vaddr)
  - **Physical address** – address seen by the memory unit

    (MMU ทำงานกับ paddr)

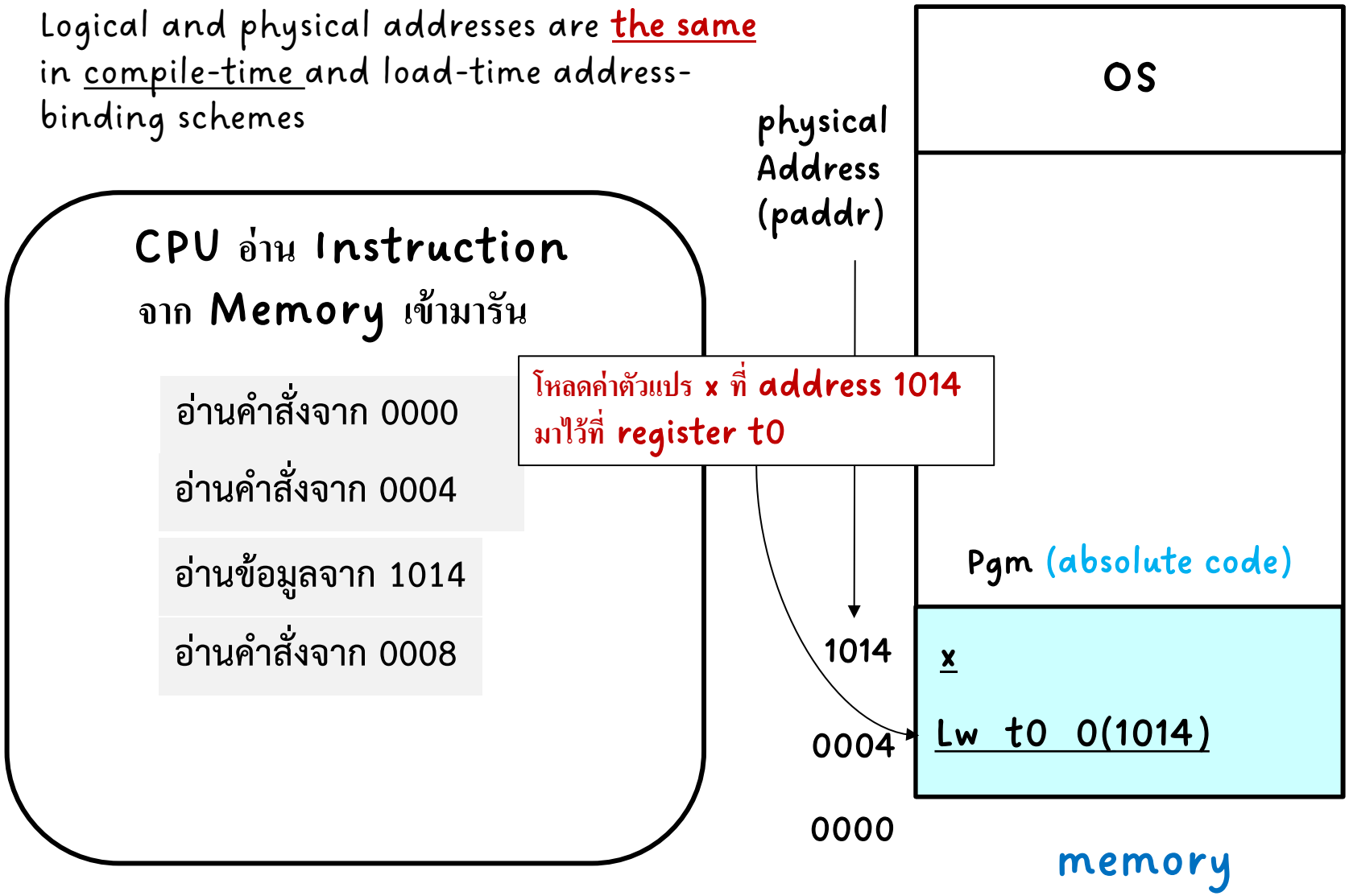


Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
  - หมายความว่า ซีพียูใช้ vaddr ค่าเดียวกันกับ psaddr ในกรณี compile-time & load time binding
  - ซีพียูใช้ vaddr ที่แตกต่างจาก paddr ในกรณี execution time binding

ตัวอย่าง compile time binding

Logical and physical addresses are the same in compile-time and load-time address-binding schemes



ตัวอย่าง load time binding

Logical and physical addresses are the same in compile-time and load-time address-binding schemes

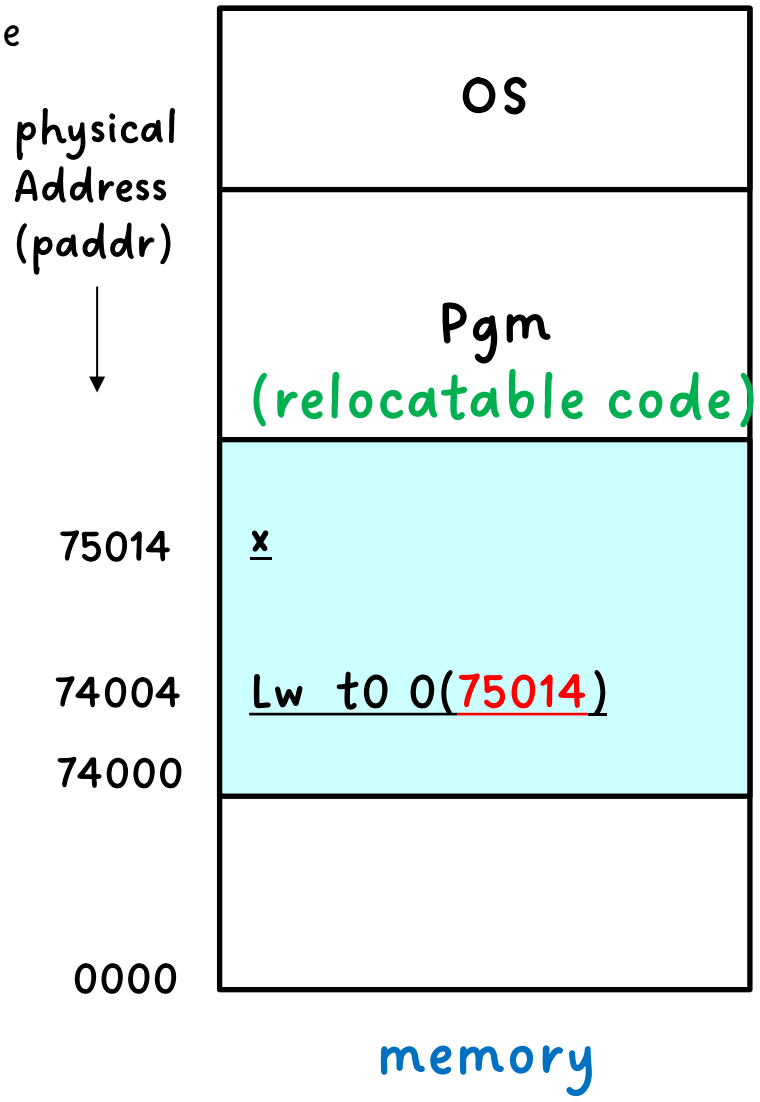
CPU อ่าน Instruction จาก Memory เข้ามารัน

อ่านคำสั่งจาก 74000

อ่านคำสั่งจาก 74004

อ่านข้อมูลจาก 75014

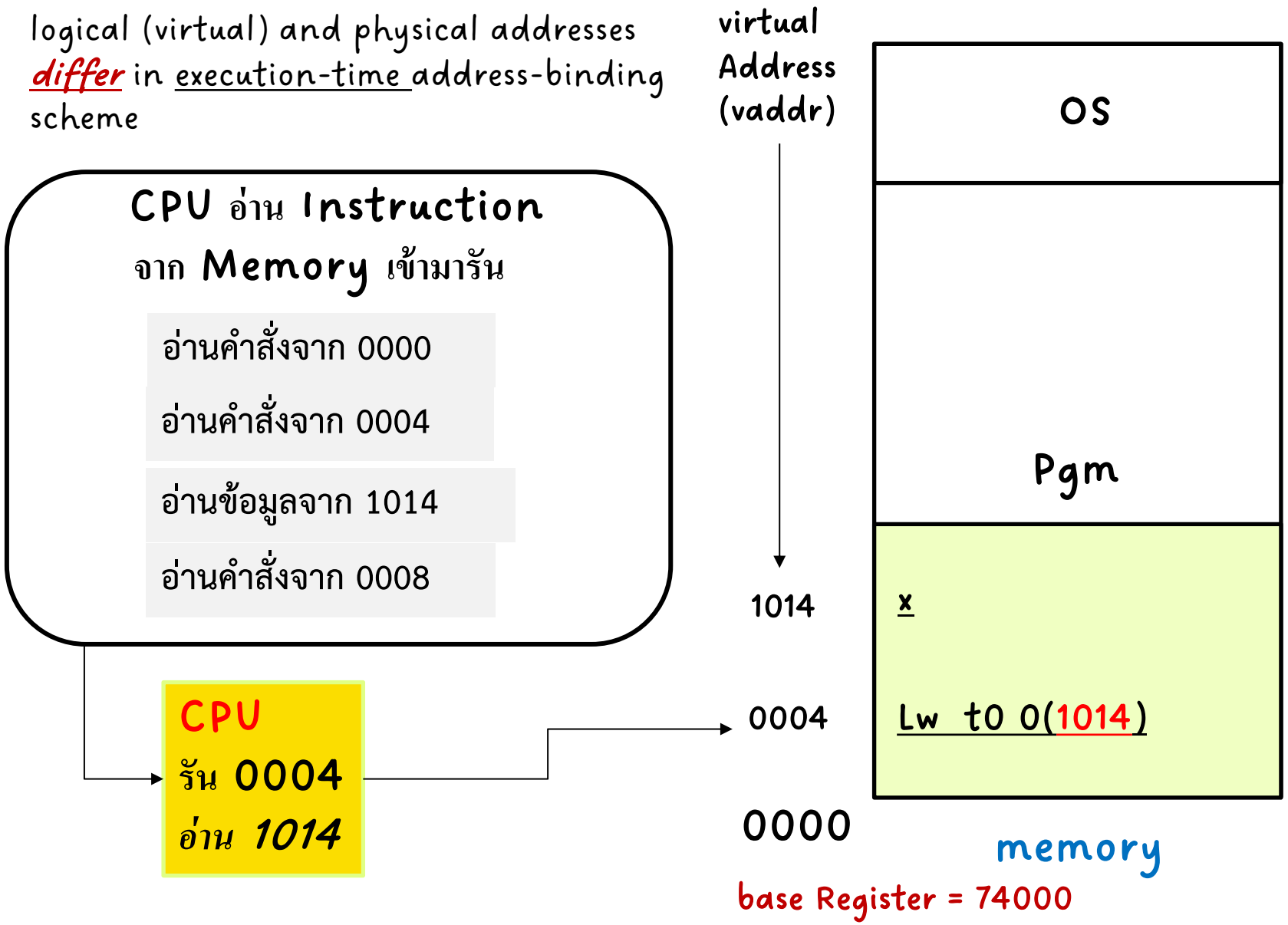
อ่านคำสั่งจาก 74008



Reloc Register = 74000

ตัวอย่าง execution time binding (1)

logical (virtual) and physical addresses *differ* in execution-time address-binding scheme



ตัวอย่าง execution time binding (2)

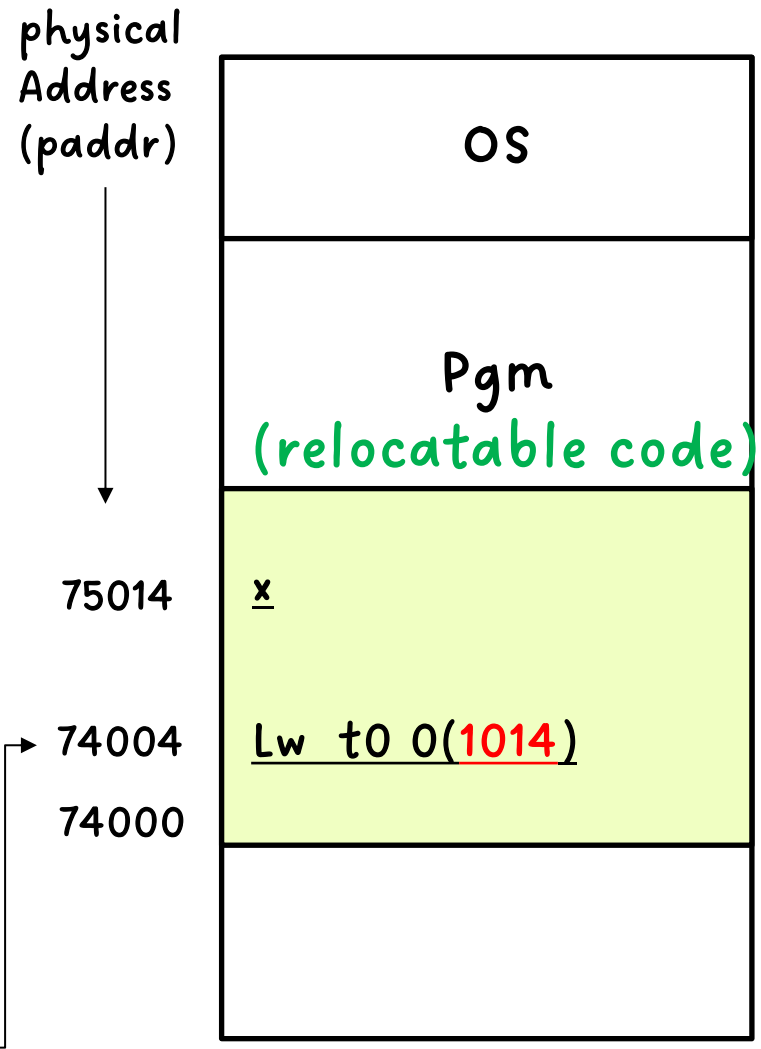
logical (virtual) and physical addresses *differ* in execution-time address-binding scheme

CPU อ่าน Instruction จาก Memory เข้ามารัน

- อ่านคำสั่งจาก 0000
- อ่านคำสั่งจาก 0004
- อ่านข้อมูลจาก 1014
- อ่านคำสั่งจาก 0008

CPU
รัน 0004
อ่าน 1014

HW แปลง
paddr = base + vaddr
74004 = 74000 + 0004



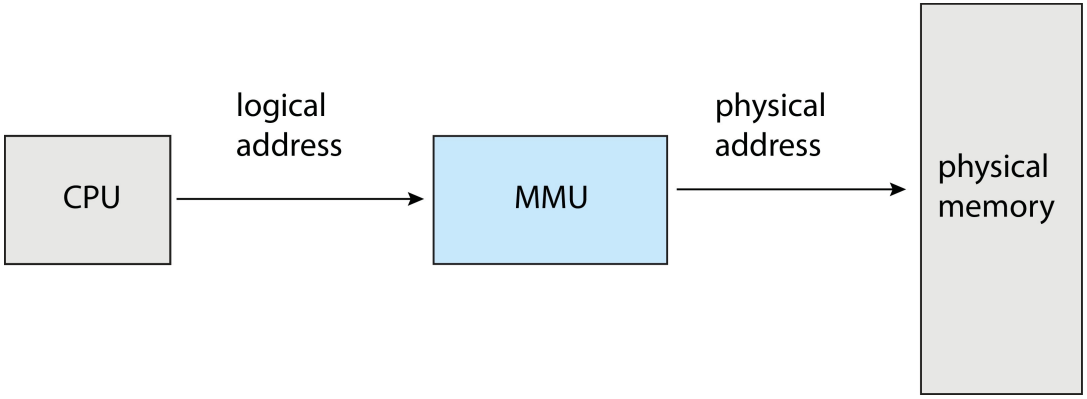
Logical Address และ Physical Address และ Memory Management Unit (MMU)

Logical vs. Physical Address Space

- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program
- ในปัจจุบัน OS และ HW ส่วนใหญ่เป็น Exe time binding
- ใช้ MMU ช่วย

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



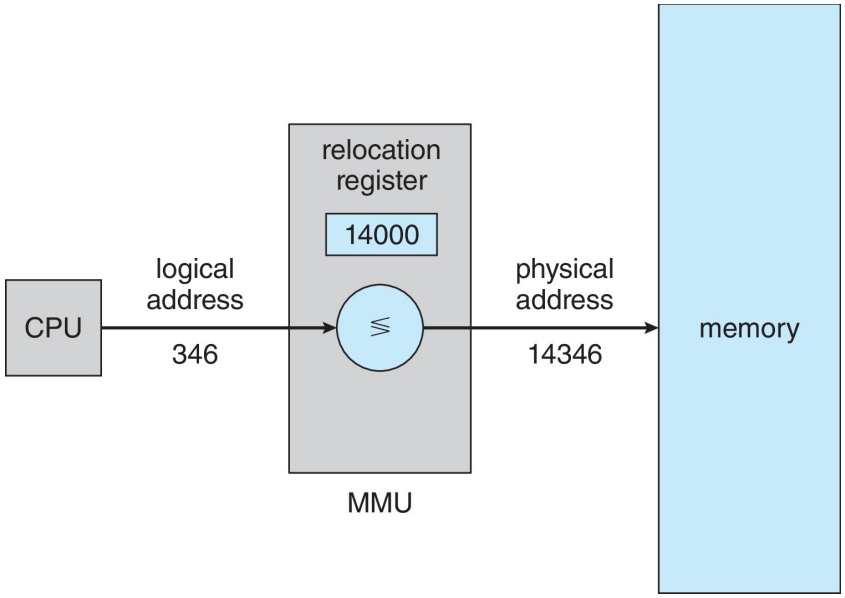
- Many methods possible, covered in the rest of this chapter

Memory-Management Unit (Cont.)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program (และ CPU) deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Memory-Management Unit (Cont.)

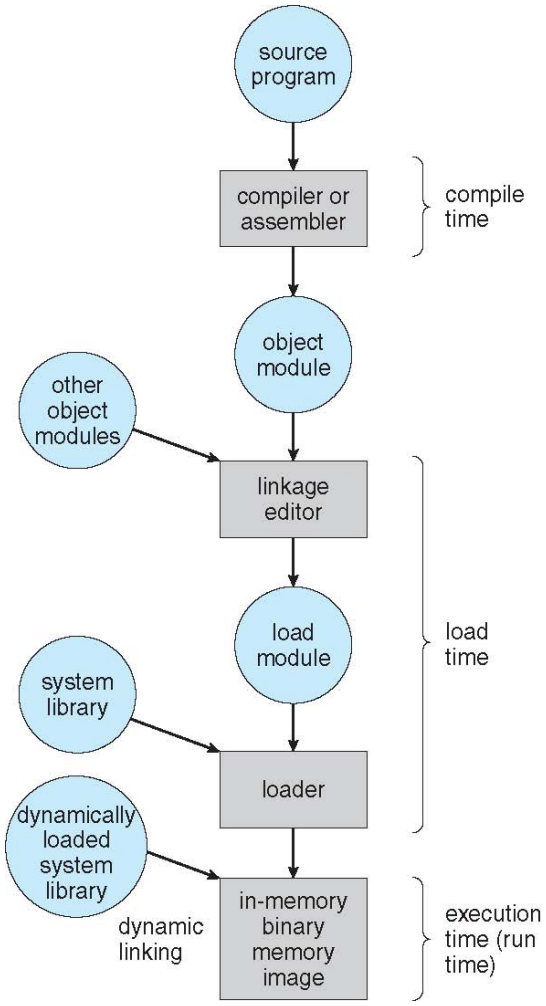
- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



การโหลดโปรแกรมเข้าสู่ Memory

- **Dynamic Loading**
- **Dynamic Linking**

Multistep Processing of a User Program



Dynamic Loading

- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

Dynamic Loading

- โปรแกรมต้องถูก load เข้ามาใน Memory เพื่อ Execute
- Routine จะไม่ถูก load จนกระทั่งโปรแกรมเรียกใช้ routine นั้น
- ประหยัดพื้นที่หน่วยความจำ routine ที่ไม่ถูกใช้ก็ไม่ถูกนำเข้ามาใน mem
- ทุก routine จะถูกเก็บใน disk/ssd ในรูปแบบ **relocatable** load format
- โหลดเฉพาะ routine และ dependency ของ routine นั้น ไม่ใช่ทั้ง library
- มีประโยชน์เพราะโปรแกรมอาจไม่ได้ใช้ routine บ่อยนักแต่เมื่อต้องใช้ก็โหลดได้
- ไม่ต้องใช้กลไกของ operating system (OS ไม่ใช่คนจัดการให้)
  - การโหลดเป็นส่วนหนึ่งของการออกแบบโปรแกรม เช่นเมื่อผู้ใช้เลือก option อันใดอันหนึ่งใน menu ก็ค่อยโหลด
  - OS อาจให้บริการ libraries หรือเครื่องมือเพื่อให้โปรแกรมเรียกใช้ routine ใน libraries เพื่อทำ dynamic loading ด้วยตัวโปรแกรมเอง

Dynamic Loading (wikipedia)

- ใช้ใน IBM mainframe ตั้งแต่ IBM System/360 ใน OS/360
- ในเป็น plugin ของ Apache Web Server *.dso เป็นไฟล์ plugin แบบ “dynamic shared object”
- ใน UNIX-like OS และ Windows ใน C/C++ ก็มี dl library ให้โปรแกรมเรียกใช้เพื่อ โหลด dynamic library หรือ shared library เมื่อต้องการ

Name	Standard POSIX/UNIX API	Microsoft Windows API
Header file inclusion	`#include <dlfcn.h>`	`#include <windows.h>`
Definitions for header	`dl` (`libdl.so` , `libdl.dylib` , etc. depending on the OS)	`kernel32.dll`
Loading the library	`dlopen`	`LoadLibrary` `LoadLibraryEx`
Extracting contents	`dlsym`	`GetProcAddress`
Unloading the library	`dlclose`	`FreeLibrary`

https://en.wikipedia.org/wiki/Dynamic_loading

ตัวอย่าง Dynamic Loading (wikipedia)

- UNIX

Most UNIX-like operating systems (Solaris, Linux, *BSD, etc.) [edit]

```
void* sdl_library = dlopen("libSDL.so", RTLD_LAZY);
if (sdl_library == NULL) {
   // report error ...
} else {
   // use the result in a call to dlsym
}
```

- โหลดเฉพาะ routine และ dependency ของ routine นั้น ไม่ใช่ทั้ง library

- Windows

Windows [edit]

```
HMODULE sdl_library = LoadLibrary(TEXT("SDL.dll"));
if (sdl_library == NULL) {
   // report error ...
} else {
   // use the result in a call to GetProcAddress
}
```

- โหลดเฉพาะ routine และ dependency ของ routine นั้น ไม่ใช่ทั้ง library

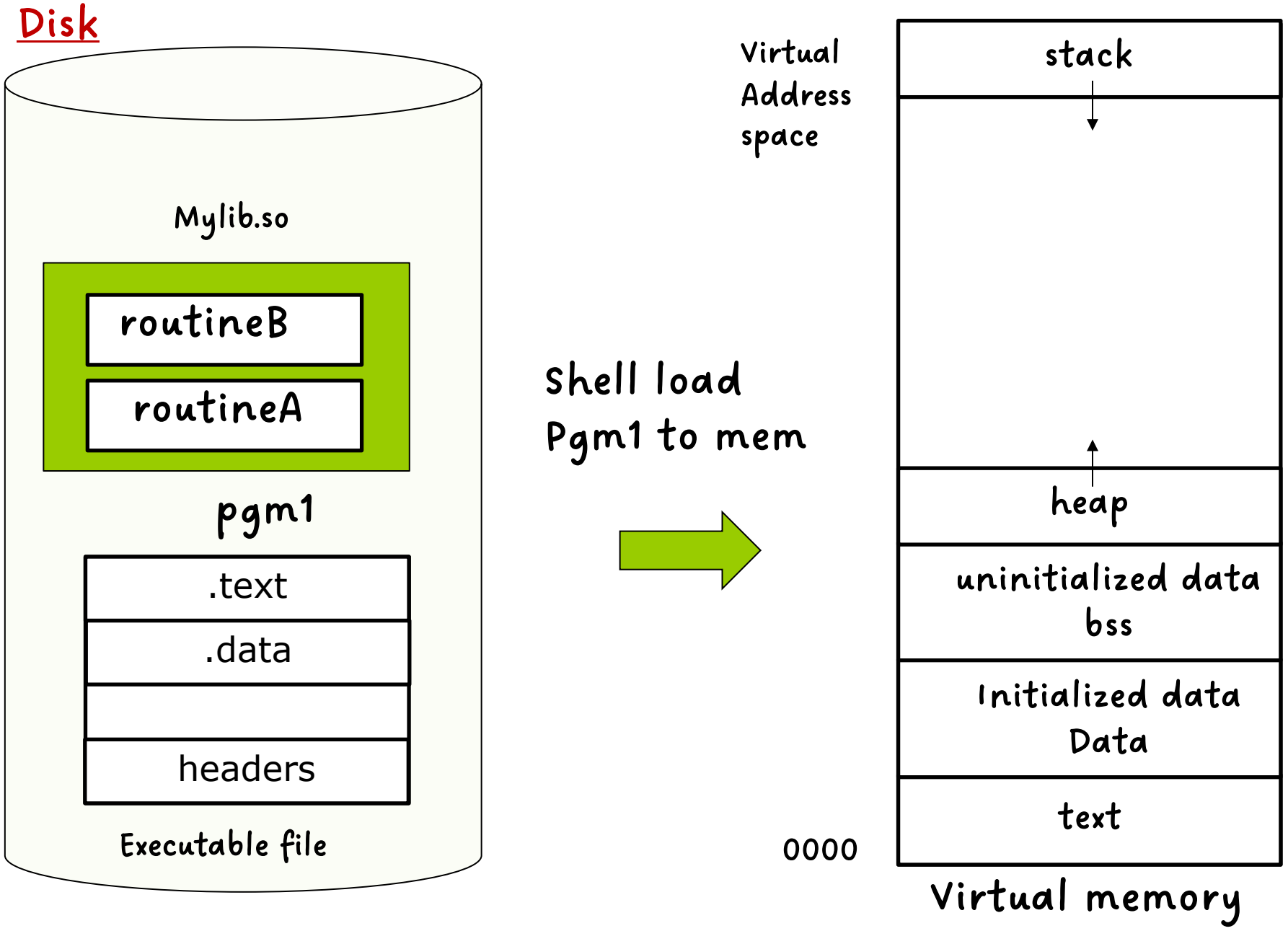
- java

```
Class type = ClassLoader.getSystemClassLoader().loadClass(name);
Object obj = type.newInstance();
```

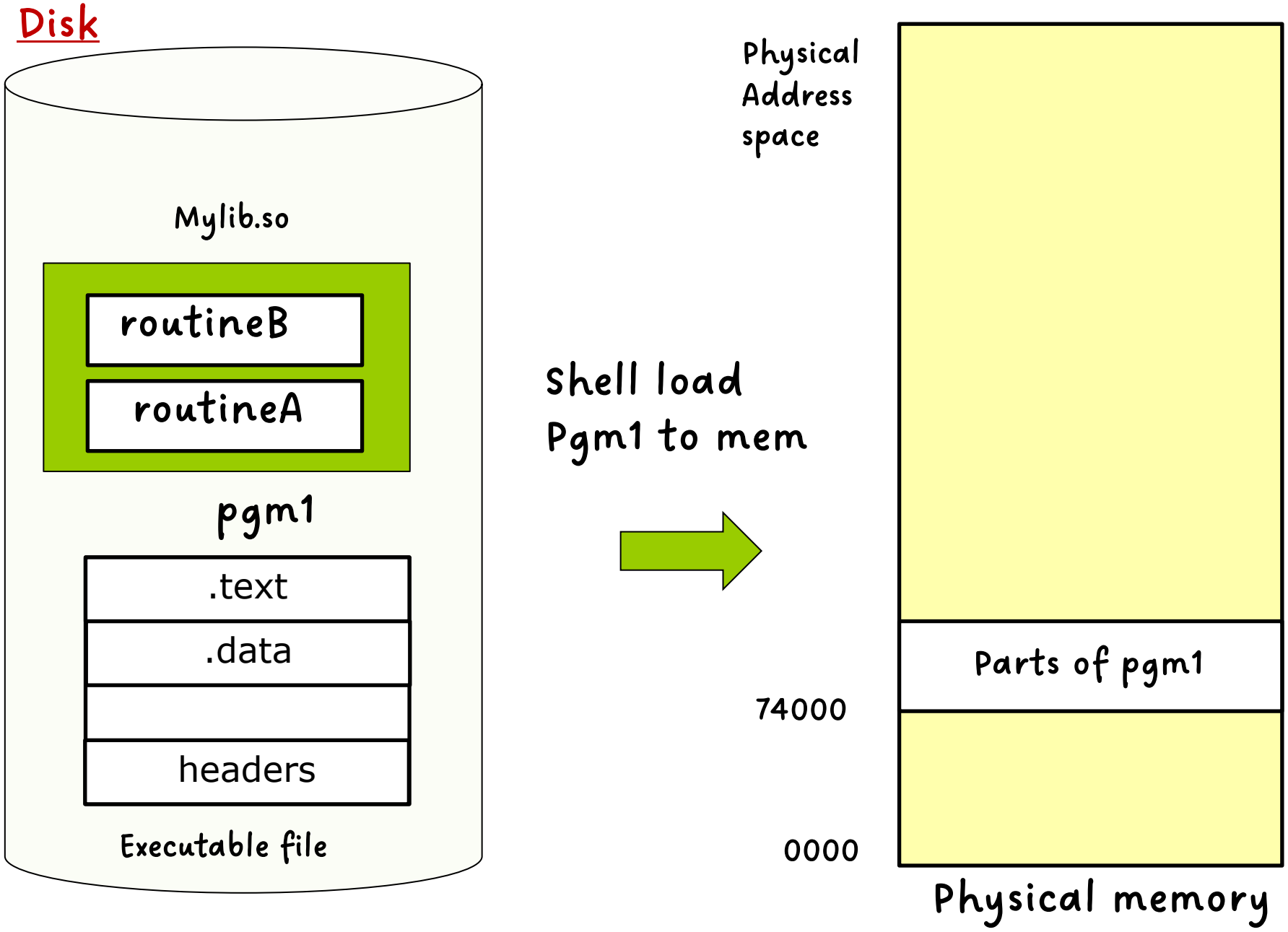
```
Class type = Class.forName(name);
Object obj = type.newInstance();
```

https://en.wikipedia.org/wiki/Dynamic_loading

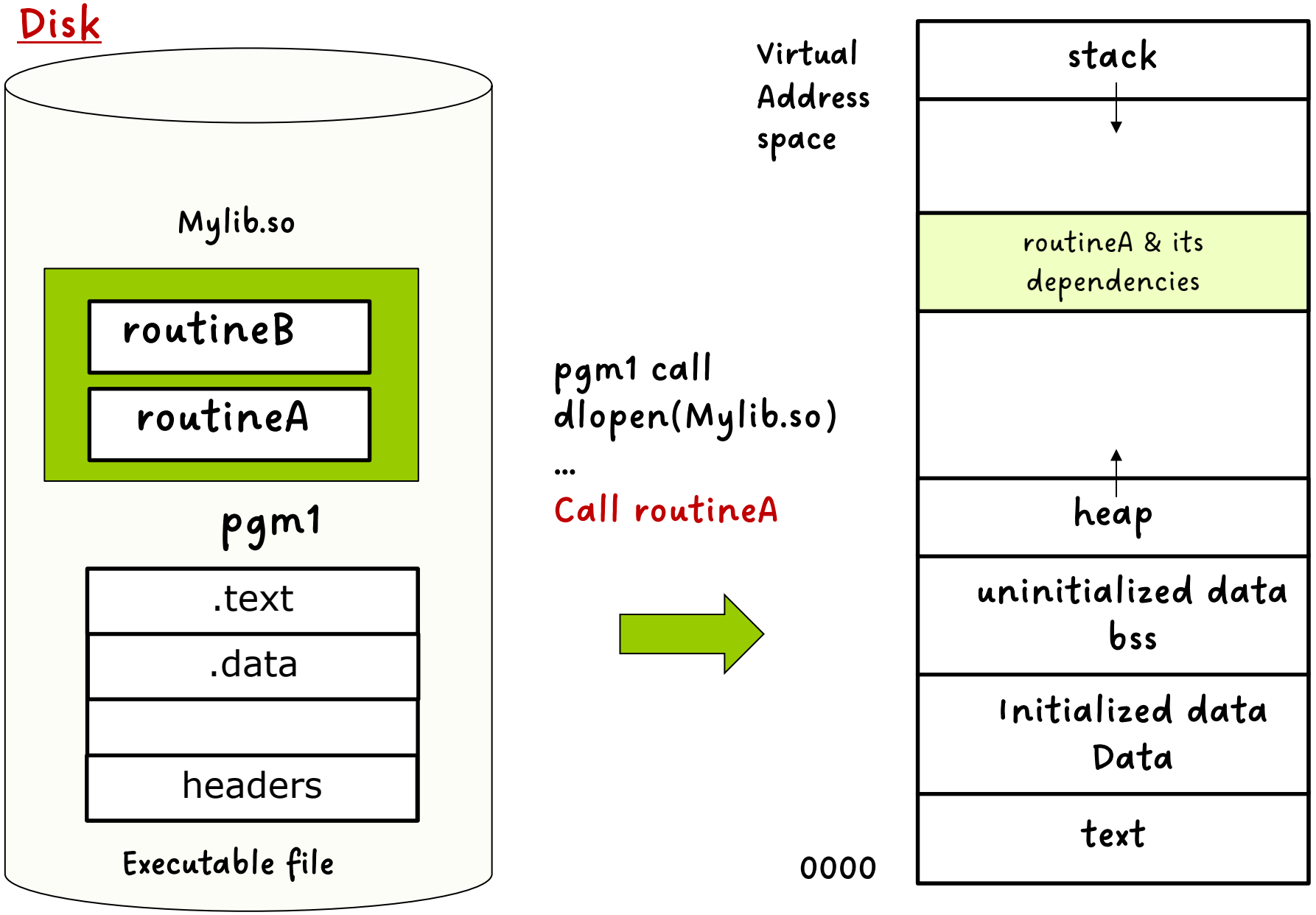
ตัวอย่าง dynamic loading



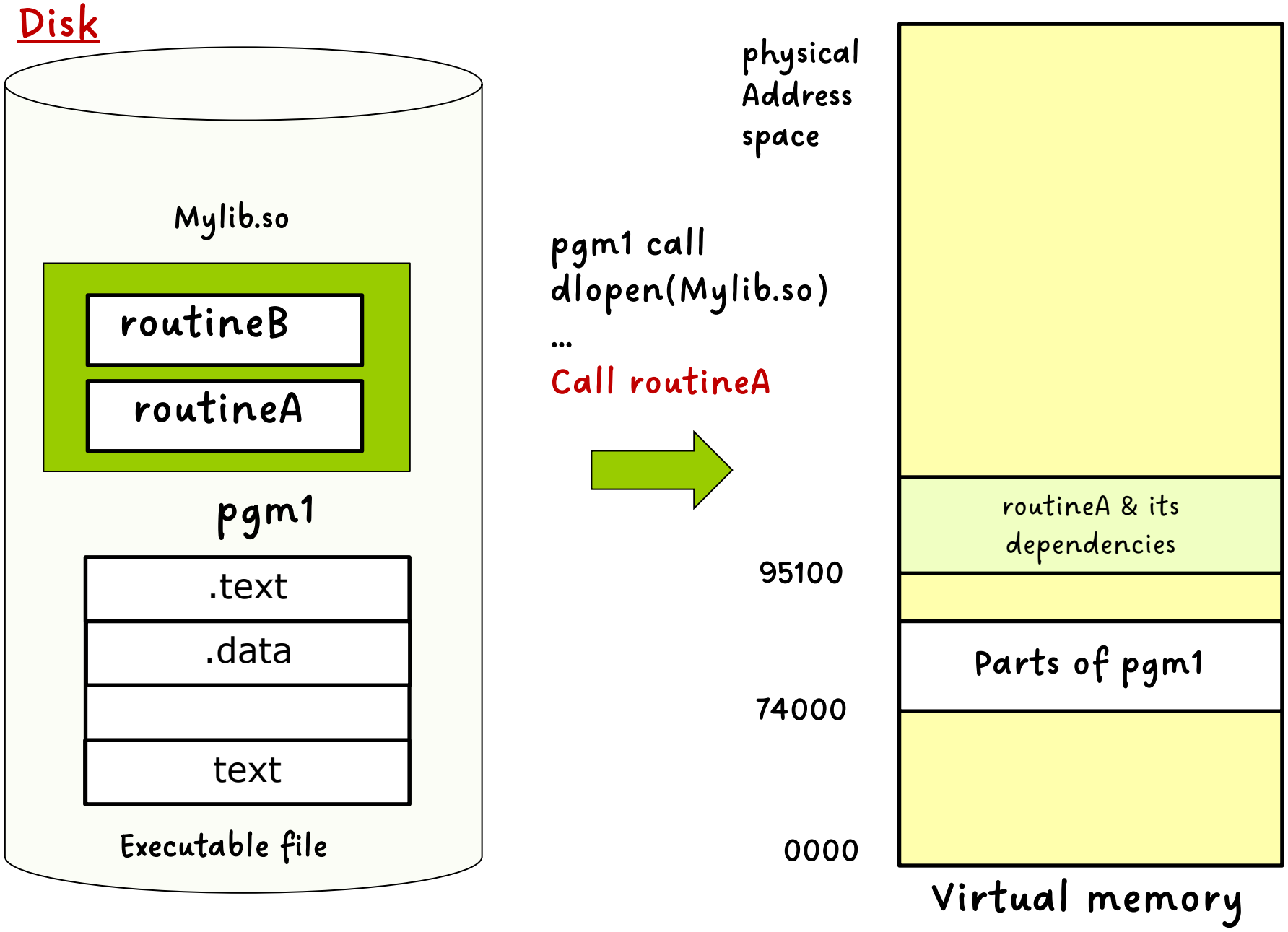
ตัวอย่าง dynamic loading



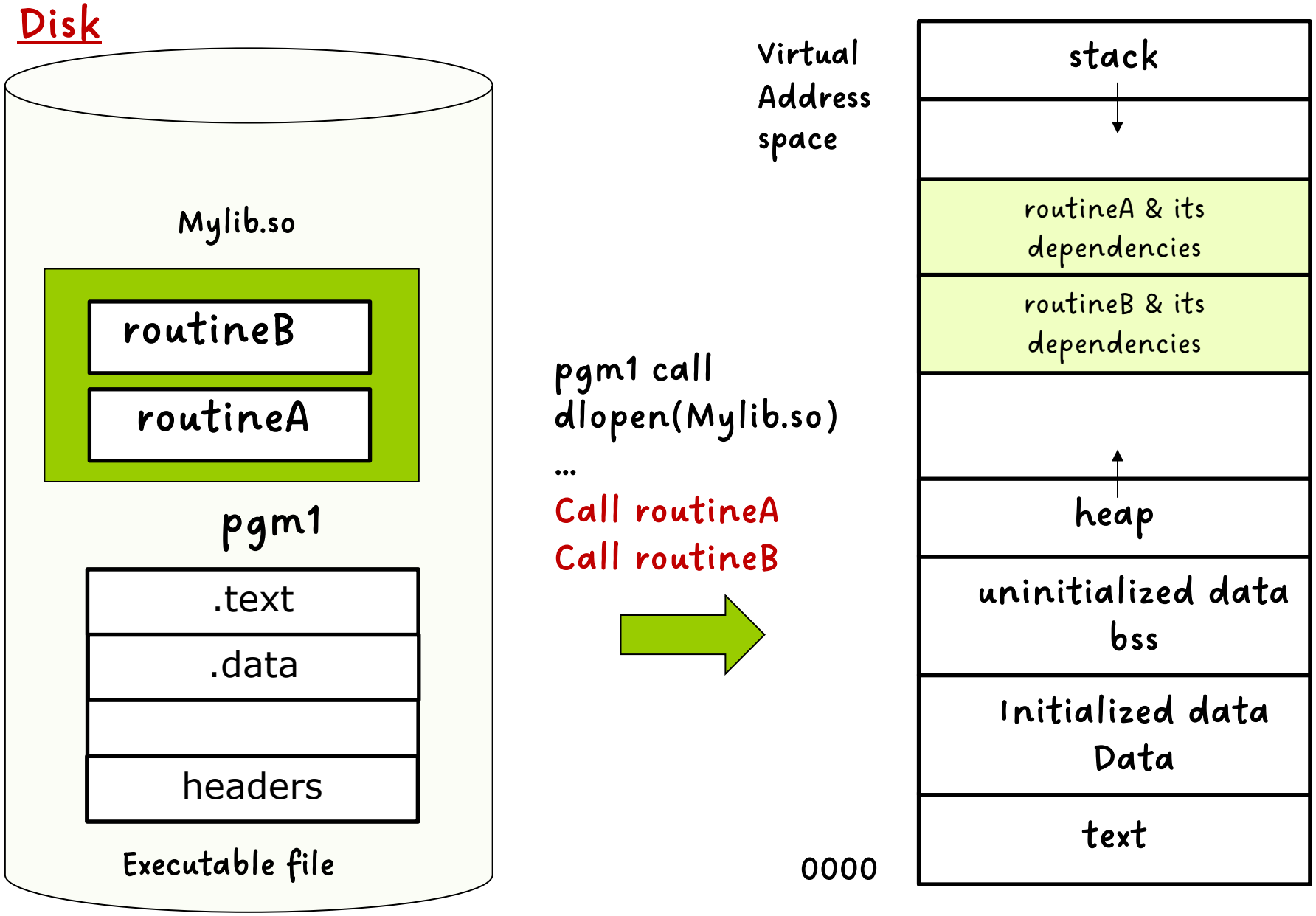
ตัวอย่าง dynamic loading



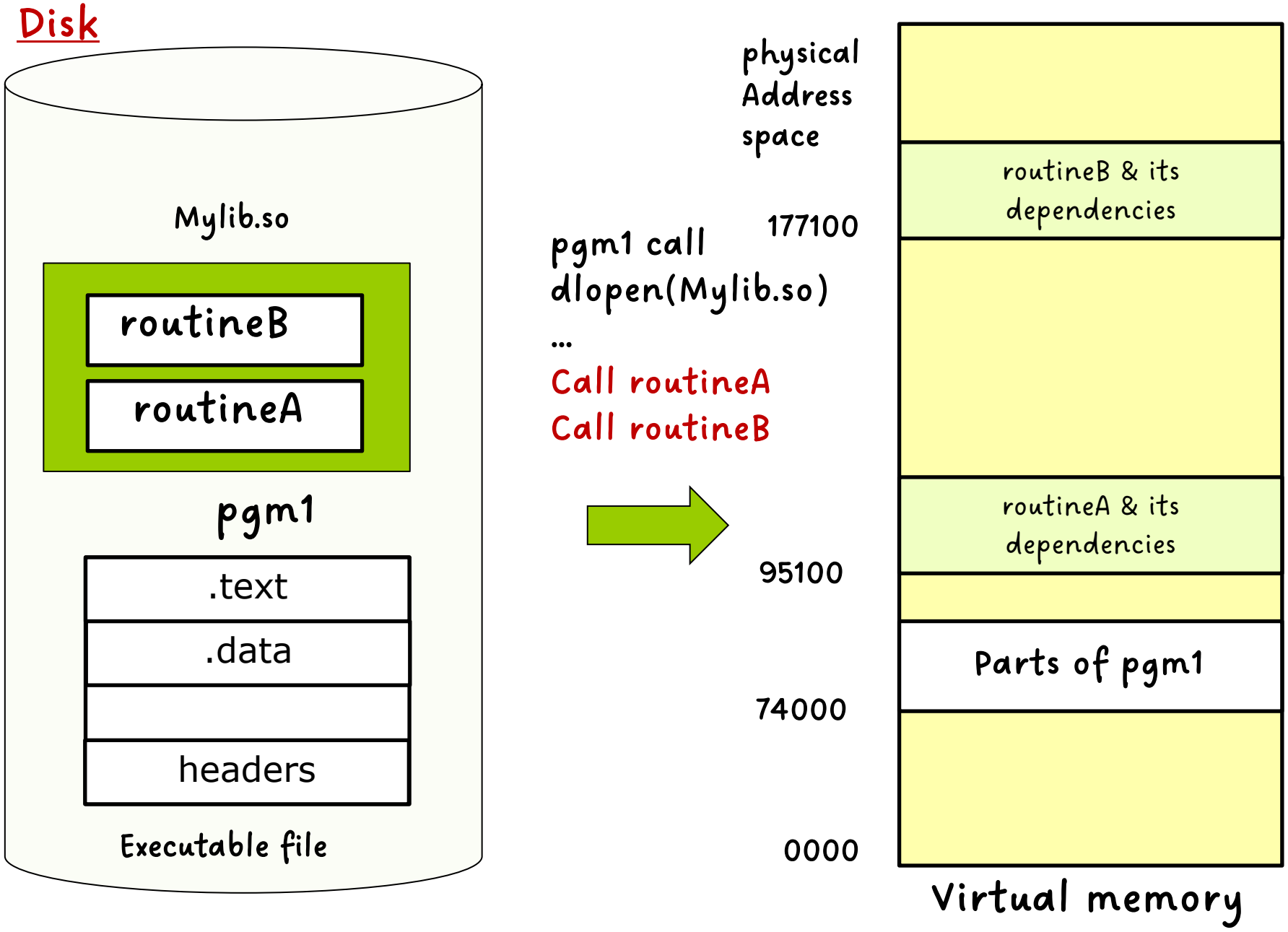
ตัวอย่าง dynamic loading



ตัวอย่าง dynamic loading



ตัวอย่าง dynamic loading



Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **Dynamic linking** –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

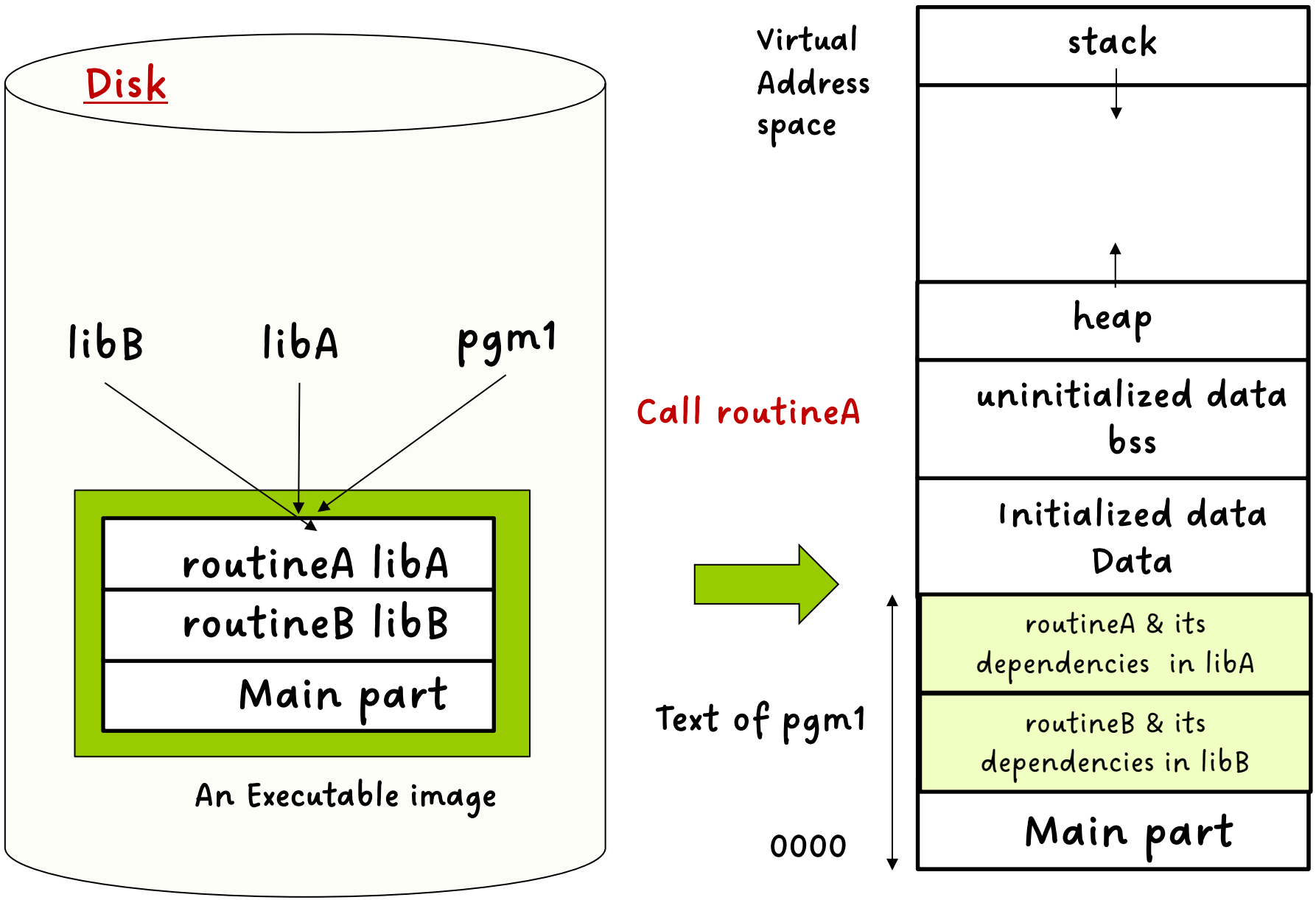
Dynamic Linking

- Static linking คอมไพเลอร์จะรวมโค๊ดของ routines ที่มันต้องใช้พร้อมทั้ง dependencies ของ routine เหล่านั้นใน library ที่ statically link เข้ามาในโปรแกรม executable code
- Dynamic linking จะ link routines ที่ต้องการเข้ามาตอน execution time
- คอมไพเลอร์จะใส่ routine ปลอมที่ไม่มีเนื้อหาจริงเรียกว่า stub เข้าไปใน executable code
- เมื่อมีการเรียกใช้ routine ที่เป็น stub OS จะเช็คว่า มี routine นี้ใน (virtual) memory ของ process หรือไม่ ถ้ามีก็จะแทน stub ด้วย (virtual) memory address ของ routine แต่ถ้าไม่มี ก็จะเพิ่ม routine นั้นใน virtual memory ของ process
- OS เป็นผู้จัดการ load และ link dynamic linking library ให้

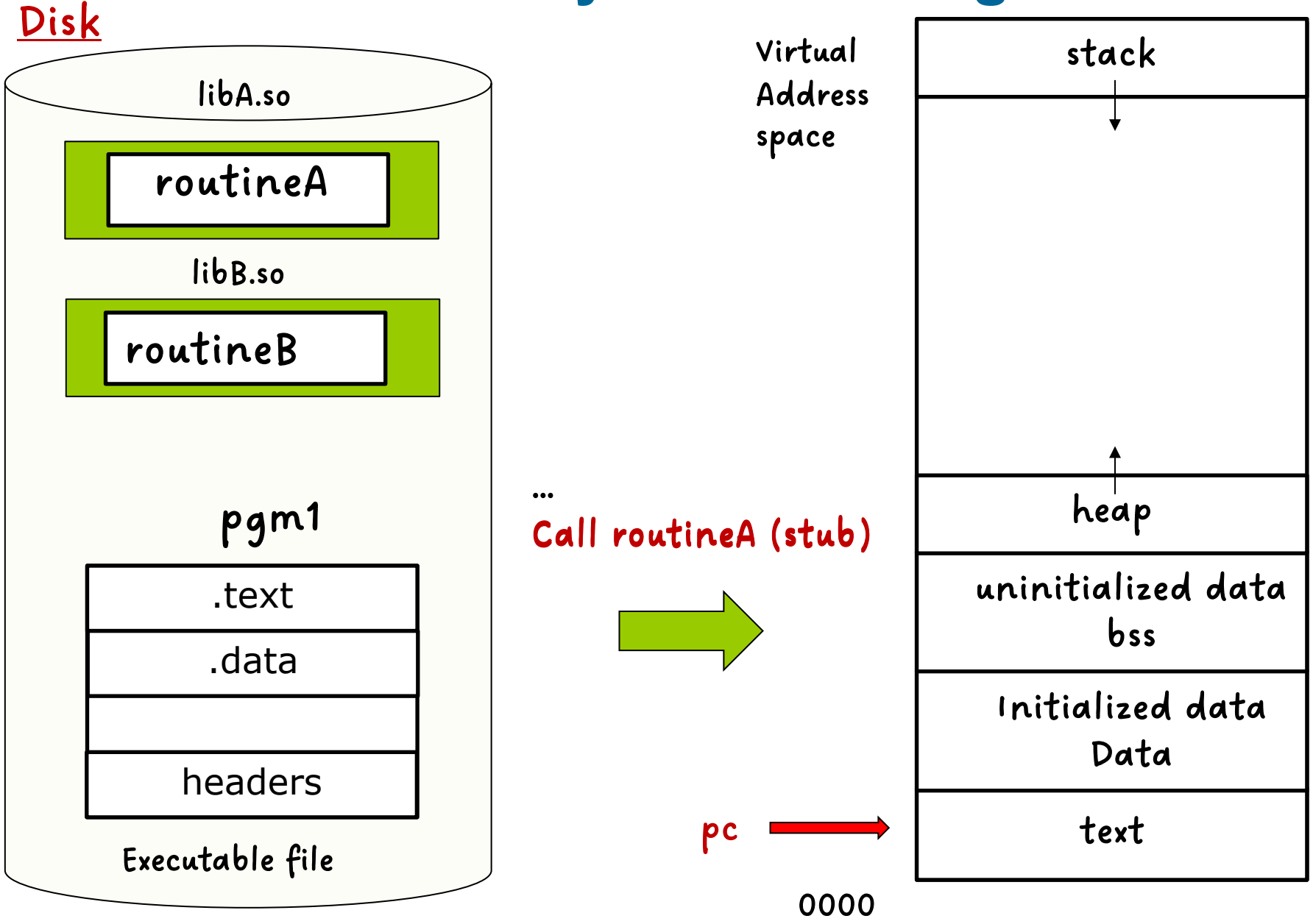
Dynamic Linking

- Dynamic linking มีประโยชน์มากเพราะช่วยประหยัดพื้นที่ของ executable
- library มักจะเป็น shared library
- Shared library เป็น dynamic library ที่ Process หลาย Process สามารถแชร์ร่วมกันได้ โดยโอเอสจะเป็นคนจัดการๆ share ให้
- ถ้า Process เรียกใช้ routine แล้ว OS พบว่า Process นั้นไม่มี routine นั้นใน process memory space OS ก็จะเช็คว่า routine นี้มีอยู่ใน memory space ของ Process อื่นหรือไม่ ถ้ามีมันจะขอแชร์ใช้ด้วย แต่ถ้าไม่มี OS จะโหลด routine นั้นให้ Process นั้น
- ต้องมีการระบุเวอร์ชั่นของ shared library ที่ process ด้วยเพื่อความถูกต้องในการประมวลผล

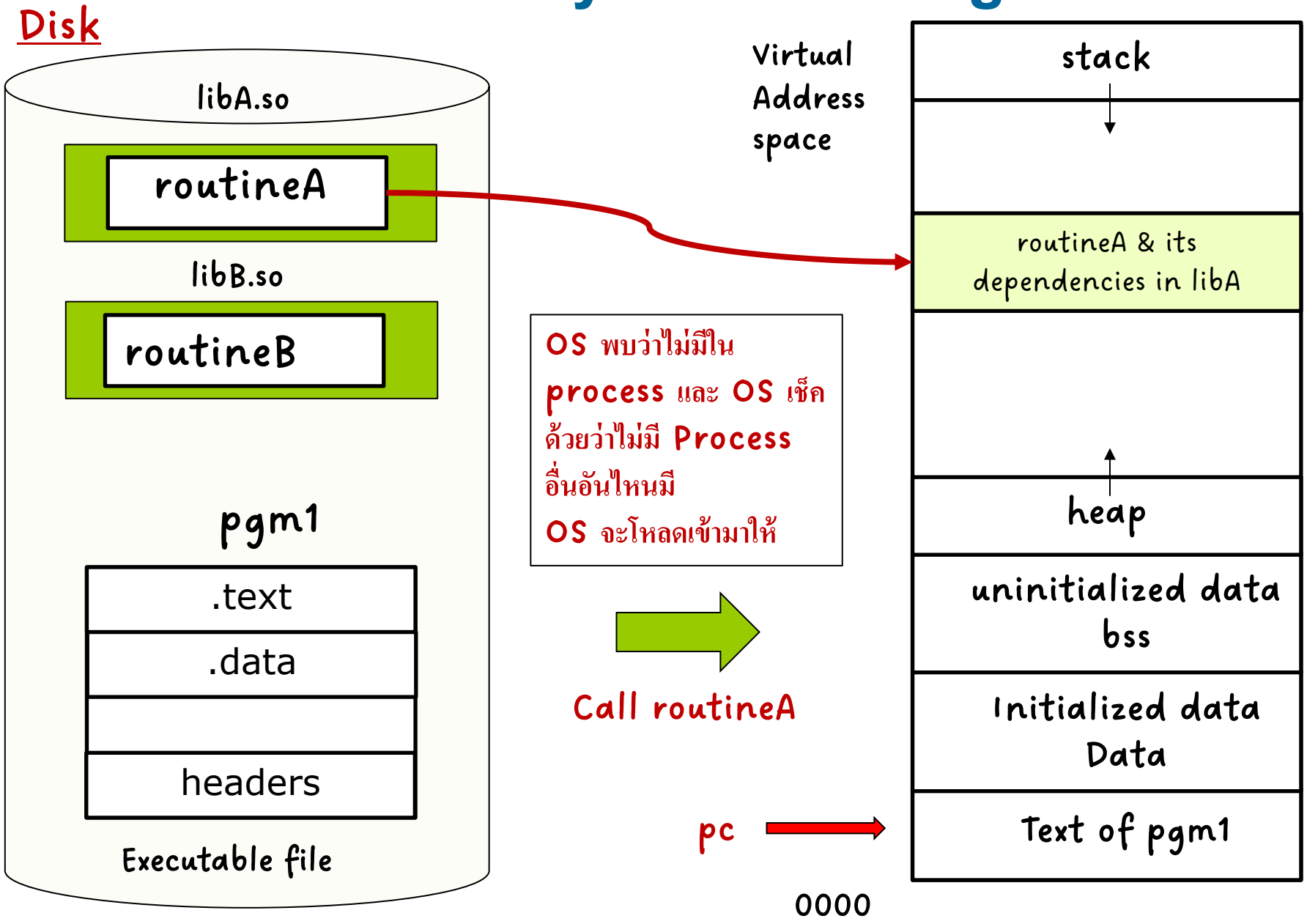
ตัวอย่าง static linking



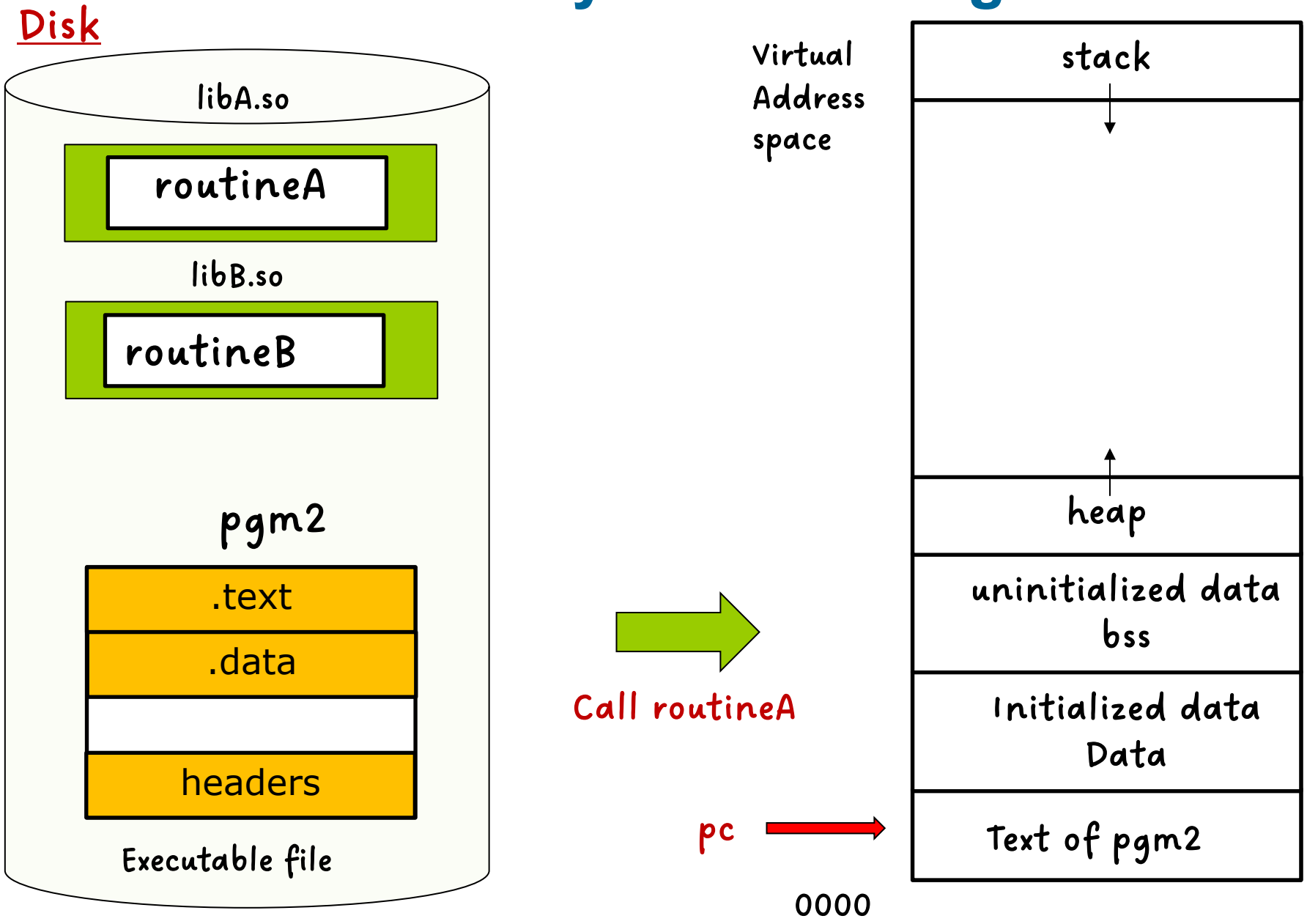
ตัวอย่าง dynamic linking



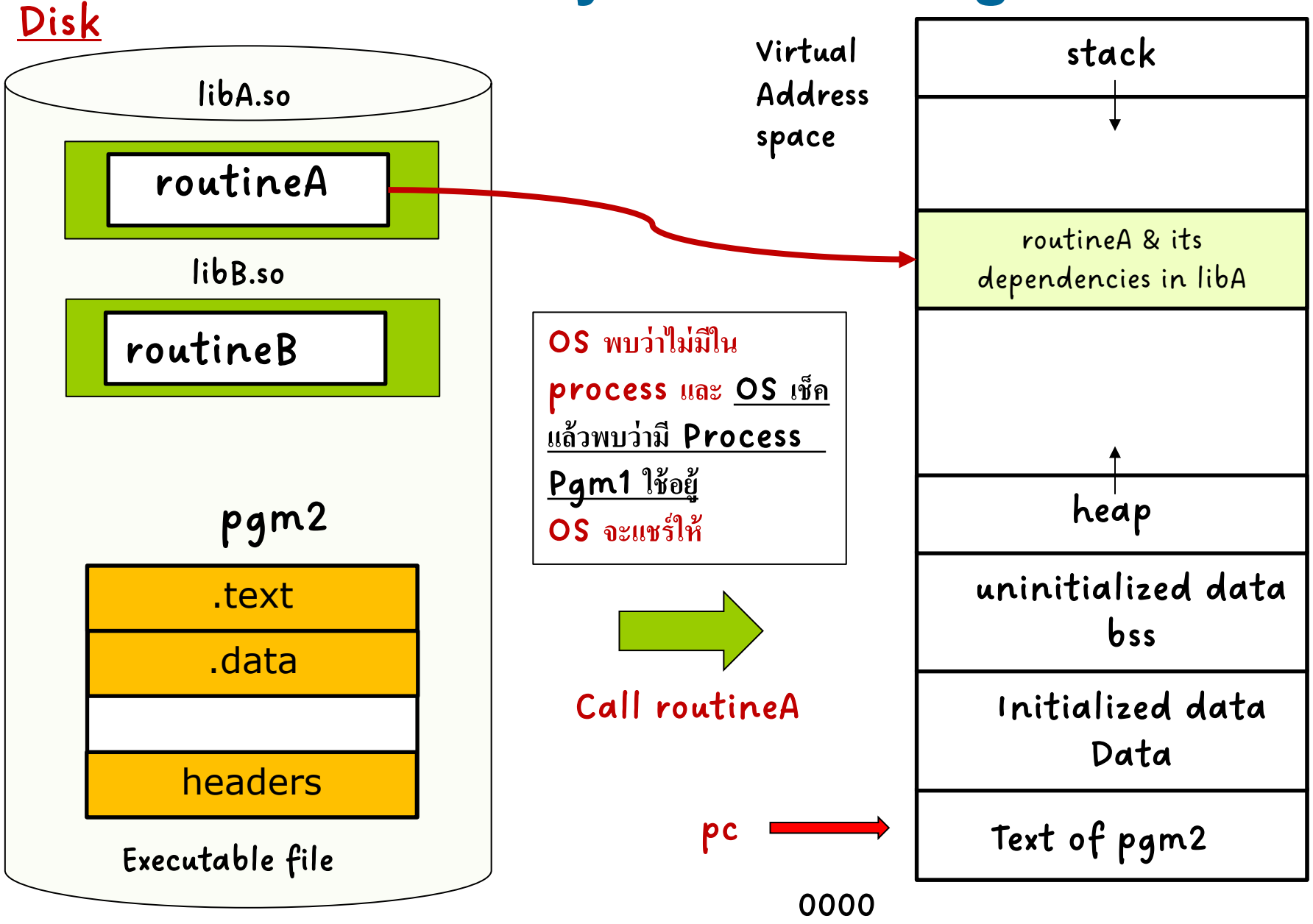
ตัวอย่าง dynamic linking



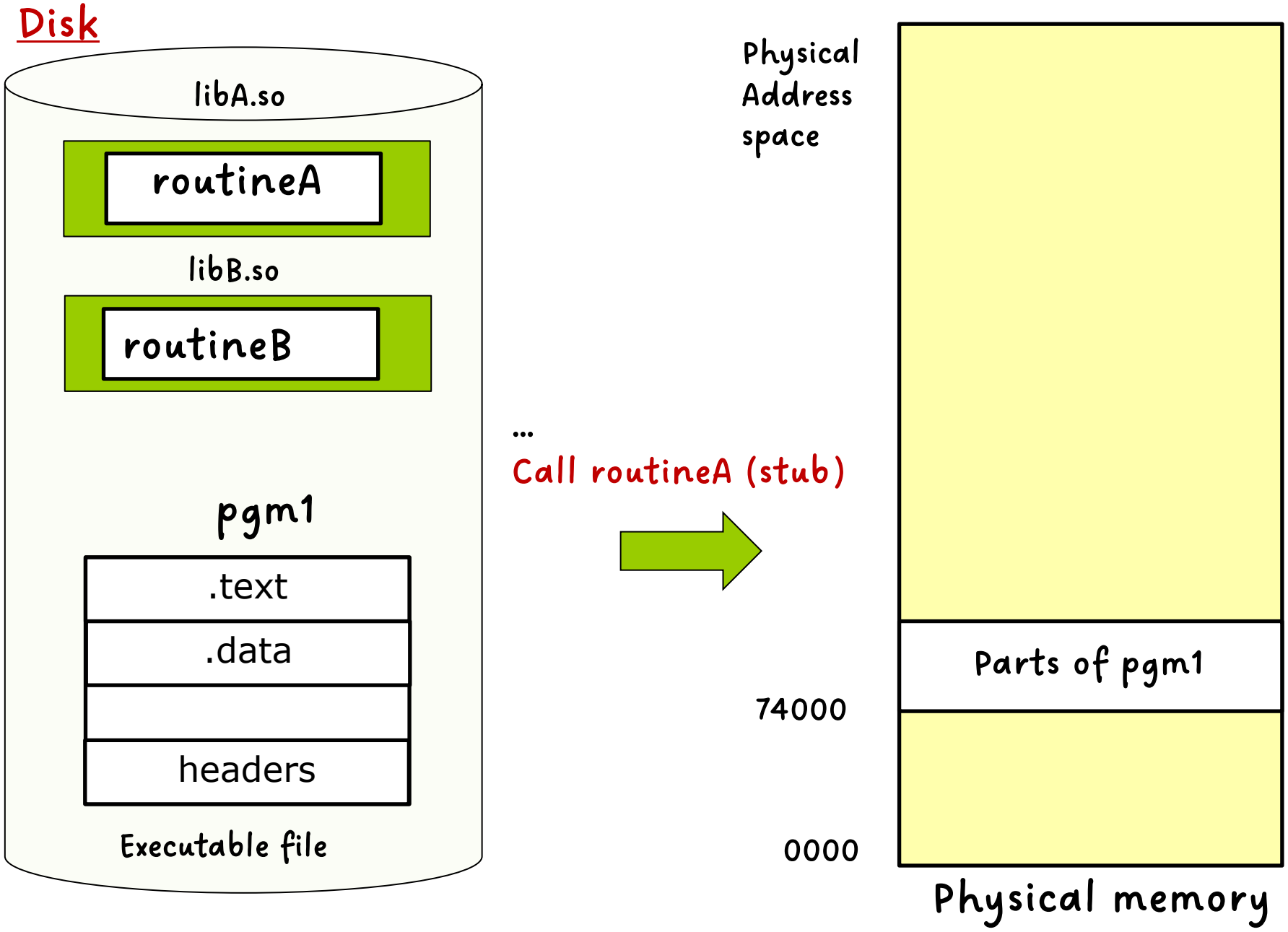
ตัวอย่าง dynamic linking



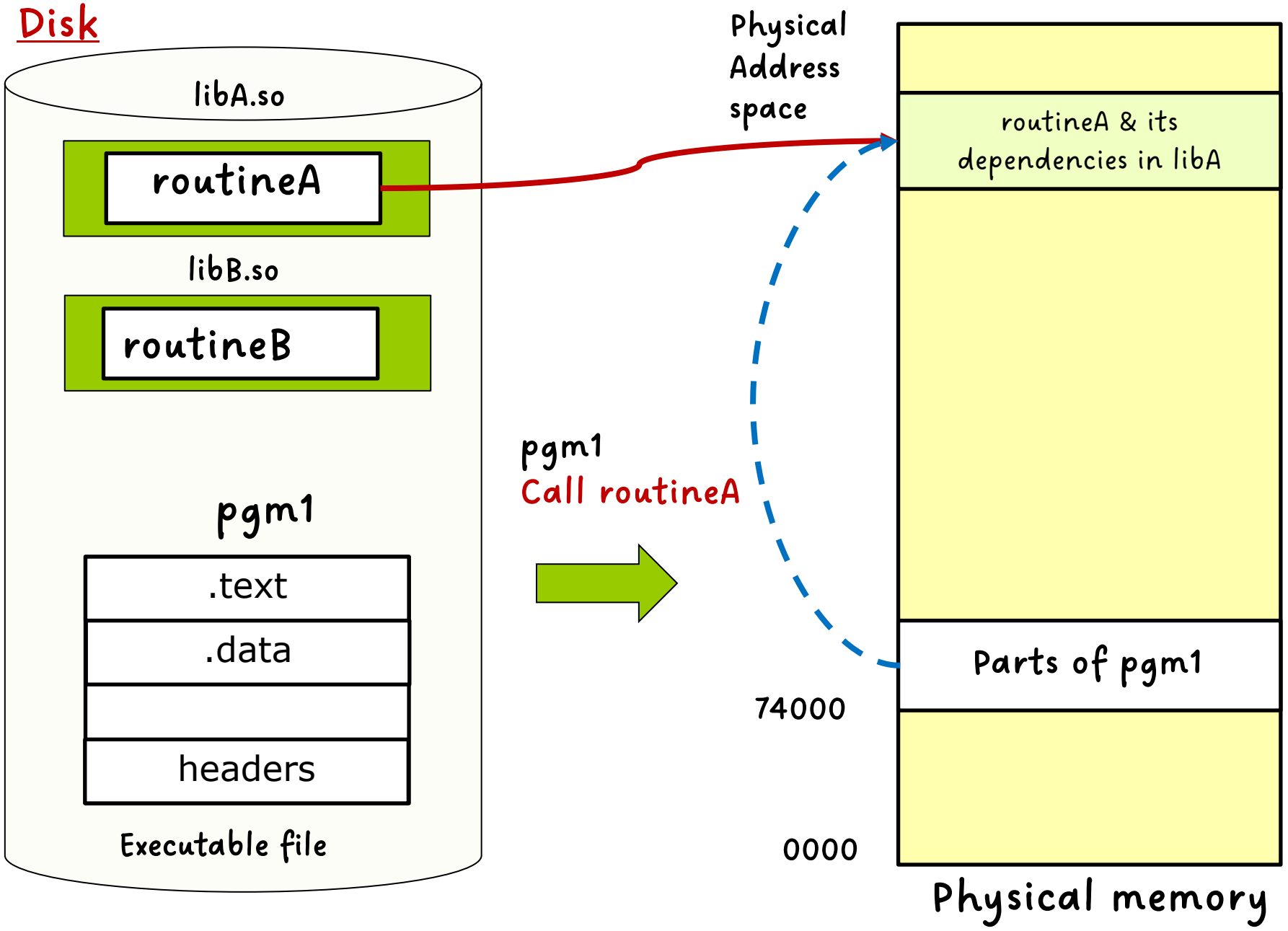
ตัวอย่าง dynamic linking



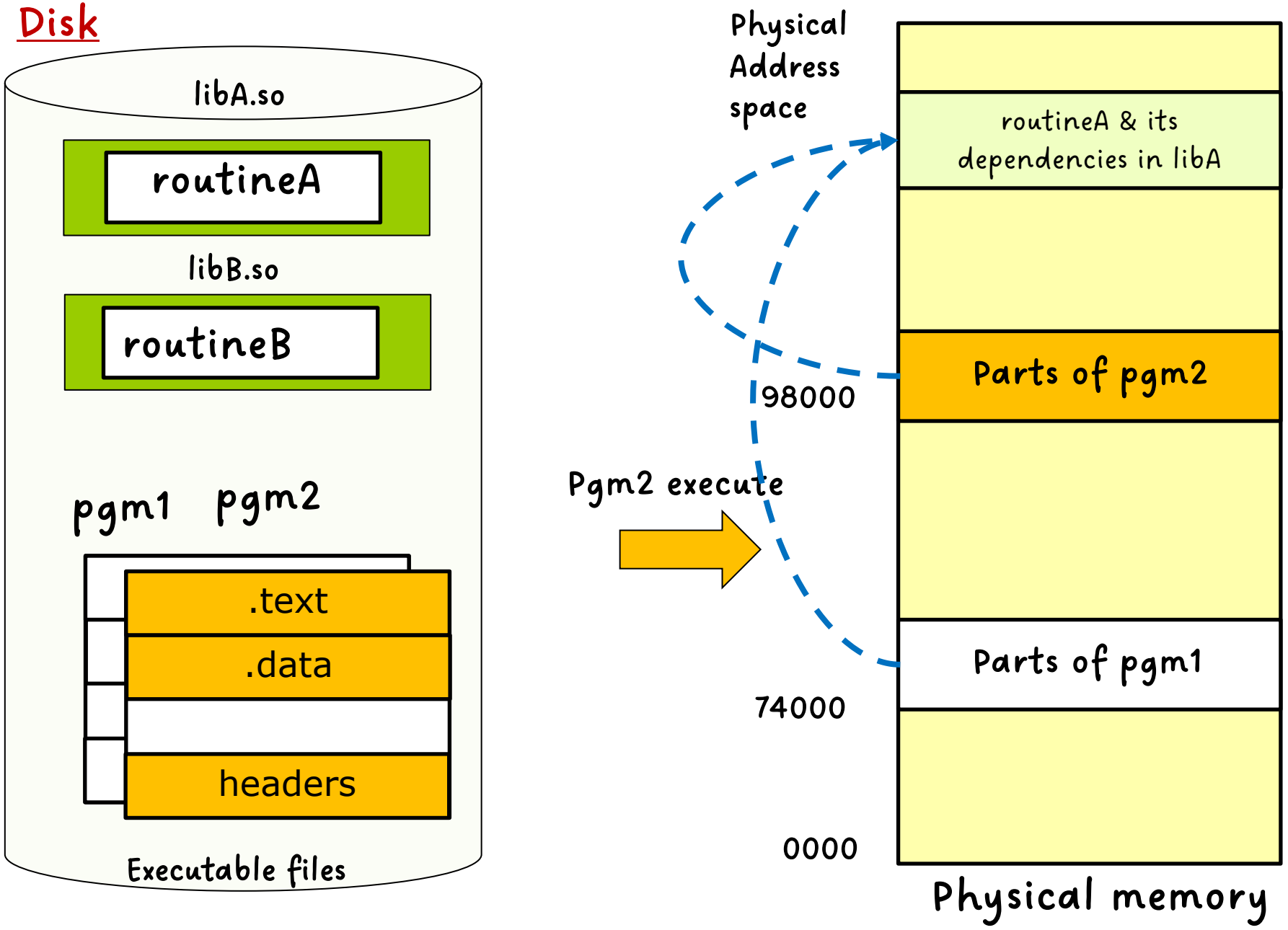
ตัวอย่าง dynamic linking



ตัวอย่าง dynamic linking

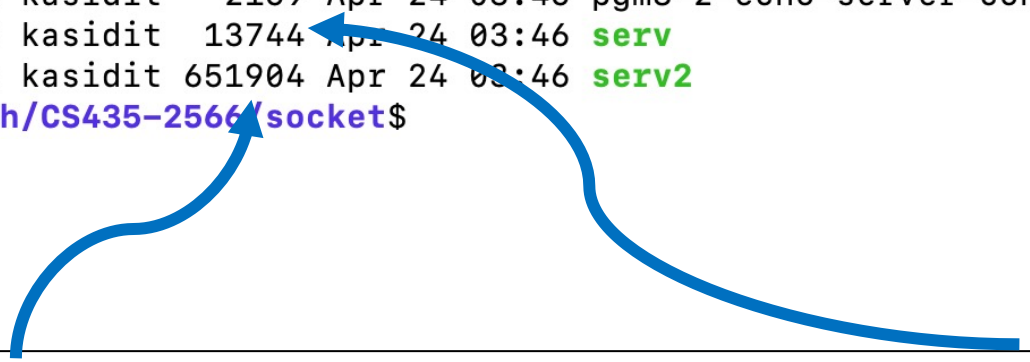


ตัวอย่าง dynamic linking



ตัวอย่าง static vs dynamic linking

```
kasidit@vm:~/Research/CS435-2566/socket$ gcc -o serv pgm2-echo-server.c
kasidit@vm:~/Research/CS435-2566/socket$
kasidit@vm:~/Research/CS435-2566/socket$
kasidit@vm:~/Research/CS435-2566/socket$ gcc -o serv2 -static  pgm2-echo-server.c
kasidit@vm:~/Research/CS435-2566/socket$
kasidit@vm:~/Research/CS435-2566/socket$
kasidit@vm:~/Research/CS435-2566/socket$ ls -l
total 672
-rw-rw-r-- 1 kasidit kasidit   2208 Apr 24 03:46 pgm10-2-echo-server-select.c
-rw-rw-r-- 1 kasidit kasidit   1279 Apr 24 03:46 pgm1-echo-client.c
-rw-rw-r-- 1 kasidit kasidit   1302 Apr 24 03:46 pgm2-echo-server.c
-rw-rw-r-- 1 kasidit kasidit   2169 Apr 24 03:46 pgm8-2-echo-server-concur-thread.c
-rwxrwxr-x 1 kasidit kasidit  13744 Apr 24 03:46 serv
-rwxrwxr-x 1 kasidit kasidit 651904 Apr 24 03:46 serv2
kasidit@vm:~/Research/CS435-2566/socket$
```



ขนาดของ statically link executable file จะใหญ่หว่าของ dynamically linked file

ตัวอย่าง static vs dynamic linking

```
kasidit@vm:~/Research/CS435-2566/socket$ size serv
   text    data     bss     dec     hex filename
   3297     776      32    4105    1009 serv
kasidit@vm:~/Research/CS435-2566/socket$ size serv2
   text    data     bss     dec     hex filename
 515573   22520   22272  560365   88ced serv2
kasidit@vm:~/Research/CS435-2566/socket$ ldd serv
        linux-vdso.so.1 (0x0000ffff9945c000)
        libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000ffff99250000)
        /lib/ld-linux-aarch64.so.1 (0x0000ffff99423000)
kasidit@vm:~/Research/CS435-2566/socket$ ldd serv2
        not a dynamic executable
```

1. ขนาดของ statically link executable file จะใหญ่กว่าของ dynamically linked file
2. คำสั่ง ldd แสดงรายการ dynamically linked libraries
   → ชื่อ .so ไฟล์และ version ของ library
3. ldd จะไม่มีอะไรแสดงสำหรับ statically link executable

ตัวอย่าง static vs dynamic linking

1. ใช้ /proc ดูข้อมูลของ dynamically linked process

```
kasidit@vm:~/Research/CS435-2566/socket$ ./serv &
[1] 17586
kasidit@vm:~/Research/CS435-2566/socket$
kasidit@vm:~/Research/CS435-2566/socket$ cat /proc/17586/maps
aaaae38c0000-aaaae38c1000 r-xp 00000000 fd:00 705419                     /home/kasidit/Research/CS435-2566/socket/serv
aaaae38d1000-aaaae38d2000 r--p 00001000 fd:00 705419                     /home/kasidit/Research/CS435-2566/socket/serv
aaaae38d2000-aaaae38d3000 rw-p 00002000 fd:00 705419                     /home/kasidit/Research/CS435-2566/socket/serv
ffff898c0000-ffff89a48000 r-xp 00000000 fd:00 681451                     /usr/lib/aarch64-linux-gnu/libc.so.6
ffff89a48000-ffff89a57000 ---p 00188000 fd:00 681451                     /usr/lib/aarch64-linux-gnu/libc.so.6
ffff89a57000-ffff89a5b000 r--p 00187000 fd:00 681451                     /usr/lib/aarch64-linux-gnu/libc.so.6
ffff89a5b000-ffff89a5d000 rw-p 0018b000 fd:00 681451                     /usr/lib/aarch64-linux-gnu/libc.so.6
ffff89a5d000-ffff89a69000 rw-p 00000000 00:00 0
ffff89a70000-ffff89a9b000 r-xp 00000000 fd:00 672568                     /usr/lib/aarch64-linux-gnu/ld-linux-aarch64.so.1
ffff89aa5000-ffff89aa7000 rw-p 00000000 00:00 0
ffff89aa7000-ffff89aa9000 r--p 00000000 00:00 0                          [vvar]
ffff89aa9000-ffff89aaa000 r-xp 00000000 00:00 0                          [vdso]
ffff89aaa000-ffff89aac000 r--p 0002a000 fd:00 672568                     /usr/lib/aarch64-linux-gnu/ld-linux-aarch64.so.1
ffff89aac000-ffff89aae000 rw-p 0002c000 fd:00 672568                     /usr/lib/aarch64-linux-gnu/ld-linux-aarch64.so.1
ffffe4e34000-ffffe4e55000 rw-p 00000000 00:00 0                          [stack]
kasidit@vm:~/Research/CS435-2566/socket$ kill -9 17586
```

2. ใช้ /proc ดูข้อมูลของ statically linked process

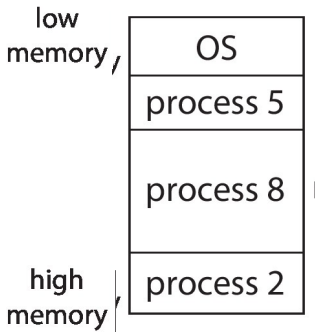
```
kasidit@vm:~/Research/CS435-2566/socket$ ./serv2 &
[2] 17592
[1]   Killed                  ./serv
kasidit@vm:~/Research/CS435-2566/socket$
kasidit@vm:~/Research/CS435-2566/socket$ cat /proc/17592/maps
00400000-0047e000 r-xp 00000000 fd:00 705420                             /home/kasidit/Research/CS435-2566/socket/serv2
0048e000-00492000 r--p 0007e000 fd:00 705420                             /home/kasidit/Research/CS435-2566/socket/serv2
00492000-00495000 rw-p 00082000 fd:00 705420                             /home/kasidit/Research/CS435-2566/socket/serv2
00495000-0049a000 rw-p 00000000 00:00 0
020c3000-020e5000 rw-p 00000000 00:00 0                                  [heap]
ffffa6391000-ffffa6393000 r--p 00000000 00:00 0                          [vvar]
ffffa6393000-ffffa6394000 r-xp 00000000 00:00 0                          [vdso]
ffffd7d78000-ffffd7d99000 rw-p 00000000 00:00 0                          [stack]
kasidit@vm:~/Research/CS435-2566/socket$ kill -9 17592
```

การจัดการเนื้อหาโปรแกรมใน Memory

- **Contiguous Allocation**
- **Paging**

Contiguous Allocation

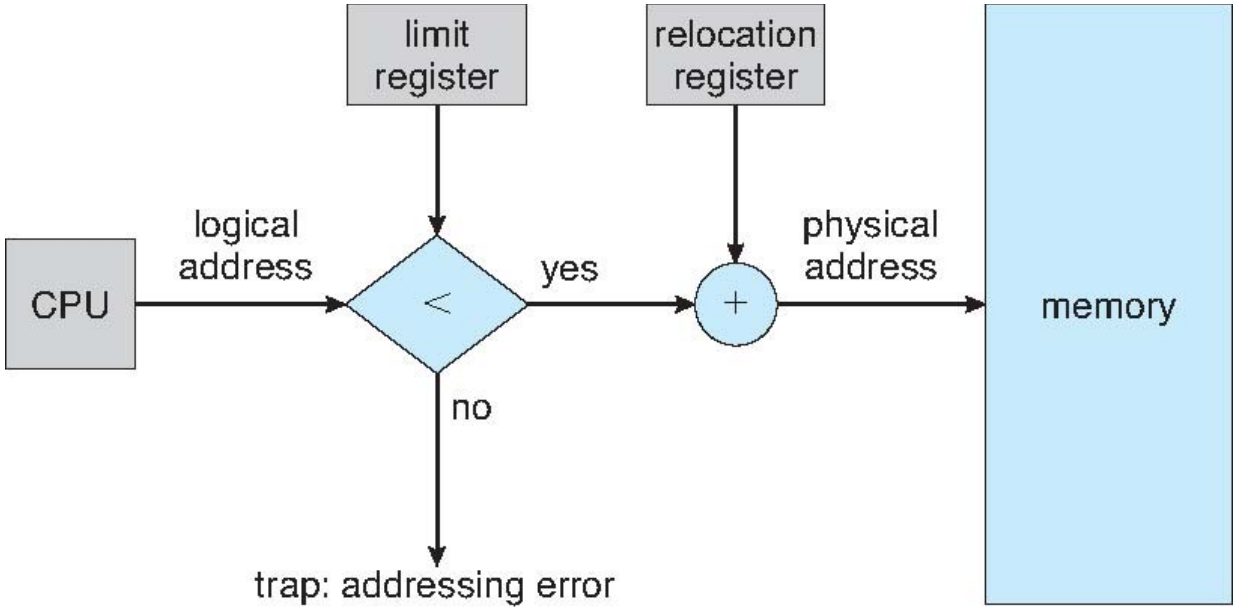
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - **Resident operating system**, usually held in low memory with interrupt vector
    - แตกต่างจากตัวอย่างที่กล่าวถึงก่อนหน้า
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory



Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size
    - เปลี่ยนตำแหน่งของ user processes เพื่อขยายขนาด kernel

Hardware Support for Relocation and Limit Registers



ตย. Relocation = 100040 และ limit = 74600

Logical Address = 100 ผลคือ

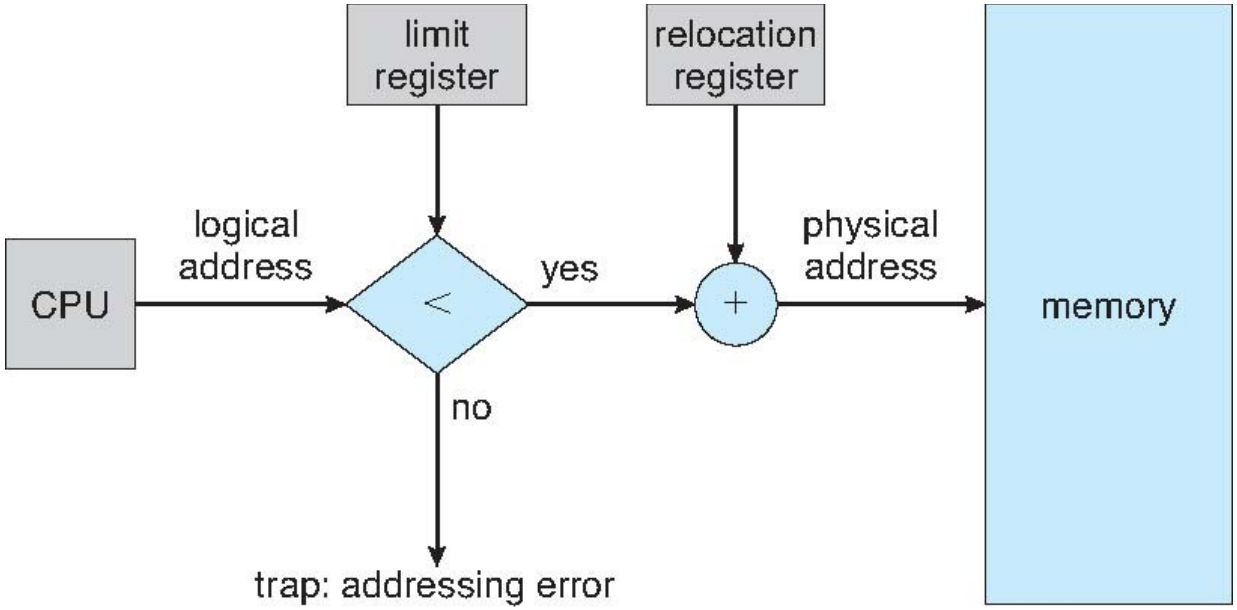
Logical Address = 0 ผลคือ

Logical Address = 100040 ผลคือ

Logical Address = 74599 ผลคือ

Logical Address = 100039 ผลคือ

Hardware Support for Relocation and Limit Registers



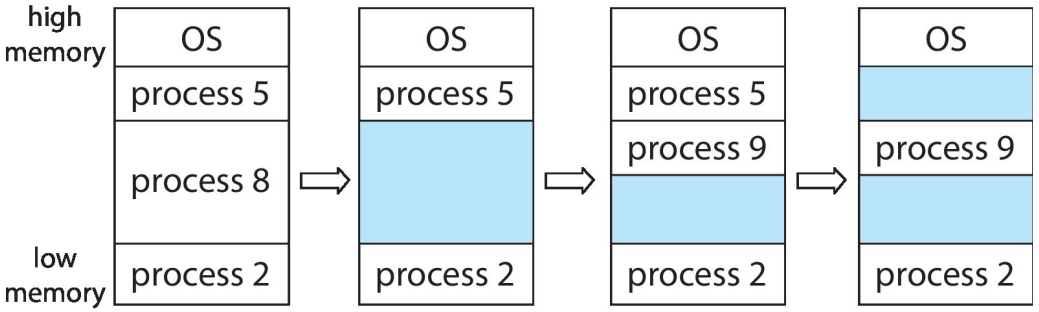
ตย. Relocation = 100040 และ limit = 74600

Logical Address Space ของ Process คือ: __ ถึง _____

Physical Address Space ของ Process คือ: __ ถึง _____

Variable Partition

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)

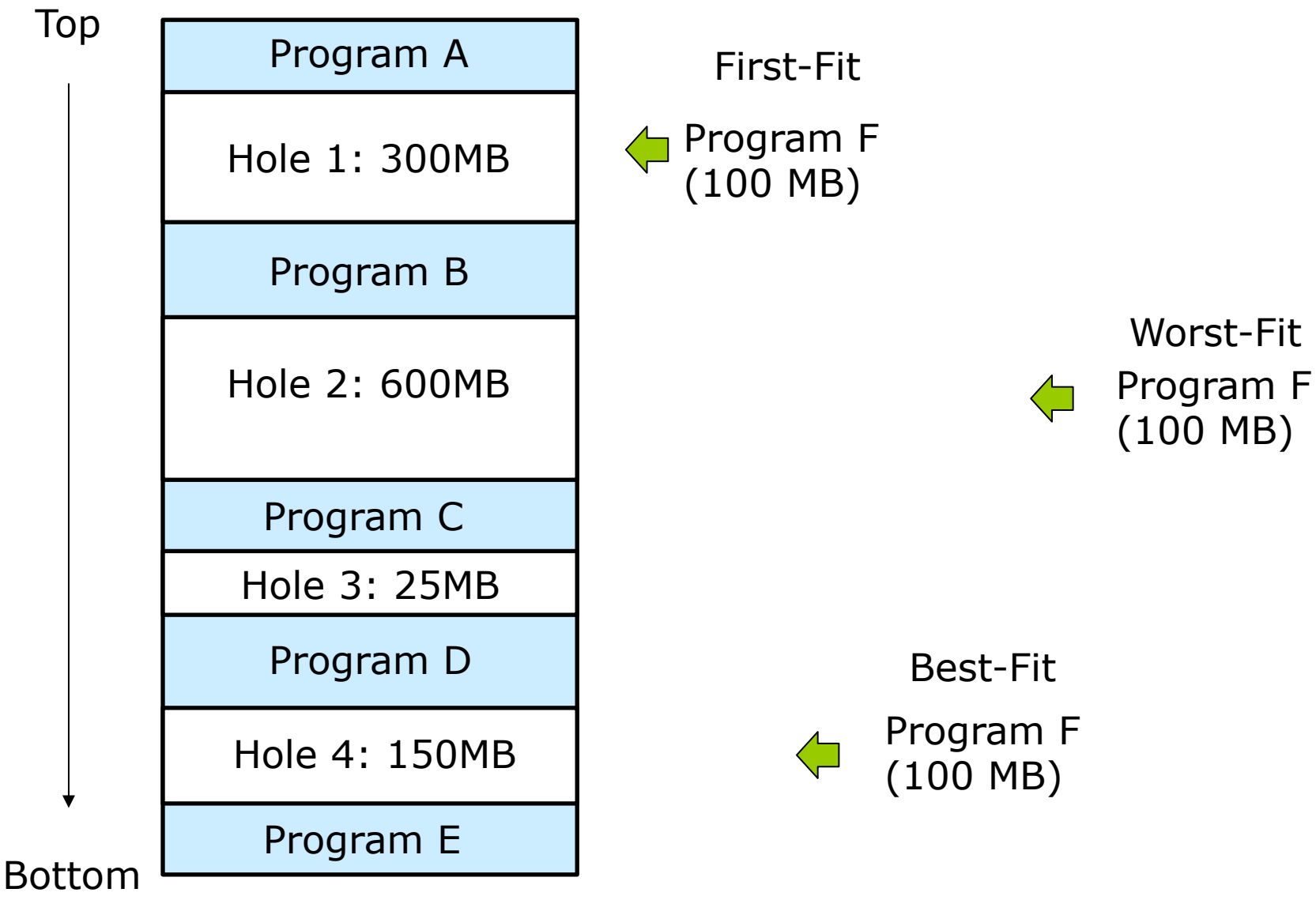


Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

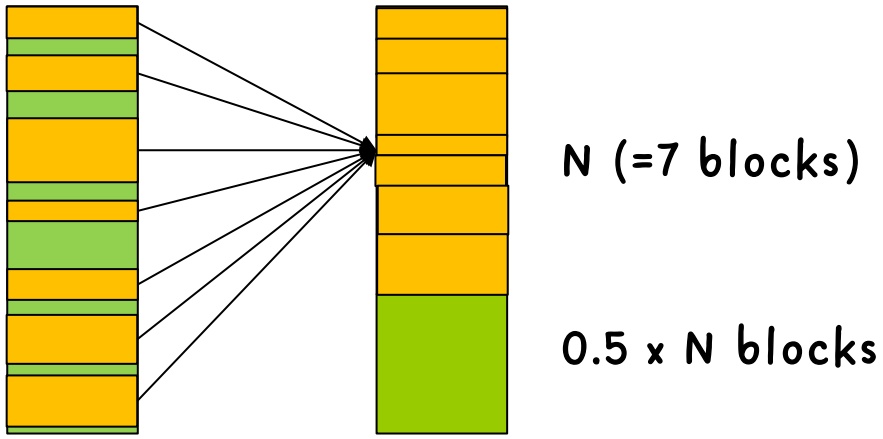
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization


Top
Program A
Hole 1: 300MB
Program B
Hole 2: 600MB
Program C
Hole 3: 25MB
Program D
Hole 4: 150MB
Program E
Bottom
First-Fit
Program F
(100 MB)
Worst-Fit
Program F
(100 MB)
Best-Fit
Program F
(100 MB)

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**
  - ตย. ถ้าใช้หน่วยความจำไป **150 blocks** (สมมุติว่า **1 block = 1 KB**) จะเสียพื้นที่ให้กับ **fragmentation 50 blocks**



Fragmentation (Cont.)

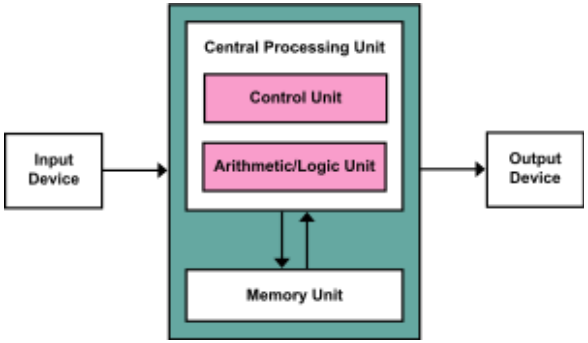
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O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

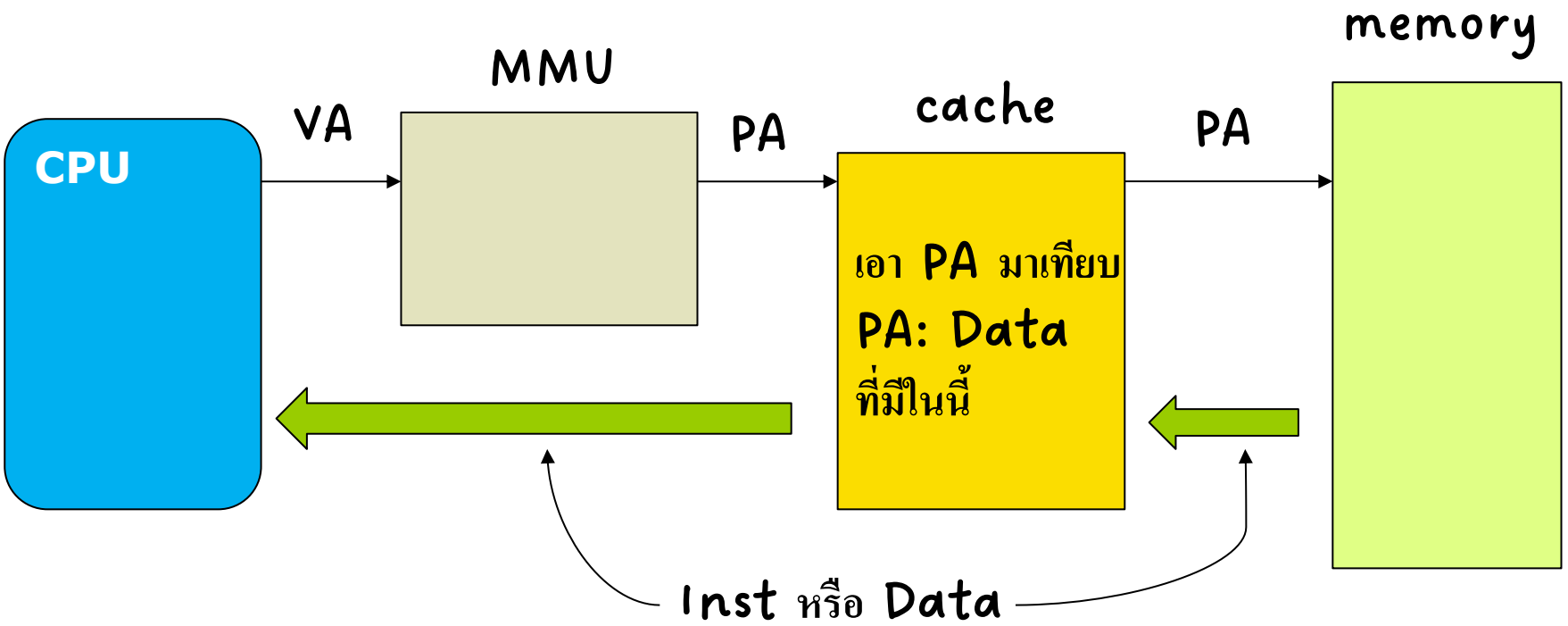
- Now consider that backing store (storage) has same fragmentation problems

ทบทวนการทำงานของ CPU และ Memory

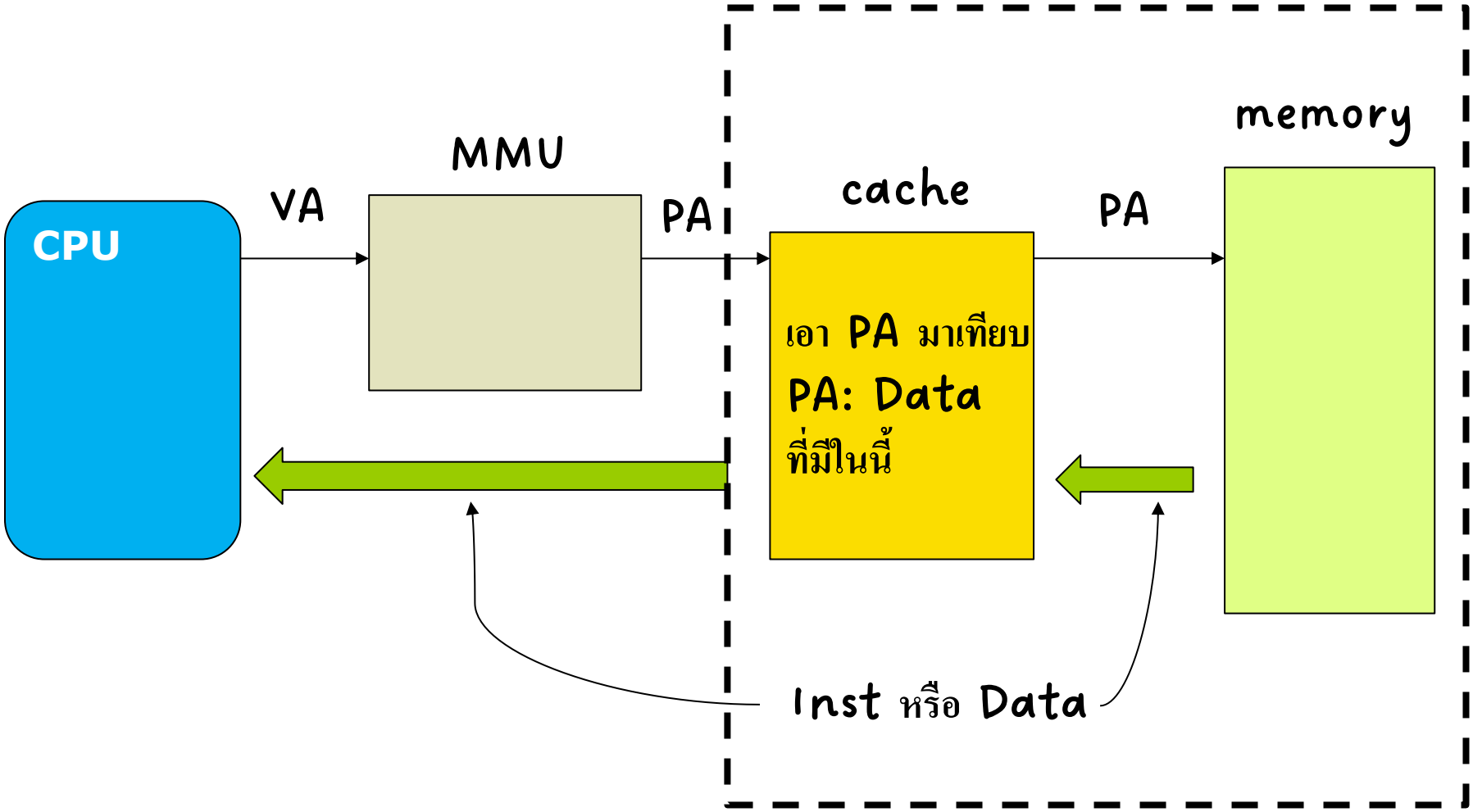
- เริ่มต้นจาก von Neumann Architecture

  ที่กล่าวถึง คอมพิวเตอร์ที่เก็บโปรแกรมใน memory
- CPU ต้องการชุดคำสั่งหรือข้อมูลก็จะส่งคำขอไปให้ Memory
- Memory รับคำขอและส่งข้อมูลกลับ

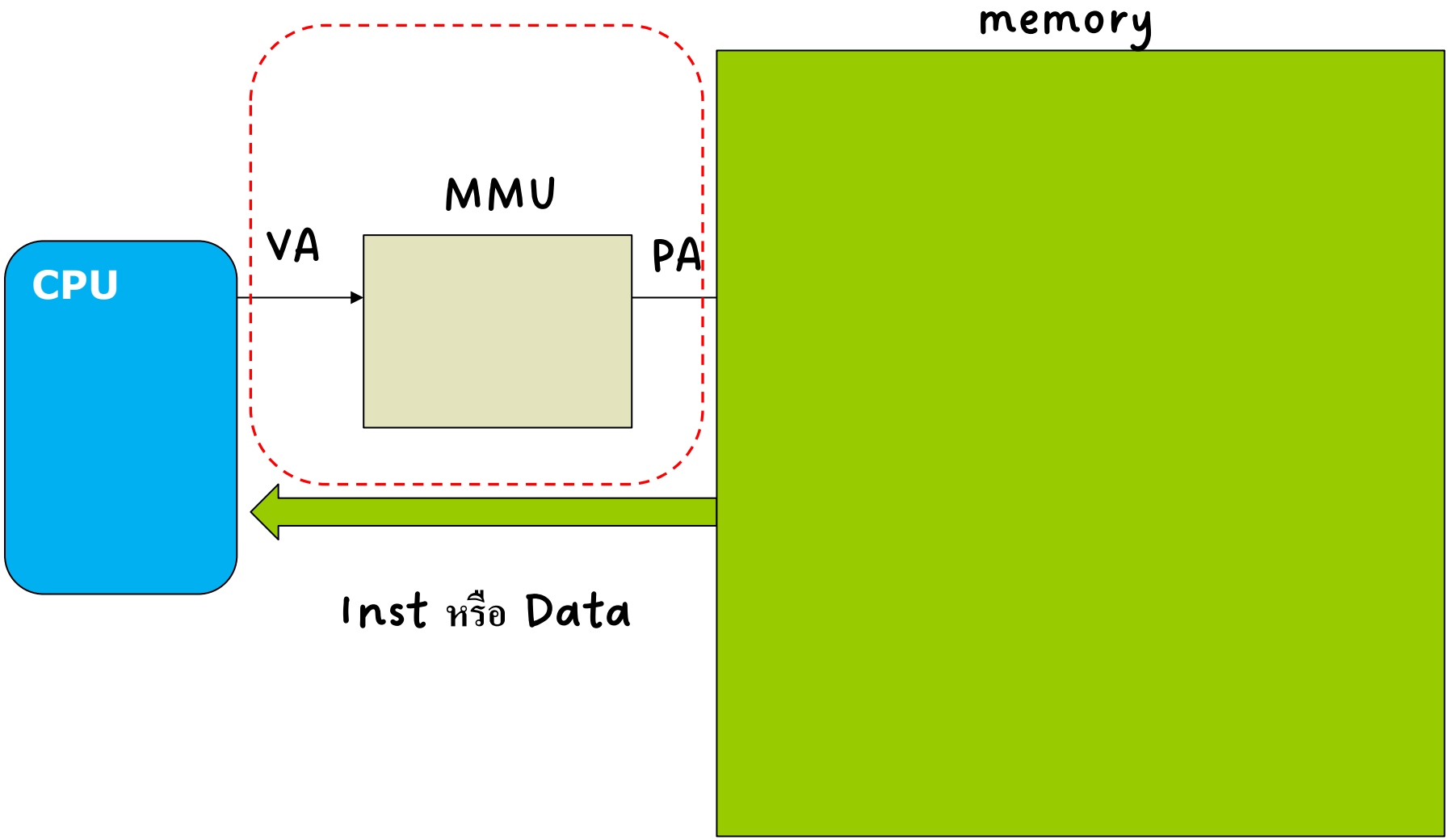




การทำงานของ CPU และ Memory



การทำงานของ CPU และ Memory

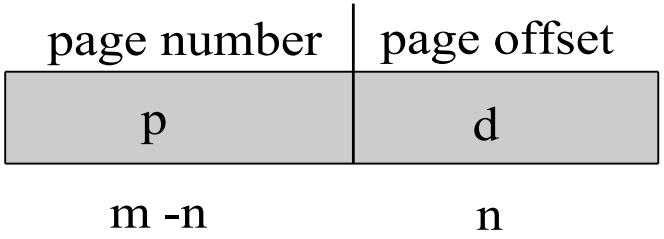


Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Address Translation Scheme

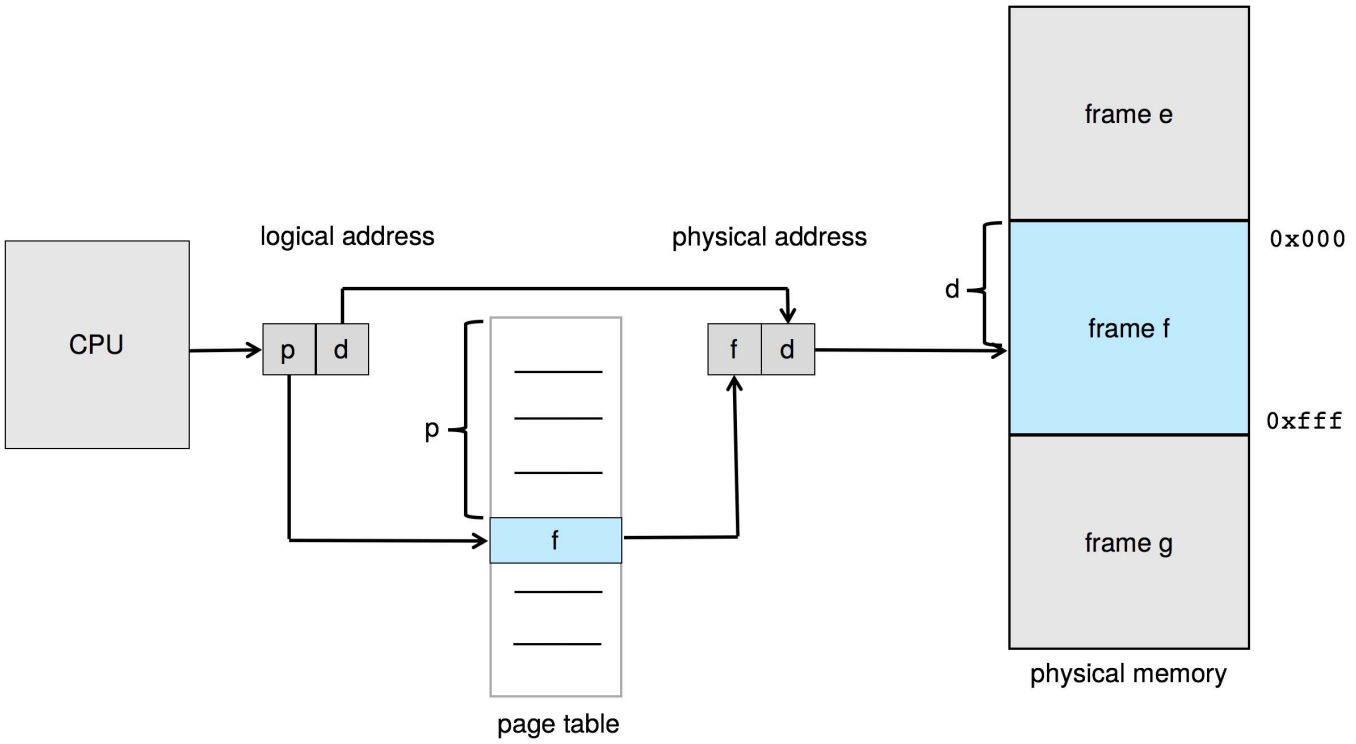
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



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

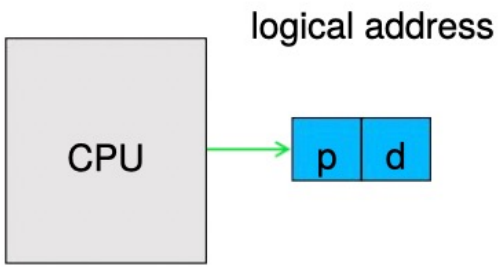
  - Memory จะเป็นแบบ Byte Addressable หมายความว่า Address 1 ค่า เป็น Address สำหรับ 1 byte ในหน่วยความจำ

Paging Hardware



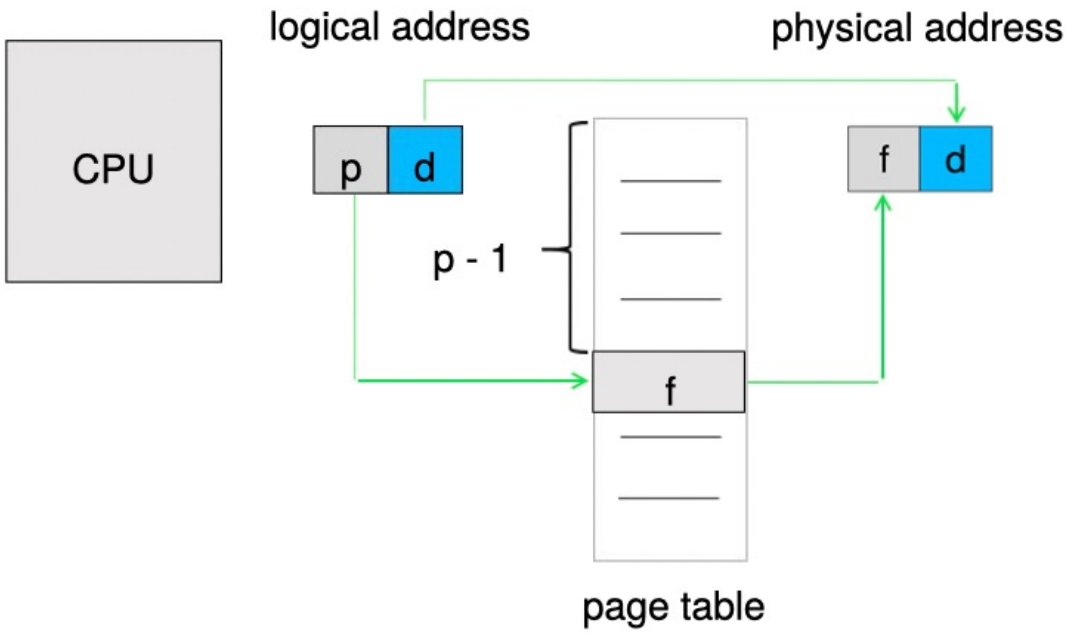
Paging Hardware

1. the logical address generated by the CPU is sent to the MMU where it is divided into a page number (p) and an offset (d)
   – the number of bits in each part depends on the page size



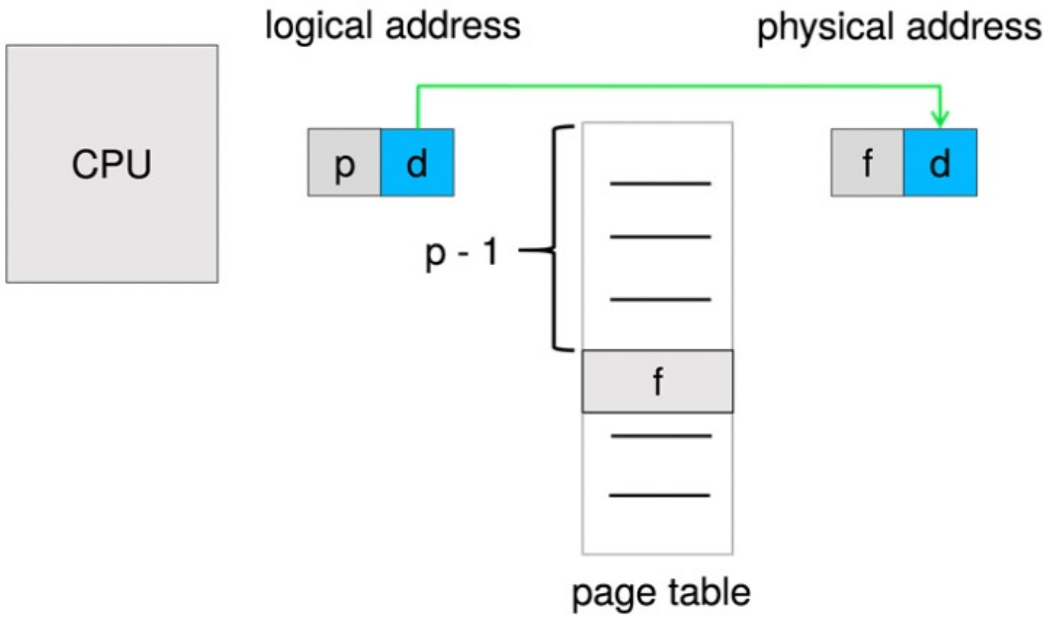
Paging Hardware

2. the page number is used as an index into the page table
   – the entry in the page table at that index is the number of the frame of physical memory containing the page
   – that frame number is the first part of the physical memory address



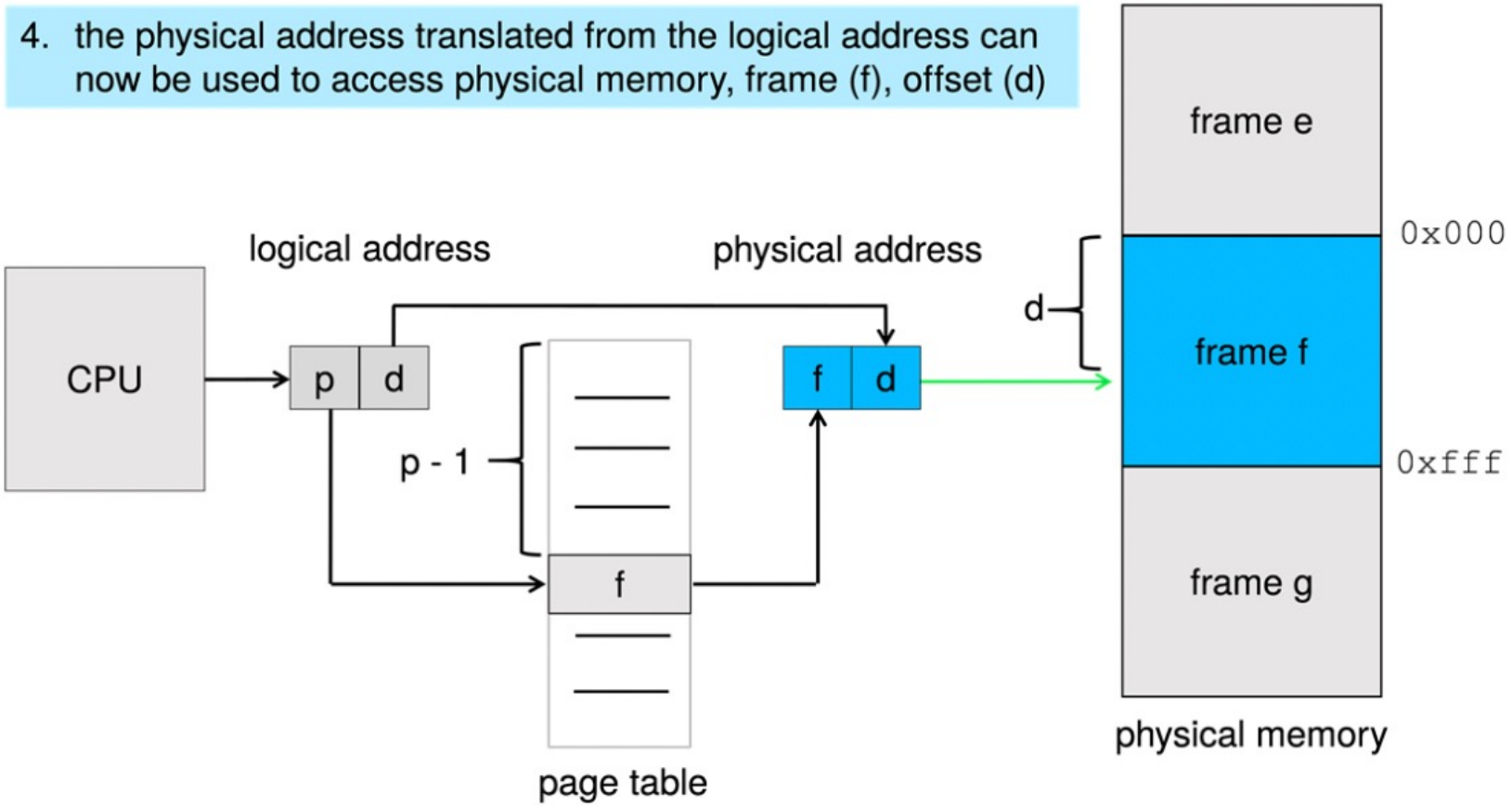
Paging Hardware

3. the offset (d) within the page is the same as the offset within the frame, so it is retained, together with the frame number forming the physical address

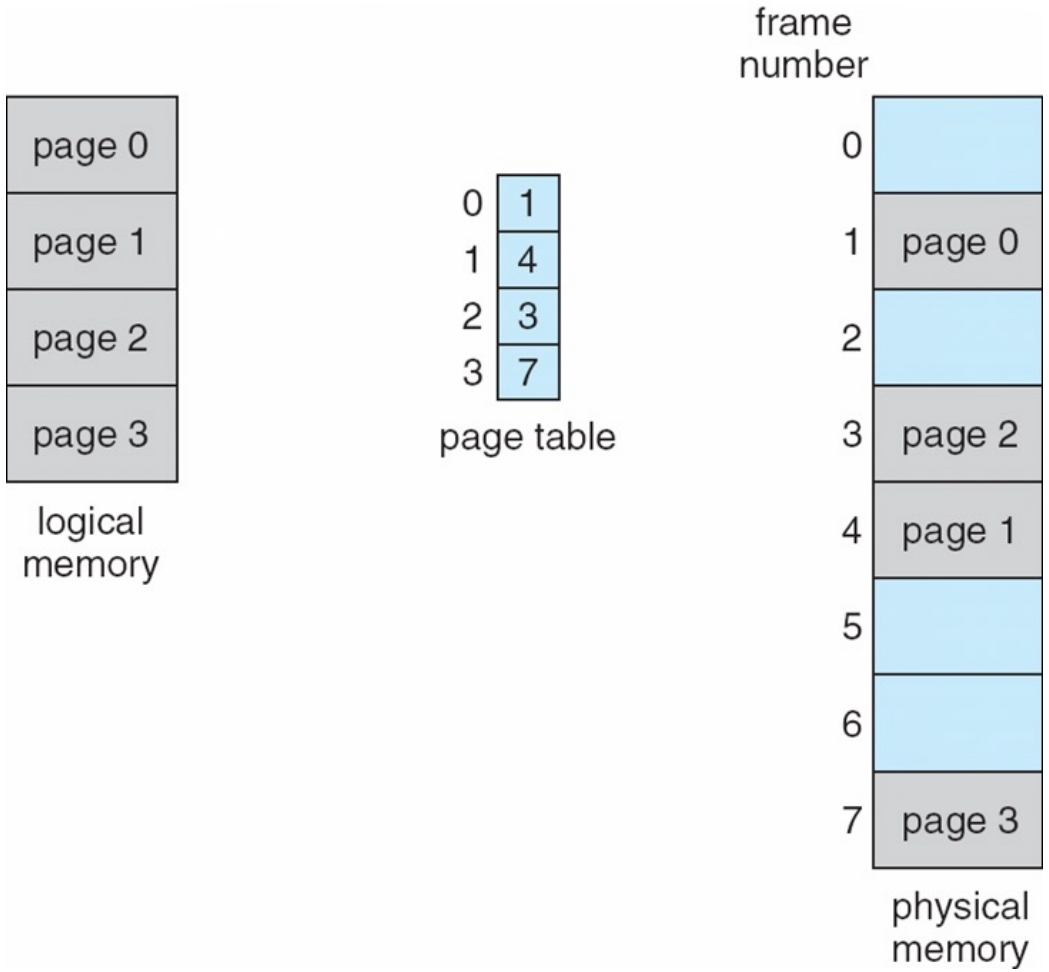


Paging Hardware

4. the physical address translated from the logical address can now be used to access physical memory, frame (f), offset (d)

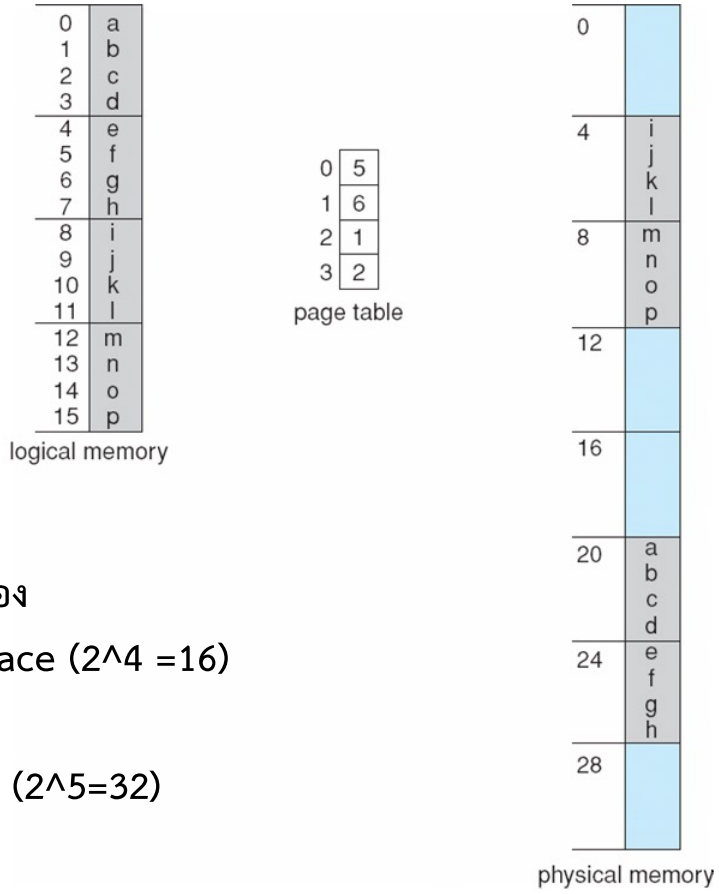


Paging Model of Logical and Physical Memory



Paging Example

- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



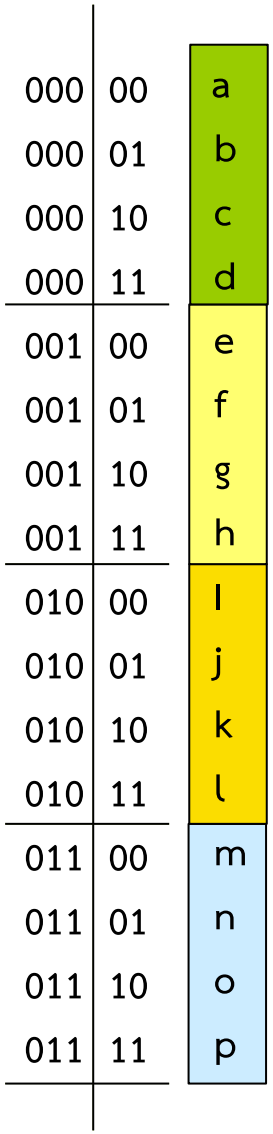
ในตัวอย่างนี้ ขนาดของ

logical address space (2^4 =16)

น้อยกว่าขนาดของ

physical memory (2^5=32)

- Logical address: n = 2 and m = 5. Using a page size of 2^2 = 4 bytes and a physical memory of 32 bytes (8 pages)

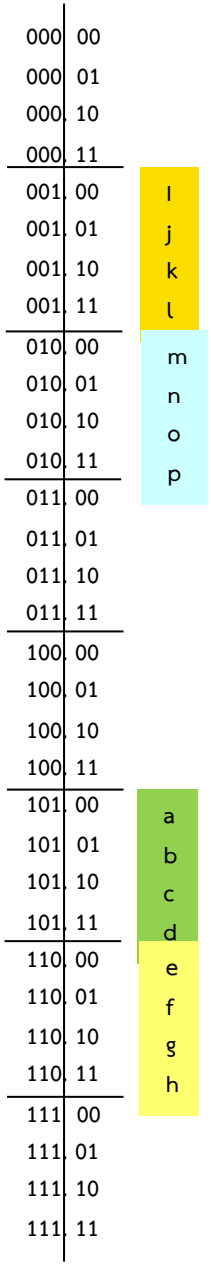


Logical memory

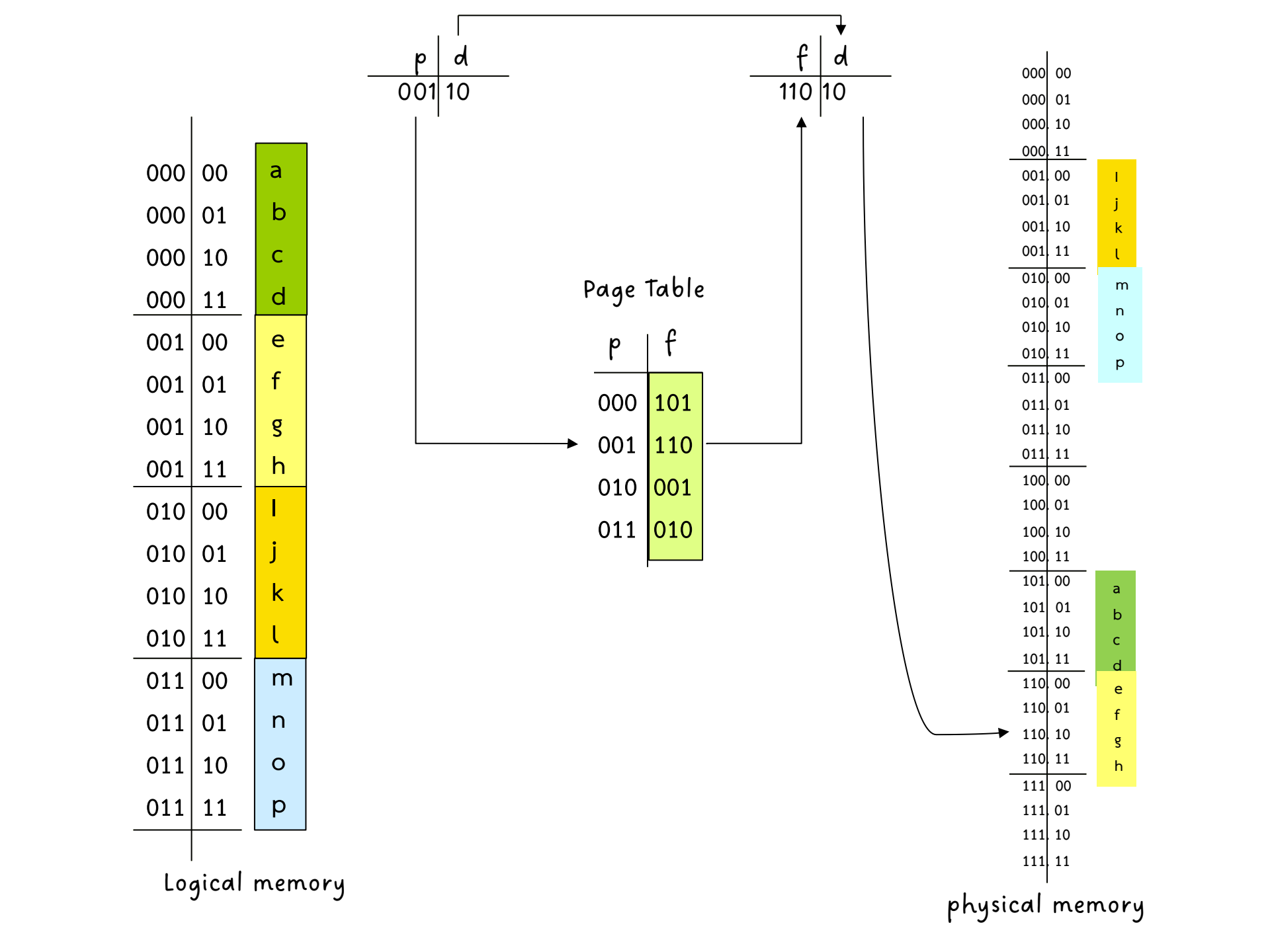
Page Table

p	f
000	101
001	110
010	001
011	010

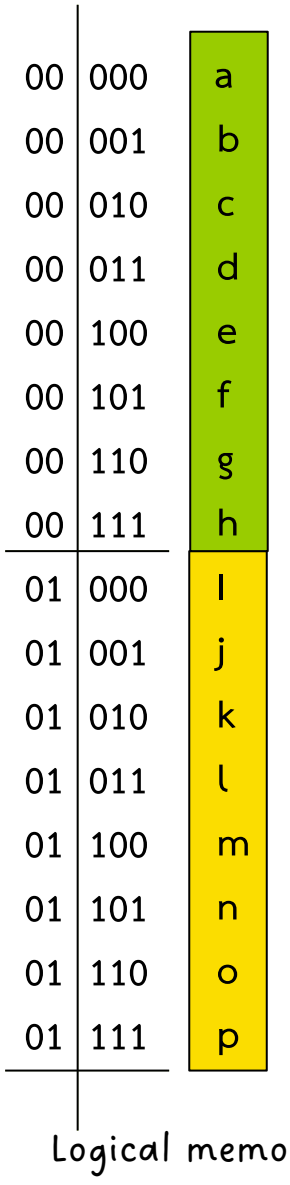
ในตัวอย่างนี้ ขนาดของ logical address space (2^5 =32) เท่ากับขนาดของ physical memory (2^5=32)



physical memory

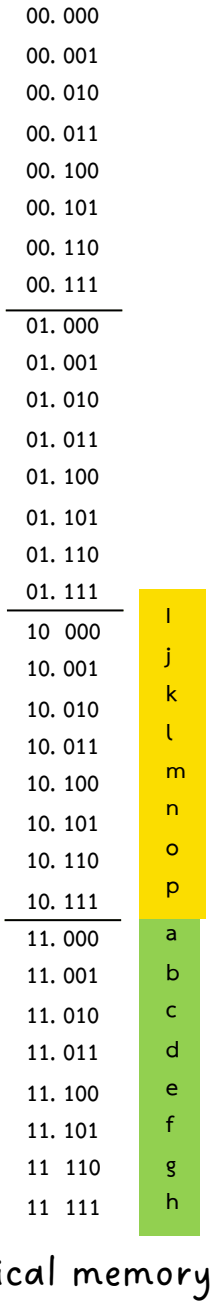

p d
001 10
f d
110 10
Page Table
p f
000 101
001 110
010 001
011 010
000 00 a
000 01 b
000 10 c
000 11 d
001 00 e
001 01 f
001 10 g
001 11 h
010 00 I
010 01 j
010 10 k
010 11 l
011 00 m
011 01 n
011 10 o
011 11 p
Logical memory
000 00
000 01
000 10
000 11
001 00 I
001 01 j
001 10 k
001 11 l
010 00 m
010 01 n
010 10 o
010 11 p
011 00
011 01
011 10
011 11
100 00
100 01
100 10
100 11
101 00 a
101 01 b
101 10 c
101 11 d
110 00 e
110 01 f
110 10 g
110 11 h
111 00
111 01
111 10
111 11
physical memory

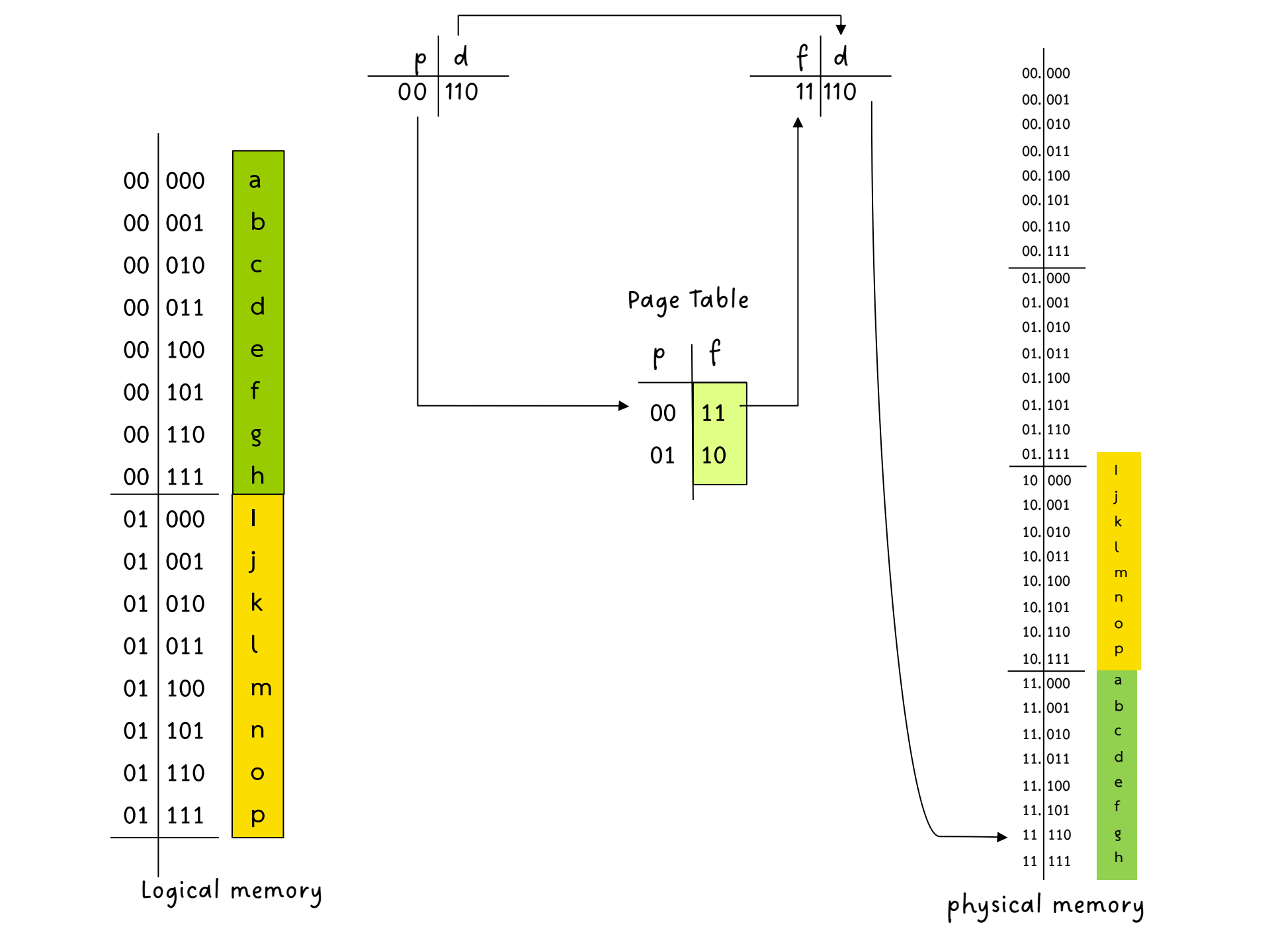
- Logical address: n = 3 and m = 5. Using a page size of 2^3 =8 bytes and a physical memory of 32 bytes (4 pages)



Page Table

p	f
00	11
01	10

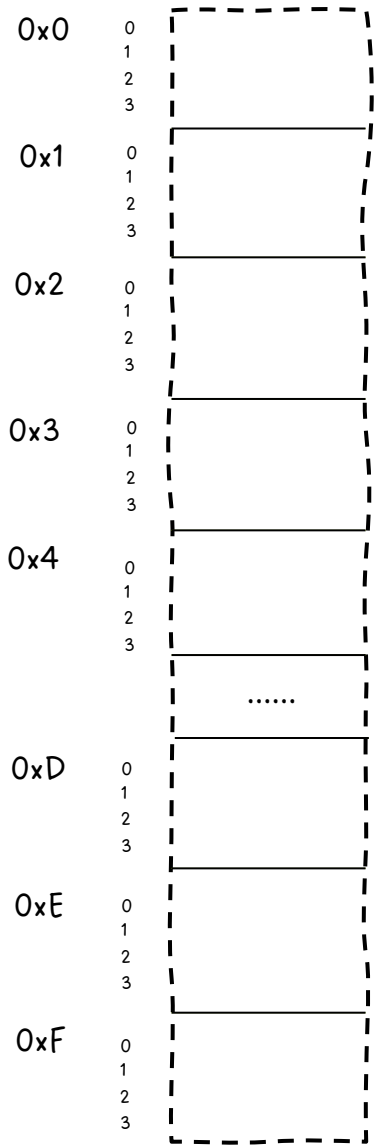



p d
00 110
f d
11 110
Page Table
p f
00 11
01 10
00 000 a
00 001 b
00 010 c
00 011 d
00 100 e
00 101 f
00 110 g
00 111 h
01 000 l
01 001 j
01 010 k
01 011 l
01 100 m
01 101 n
01 110 o
01 111 p
Logical memory
00. 000
00. 001
00. 010
00. 011
00. 100
00. 101
00. 110
00. 111
01. 000
01. 001
01. 010
01. 011
01. 100
01. 101
01. 110
01. 111
10 000
10. 001
10. 010
10. 011
10. 100
10. 101
10. 110
10. 111
11. 000
11. 001
11. 010
11. 011
11. 100
11. 101
11 110
11 111
l j k l m n o p
a b c d e f g h
physical memory

Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)

Page id (p) : offset (d)

4 bits : 2 bits

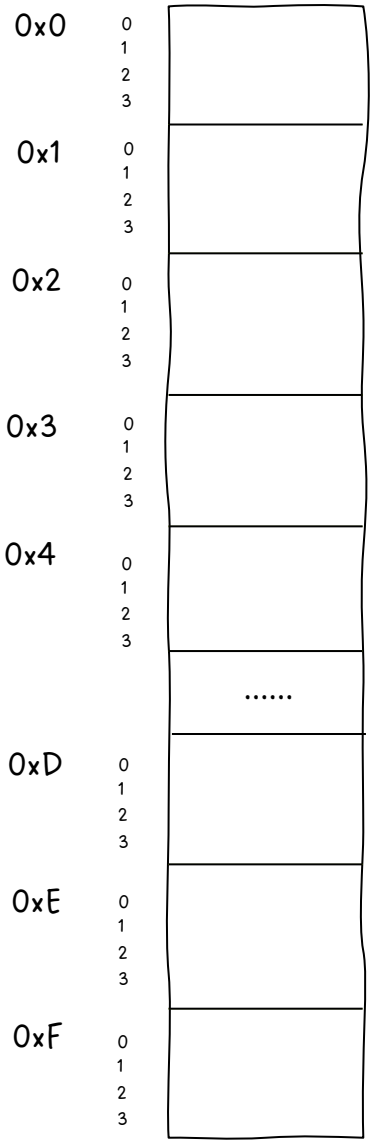


frome id	Page table contents
0x0	0x11
0x1	0x3C
0x2	0x06
0x3	----
0x4	----
0x5	0x10
0x6	----
0x7	0x2E
0x8	0x0F
0x9	----
0xA	0x1B
0xB	0x34
0xC	----
0xD	0x2A
0xE	0x14
0xF	----

- โจทย์ ให้ ตัวอย่างเนื้อหาใน Physical Mem แล้วให้ นศ สร้าง page table

Frame id (f) : offset (d)

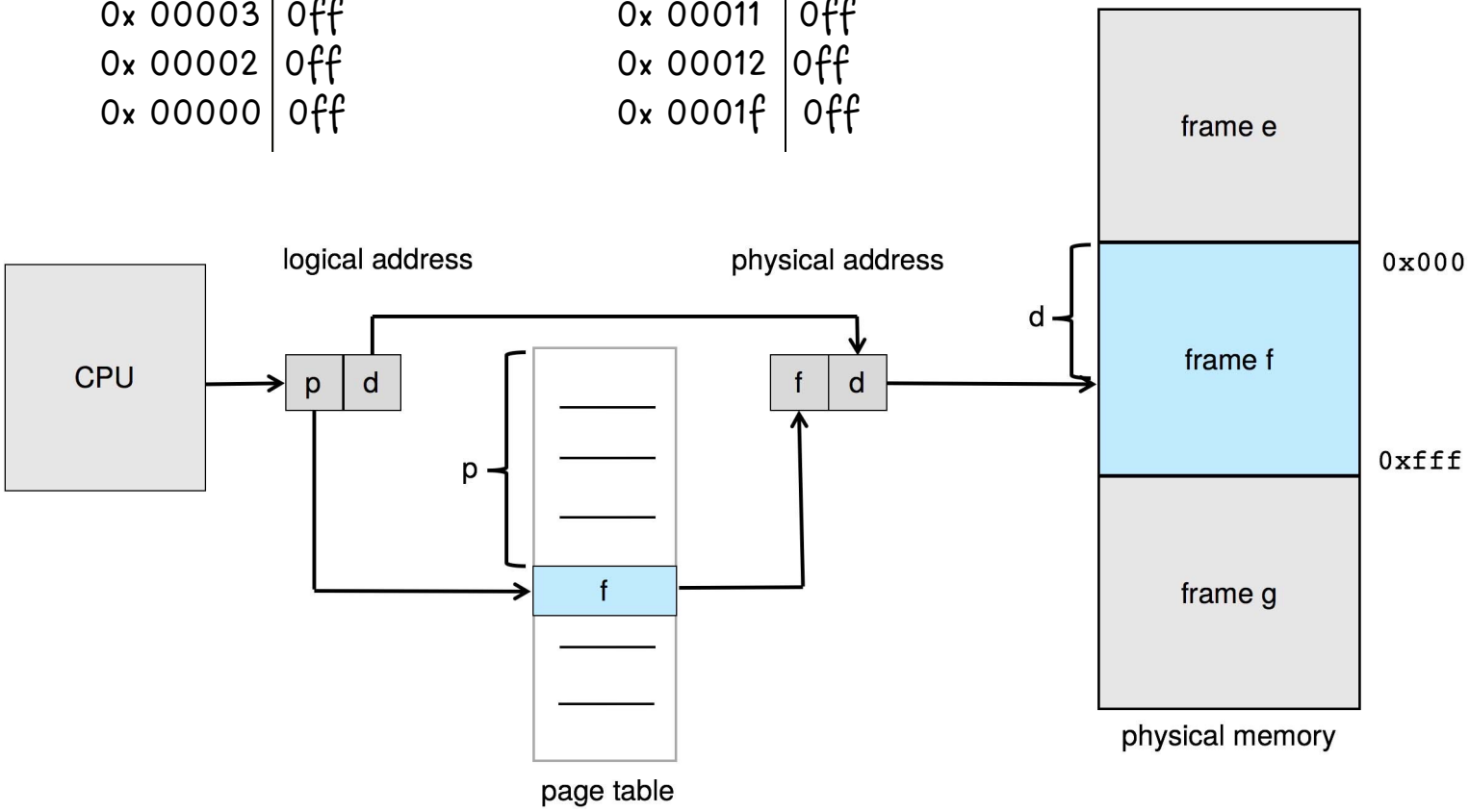
4 bits : 2 bits



- Logical address: n = 12 and m = 32. Using a page size of 2^12 = 4KB bytes and a physical memory of 2^32 = 4GB (4*1024^3/4*1024^1 = 1024^2 = 1048576 pages)

p	d
0x 00001	0ff
0x 00003	0ff
0x 00002	0ff
0x 00000	0ff

f	d
0x 00010	0ff
0x 00011	0ff
0x 00012	0ff
0x 0001f	0ff

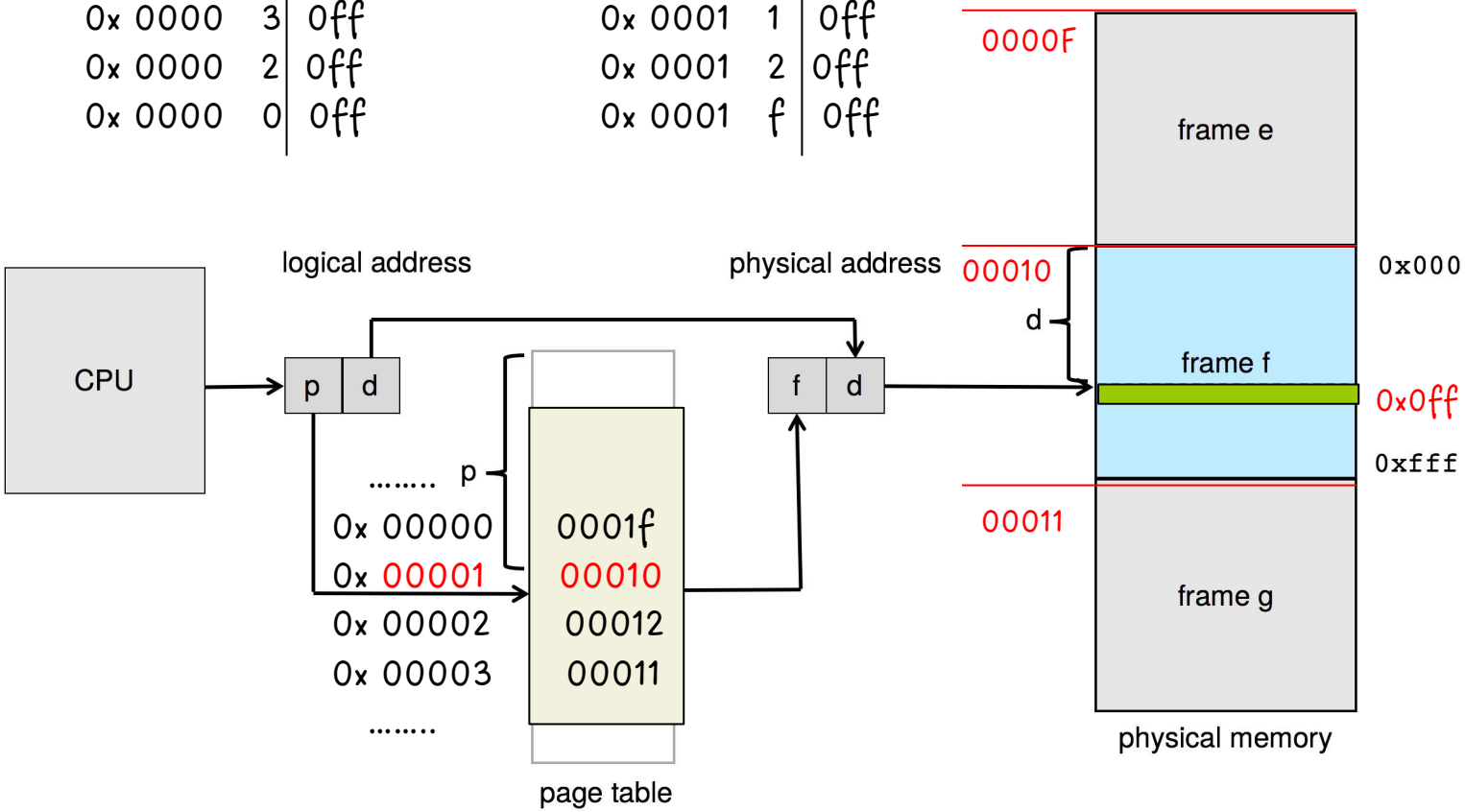


- ให้ ตัวอย่างของการแปลง logical addr ไปเป็น physical addr แล้วให้ นศ สร้าง page table

- Logical address: n = 12 and m = 32. Using a page size of 2^12 = 4KB bytes and a physical memory of 2^32 = 4GB (4*1024^3/4*1024^1 = 1024^2 = 1048576 pages)

p		d
0x 0000	1	0ff
0x 0000	3	0ff
0x 0000	2	0ff
0x 0000	0	0ff

f		d
0x 0001	0	0ff
0x 0001	1	0ff
0x 0001	2	0ff
0x 0001	f	0ff

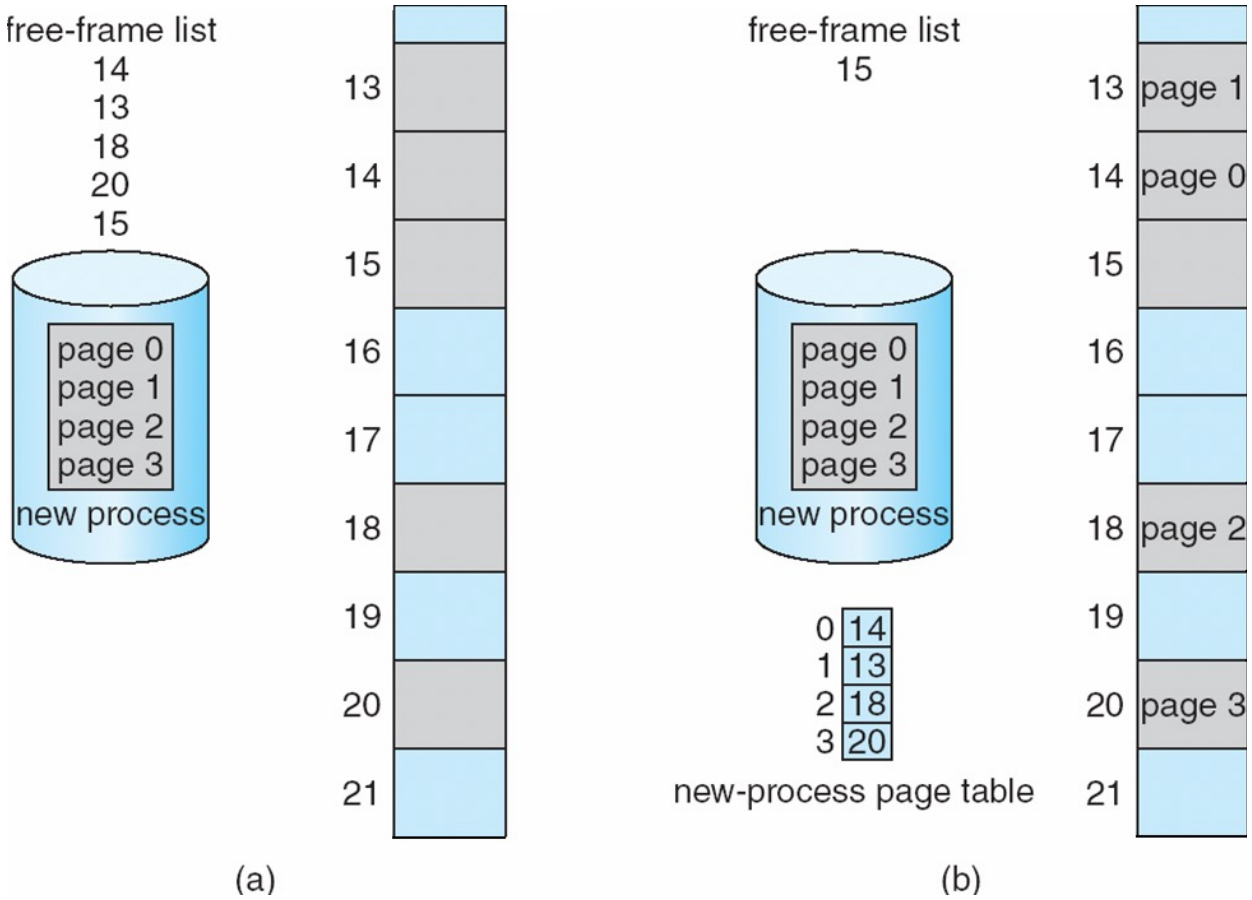


- โจทย์ ให้ ตัวอย่างของการแปลง logical addr ไปเป็น physical addr แล้วให้ นศ สร้าง page table

Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
- On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
- But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB
  - Windows 10 supports two page sizes – 4 KB and 2MB
  - Linux usual page size is 4KB
    - Linux support huge page – 2 MB to 256 MB

Free Frames



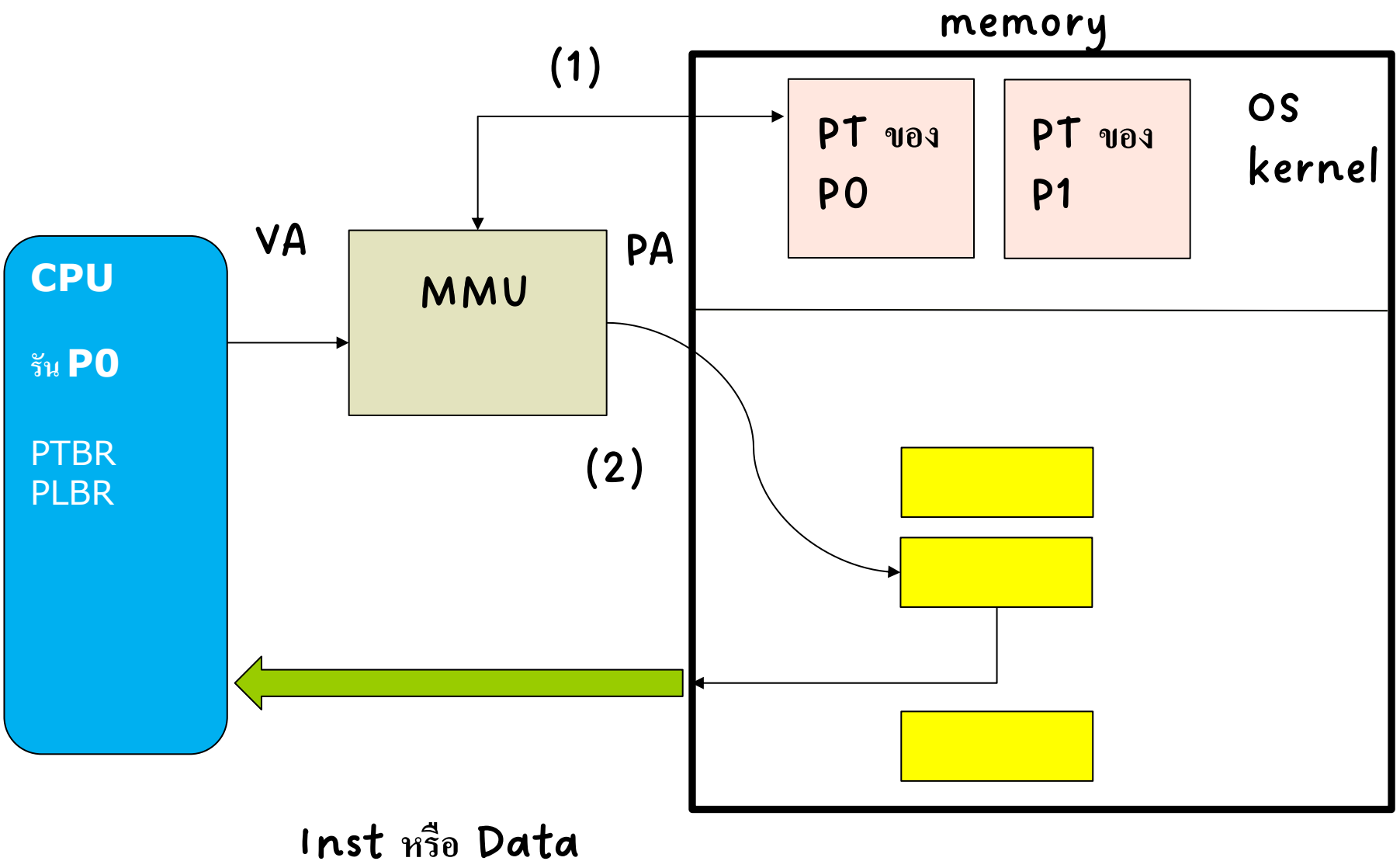
Before allocation

After allocation

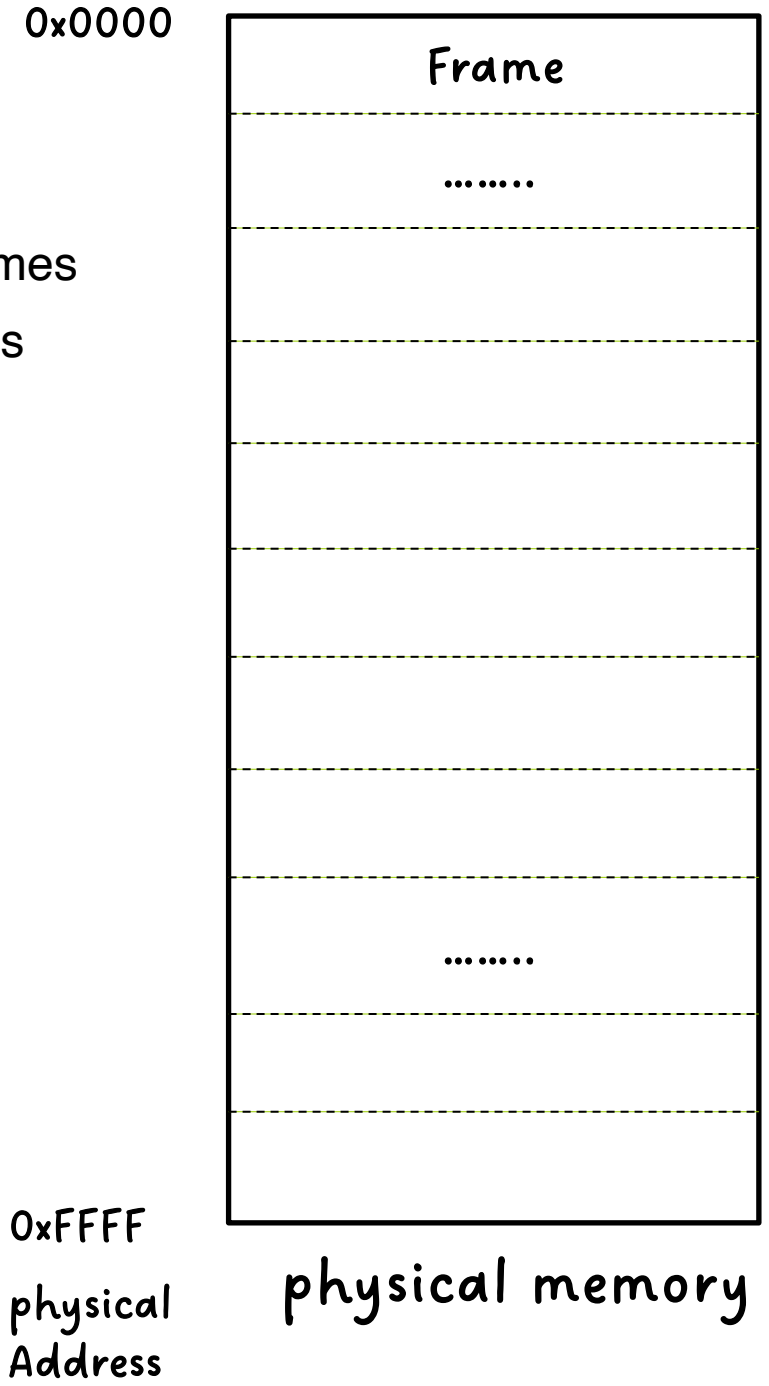
Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
    - ใน x86_64 จะใช้ CR3 register
  - **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires **two memory accesses**
  - One for the page table and one for the data / instruction
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

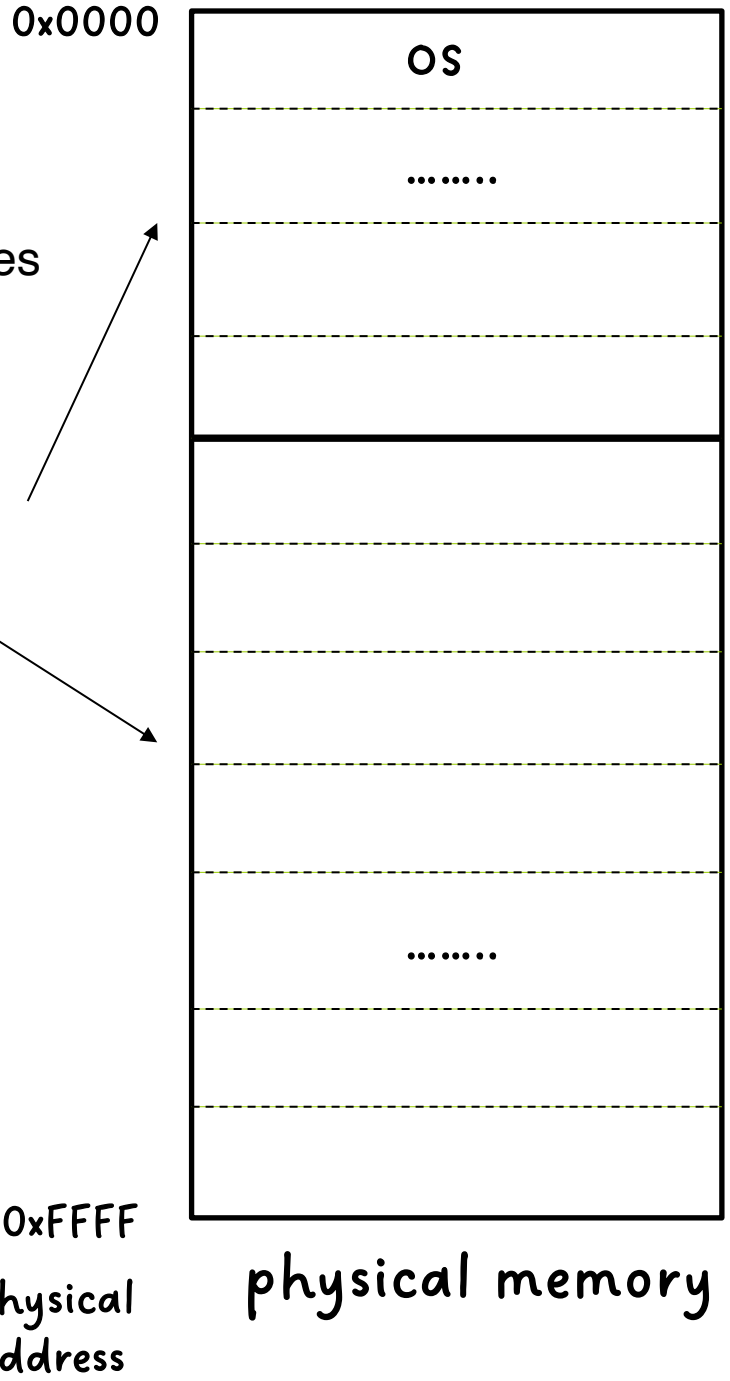
Page Table อยู่ที่ไหน



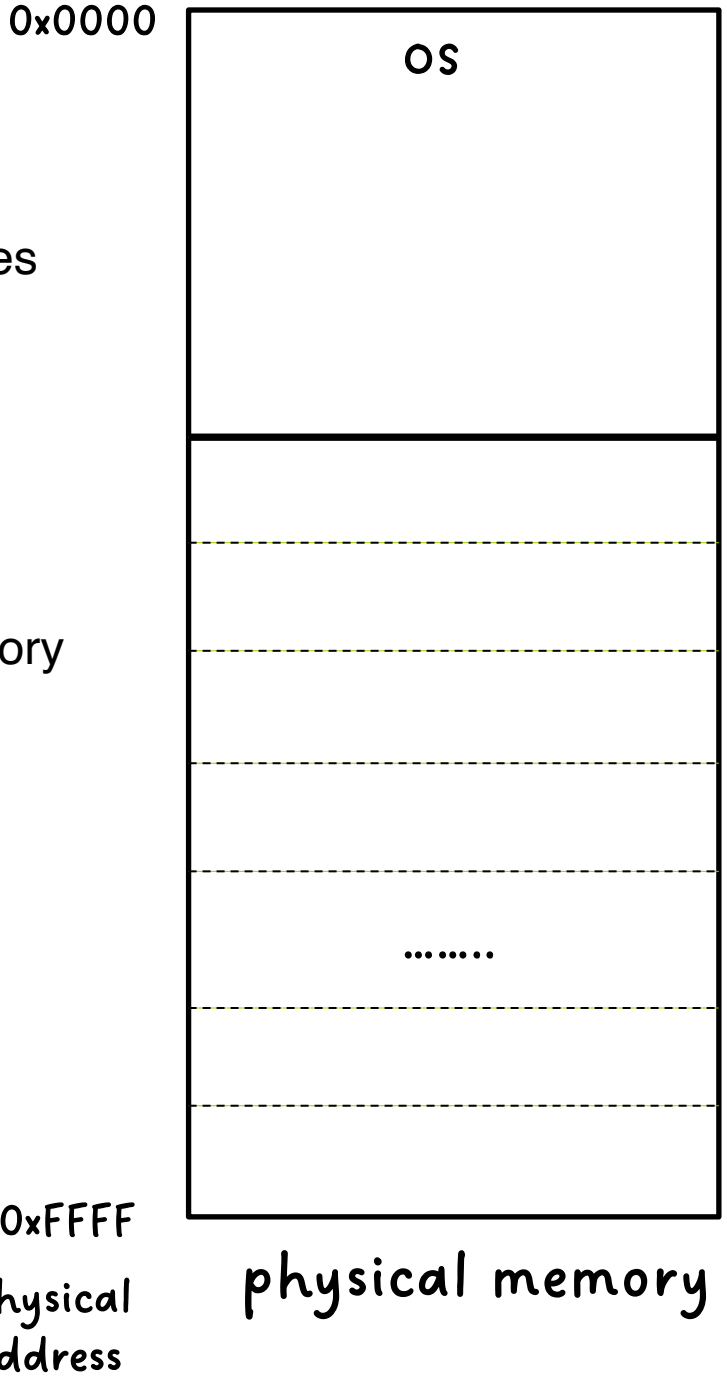
- Physical Memory Space ถูกแบ่งเป็น Frames
- Logical Memory Space ถูกแบ่งเป็น Pages
- ขนาดของ Page กับ Frame เท่ากัน



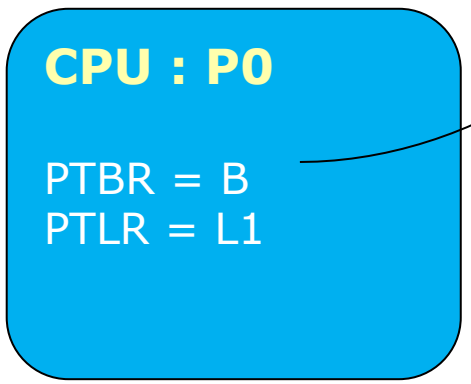
- Physical Memory Space ถูกแบ่งเป็น Frames
- Logical Memory Space ถูกแบ่งเป็น Pages
- ขนาดของ Page กับ Frame เท่ากัน
- Frames จำนวนหนึ่งจะถูกจองไว้ใช้ตลอดเวลาโดย OS
- ส่วนที่เหลือใช้สำหรับเก็บข้อมูลของ Process

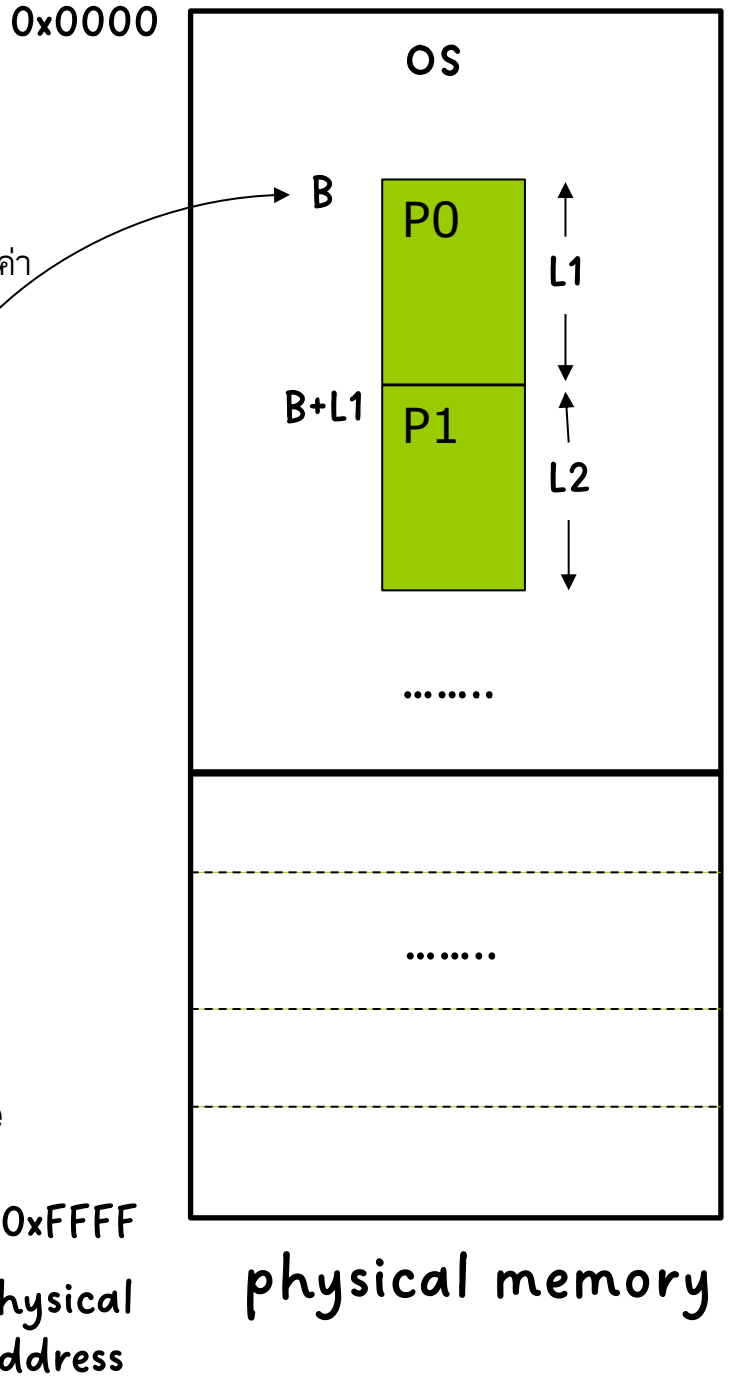


- Physical Memory Space ถูกแบ่งเป็น Frames
- Logical Memory Space ถูกแบ่งเป็น Pages
- ขนาดของ Page กับ Frame เท่ากัน
- Frames จำนวนหนึ่งจะถูกจองไว้ใช้ตลอดเวลาโดย OS
- ส่วนที่เหลือใช้สำหรับเก็บข้อมูลของ Process
- Page Table ของแต่ละ Process จะอยู่ใน memory ของ OS
- เวลาที่แต่ละ Process ประมวลผล OS จะกำหนดค่า Page-table base register (PTBR) และ Page-table length register (PTLR)



- เวลาที่แต่ละ Process P0 ประมวลผล OS จะกำหนดค่า Page-table base register (PTBR) และ Page-table length register (PTLR)





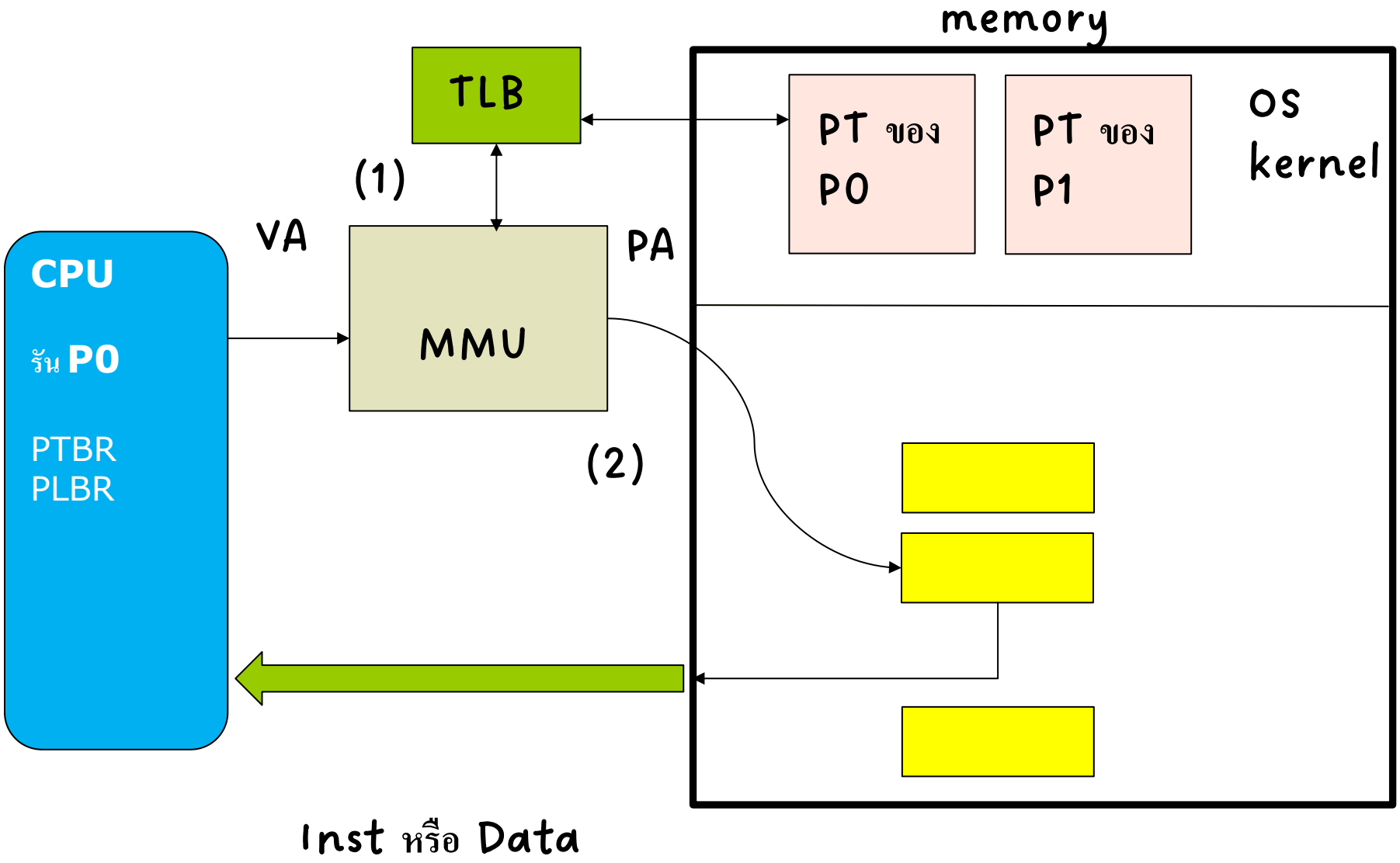
- เรามี Page-table length register (PTLR) เพราะ Process ไม่ได้ใช้ Logical Address Space ทั้งหมด และเพื่อป้องกันไม่ให้เสียพื้นที่ page table โดยเปล่าประโยชน์

Translation Look-Aside Buffer

- ปัญหาของ Page Table คือ มันทำให้การเข้าถึง Memory ครั้งหนึ่ง ต้องเข้าถึง memory ถึง 2 ครั้ง
- **Translation Look-Aside Buffer** คือ หน่วยความจำความเร็วสูง ที่ **MMU** ใช้เก็บข้อมูลการ **mapping** ของ **page table** ที่ใช้บ่อย (เหมือน **Data** หรือ **Instruction cache**)
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

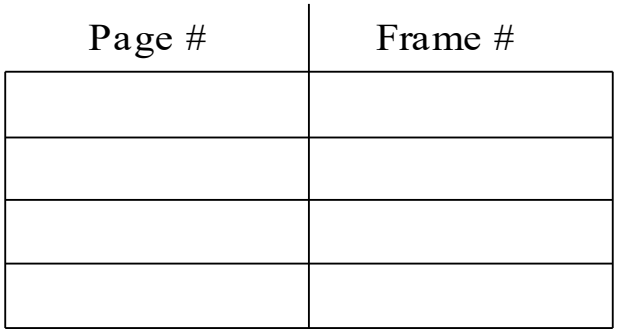
    บาง address entry สามารถถูกเก็บถาวรใน TLB เช่นตำแหน่งของ kernel code

TLB อยู่ที่ไหน


memory
TLB
PT ของ P0
PT ของ P1
OS kernel
(1)
VA
PA
CPU
รัน P0
PTBR
PLBR
MMU
(2)
Inst หรือ Data

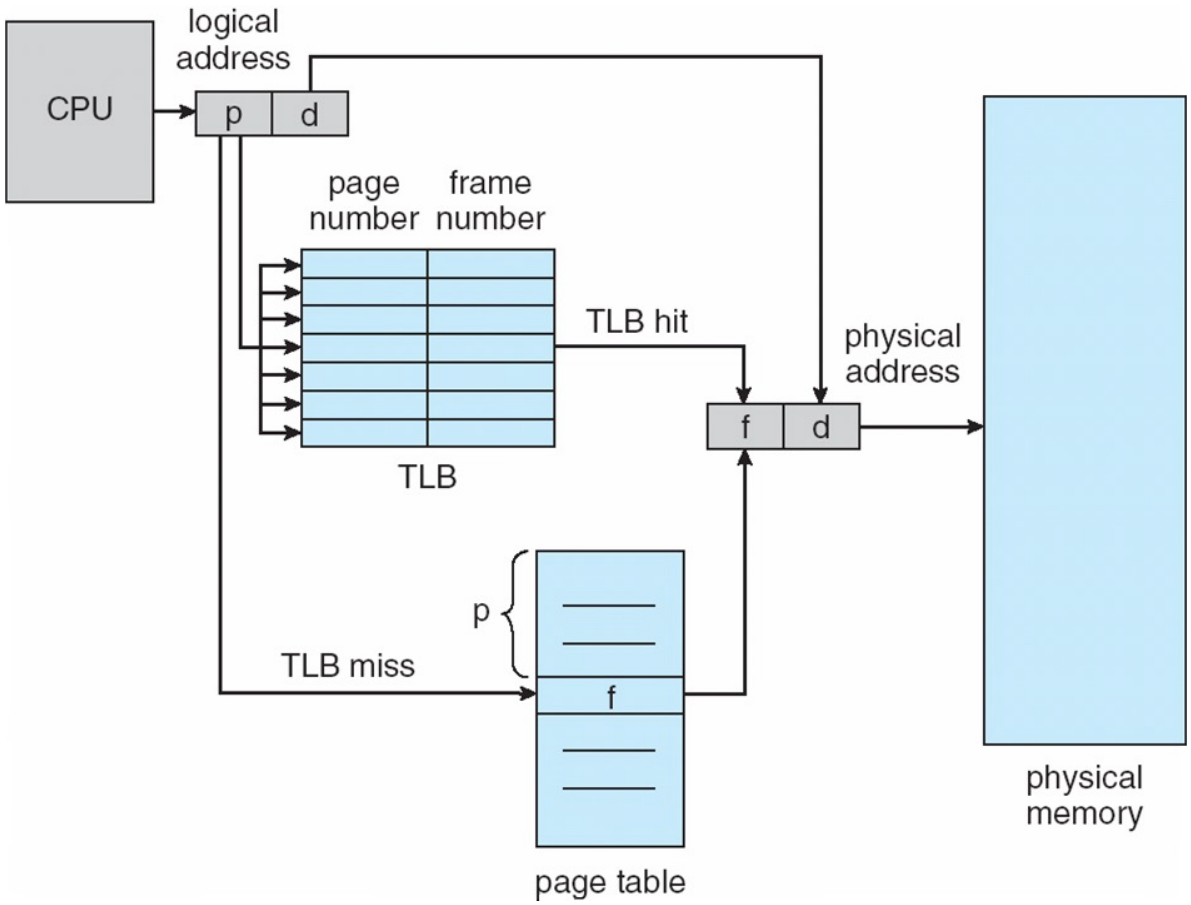
Hardware

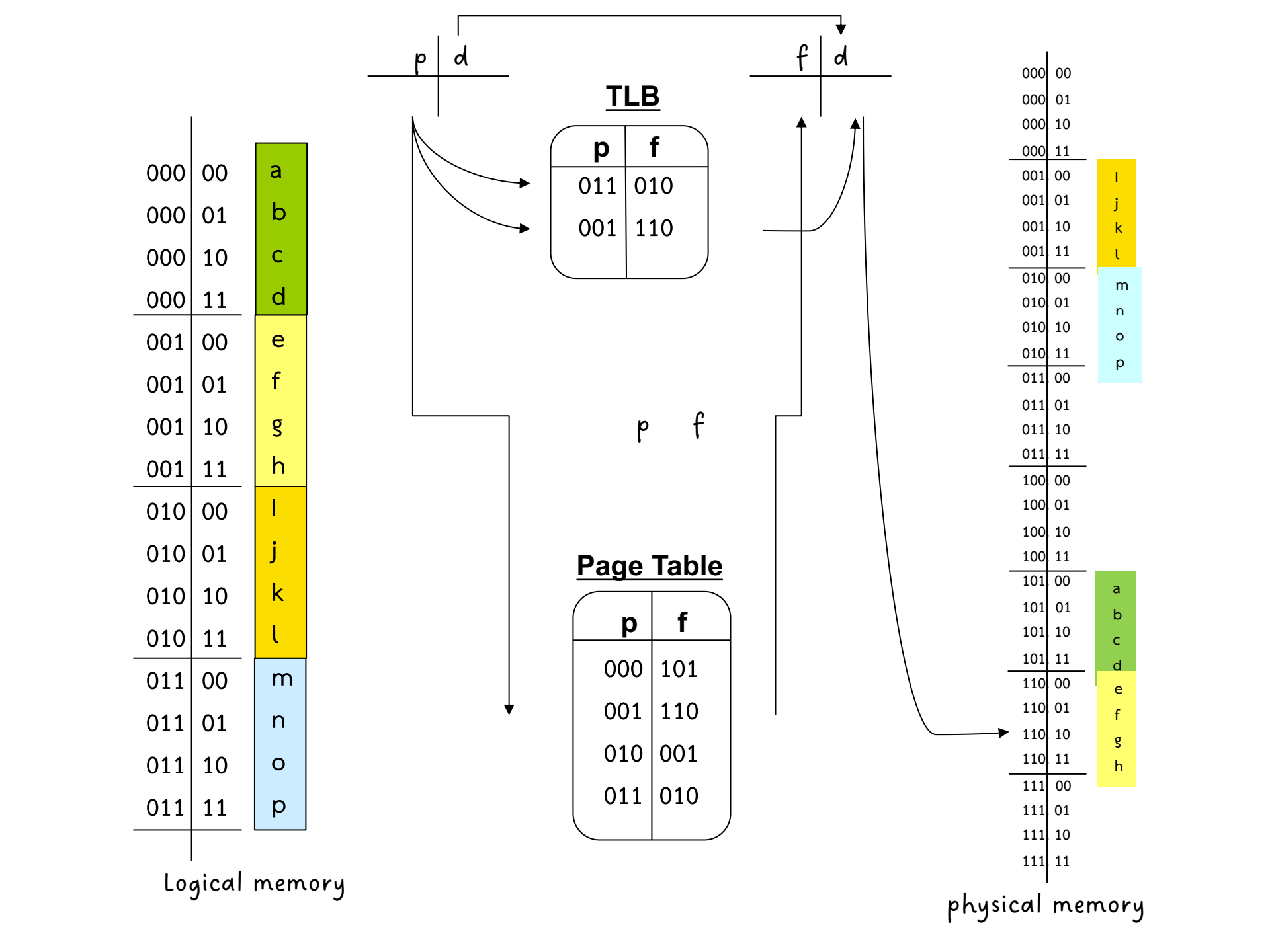
- Associative memory – parallel search

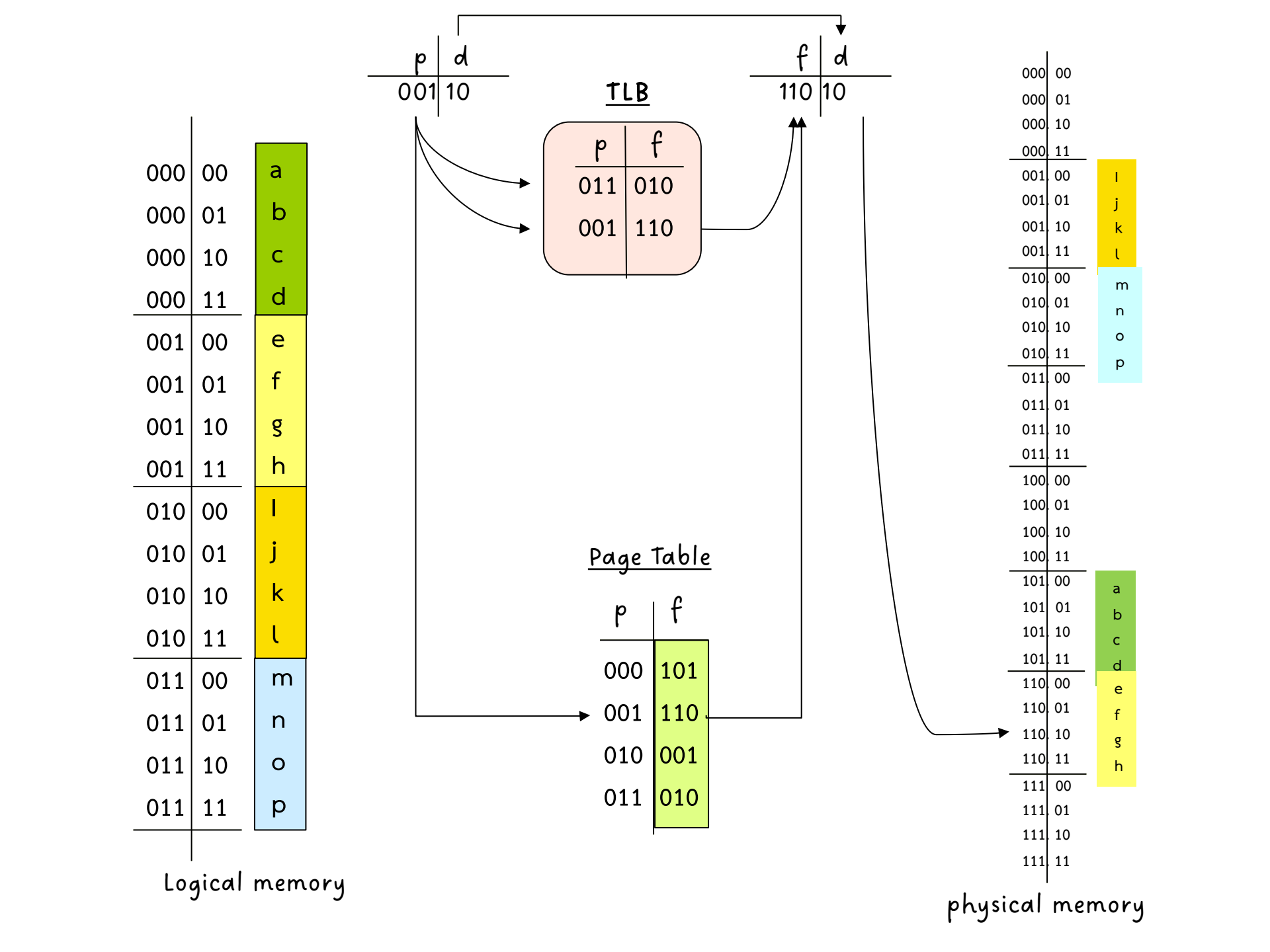


- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB




p
d
f
d
TLB
p
f
011
010
001
110
p
f
Page Table
p
f
000
101
001
110
010
001
011
010
000 00
000 01
000 10
000 11
001 00
001 01
001 10
001 11
010 00
010 01
010 10
010 11
011 00
011 01
011 10
011 11
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
Logical memory
000 00
000 01
000 10
000 11
001 00
001 01
001 10
001 11
010 00
010 01
010 10
010 11
011 00
011 01
011 10
011 11
100 00
100 01
100 10
100 11
101 00
101 01
101 10
101 11
110 00
110 01
110 10
110 11
111 00
111 01
111 10
111 11
physical memory


p d
001 10
TLB
p f
011 010
001 110
f d
110 10
Page Table
p f
000 101
001 110
010 001
011 010
000 00 a
000 01 b
000 10 c
000 11 d
001 00 e
001 01 f
001 10 g
001 11 h
010 00 I
010 01 j
010 10 k
010 11 l
011 00 m
011 01 n
011 10 o
011 11 p
Logical memory
000 00
000 01
000 10
000 11
001 00 I
001 01 j
001 10 k
001 11 l
010 00 m
010 01 n
010 10 o
010 11 p
011 00
011 01
011 10
011 11
100 00
100 01
100 10
100 11
101 00 a
101 01 b
101 10 c
101 11 d
110 00 e
110 01 f
110 10 g
110 11 h
111 00
111 01
111 10
111 11
physical memory

Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.
- Intel Core i7 มี 128 entries L1 Instruction TLB และ 64 entries L1 data TLB และมี 512 entries L2 TLB

TLB: ASID

- Address Space Identifier (ASID) เป็นตัวเลขที่ OS กำหนดให้แต่ละ Process
- ทำให้สามารถใช้ TLB เก็บ mapping page# → frame # ของหลาย Process ได้
- Associative memory – parallel search

Page #	ASID #	Frame #

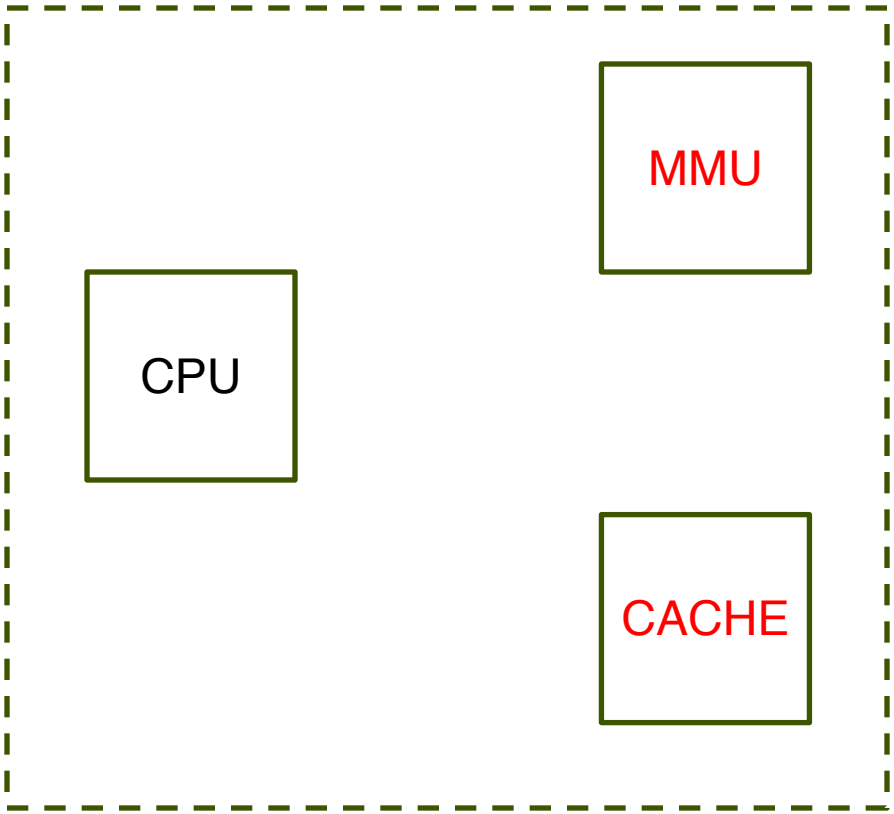
- Address translation (p, a, d)
  - If p is in associative register and a เป็น id ของ Process ที่ใช้งาน CPU อยู่,

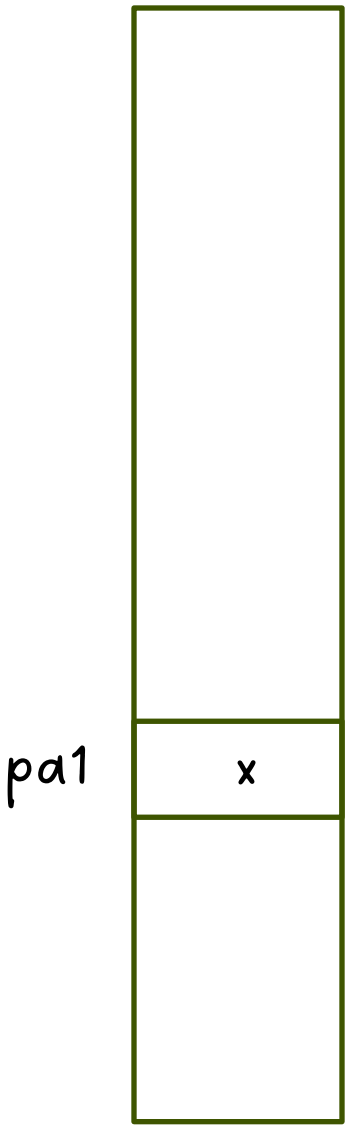
    get frame # out
  - Otherwise get frame # from page table in memory
- ASID ช่วยให้ OS ไม่ต้อง Flush TLB ทุกครั้งที่มีการ context switching process ใหม่เข้ามาใช้งาน CPU

การเข้าถึงข้อมูลที่ virtual addr = va1

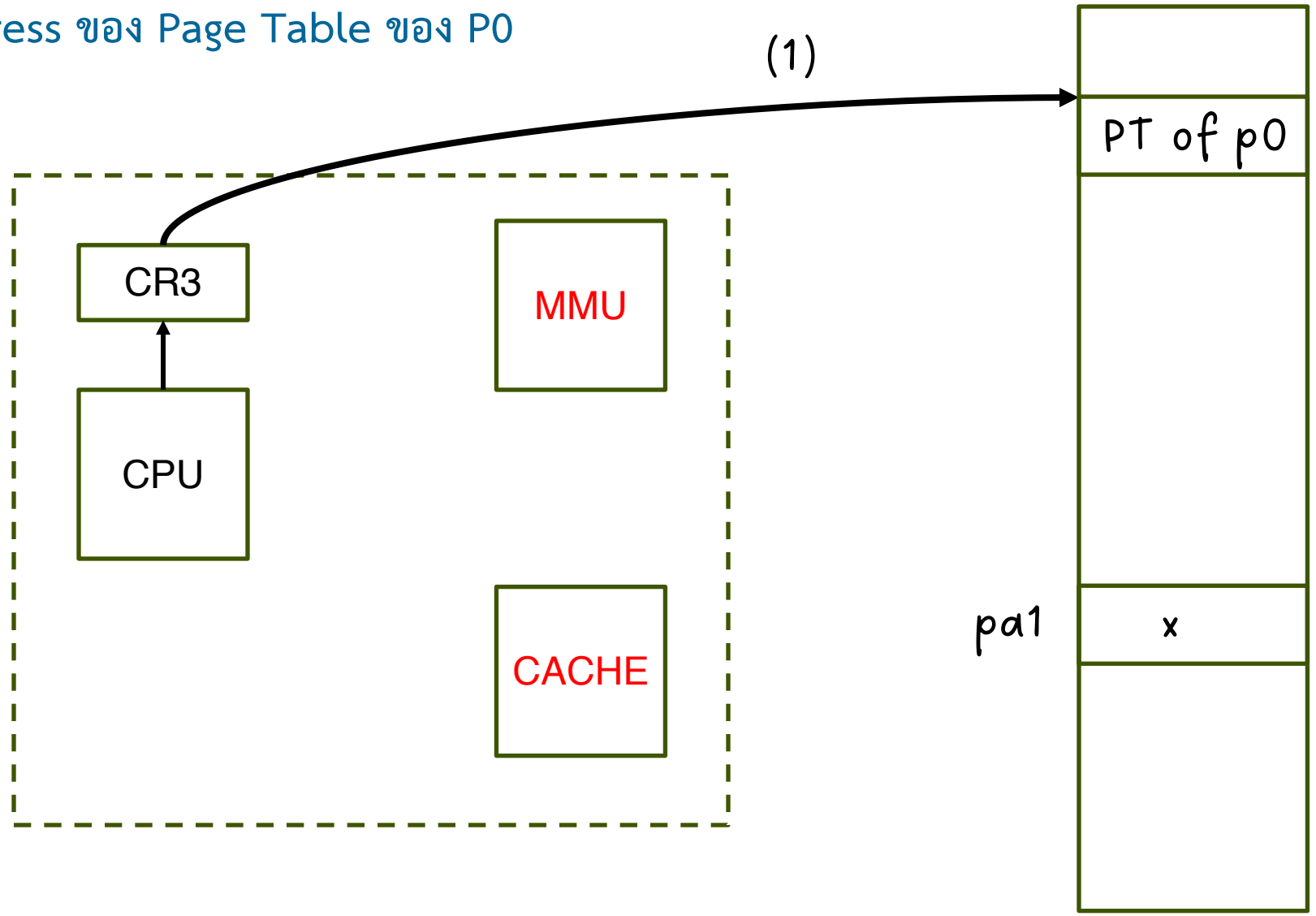
สมมุติว่า va1 คือ physical addr = pa1

ซึ่งเก็บข้อมูลคือ x



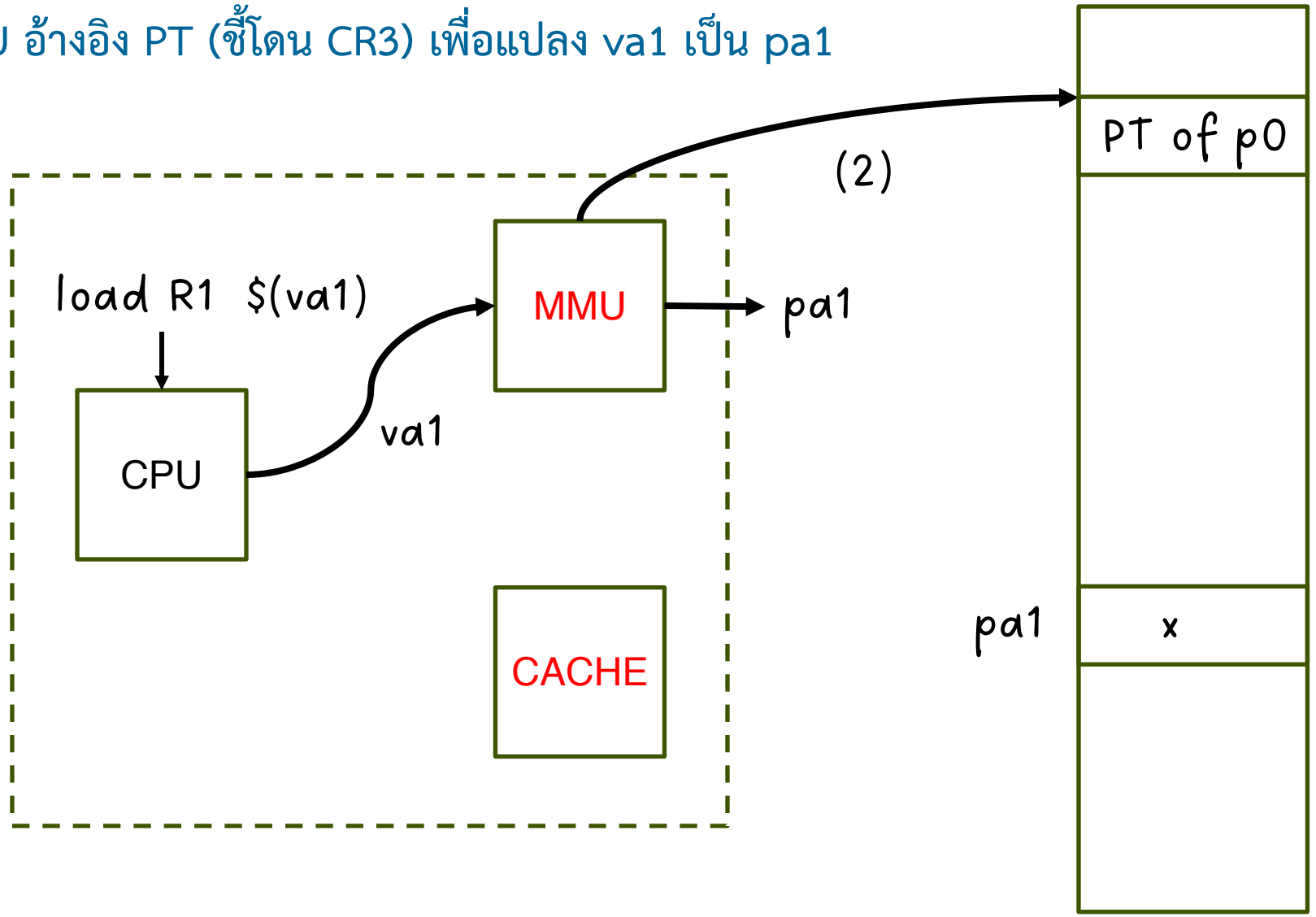


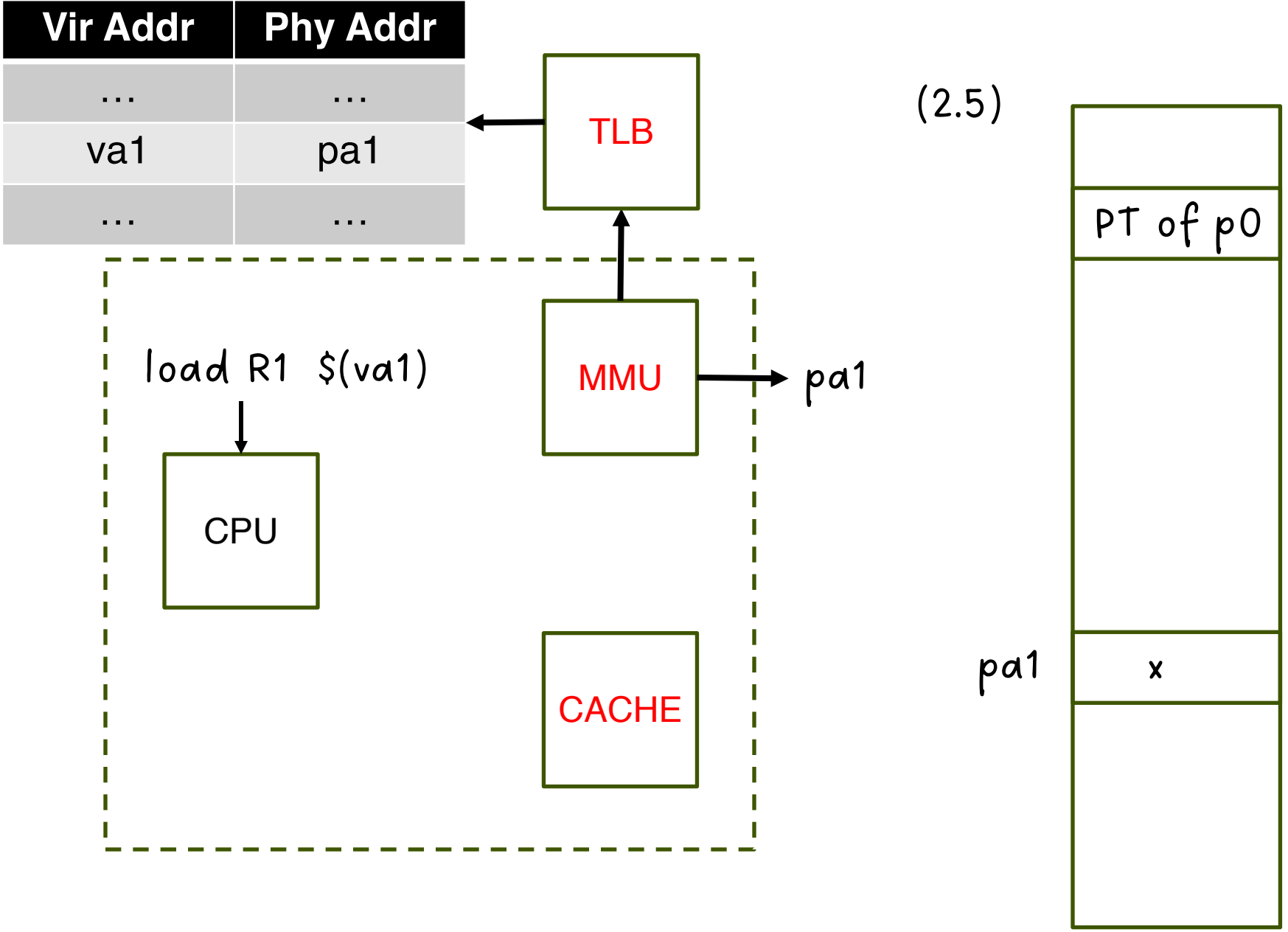
(1) กำหนดให้ CR3 ชี้ที่ physical memory address ของ Page Table ของ P0



(2) cpu รันคำสั่งอ่านค่าจาก va1 มายัง register R1

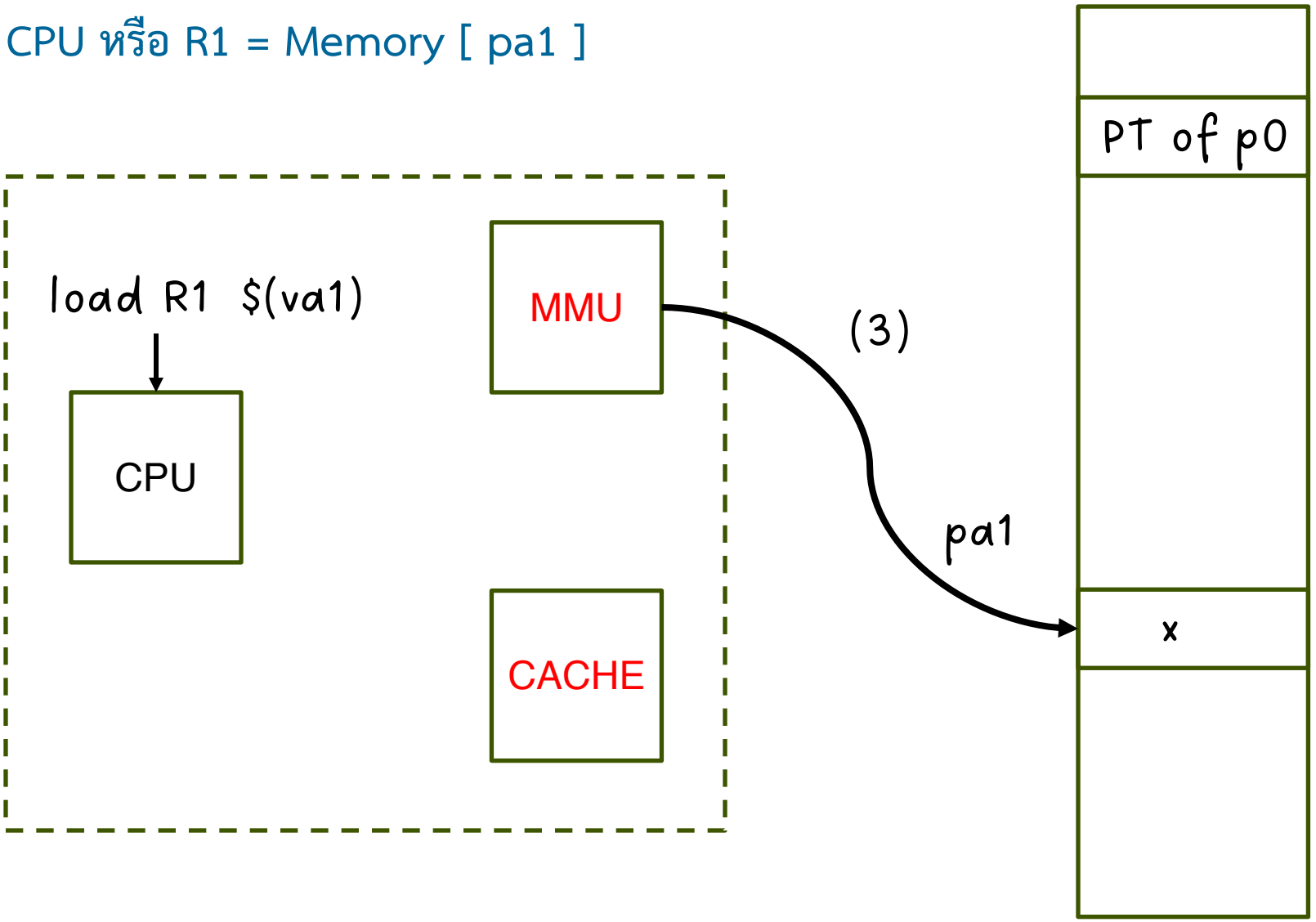
MMU อ้างอิง PT (ชี้โดน CR3) เพื่อแปลง va1 เป็น pa1



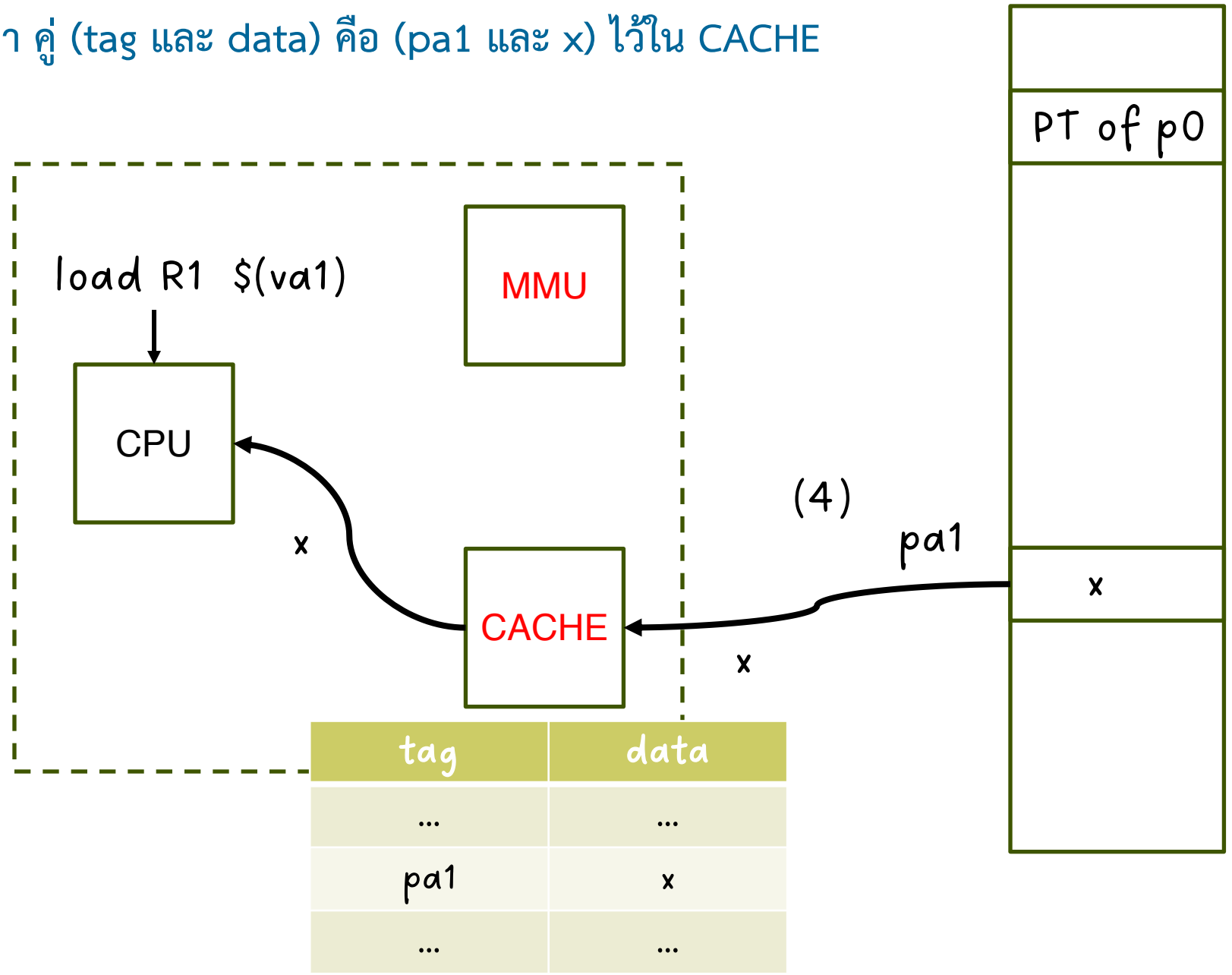


(2.5) MMU นำค่า va1 และ pa1 ไปเก็บใน TLB (เรียกอีกอย่างว่า CACHE ของ Memory Address Translation

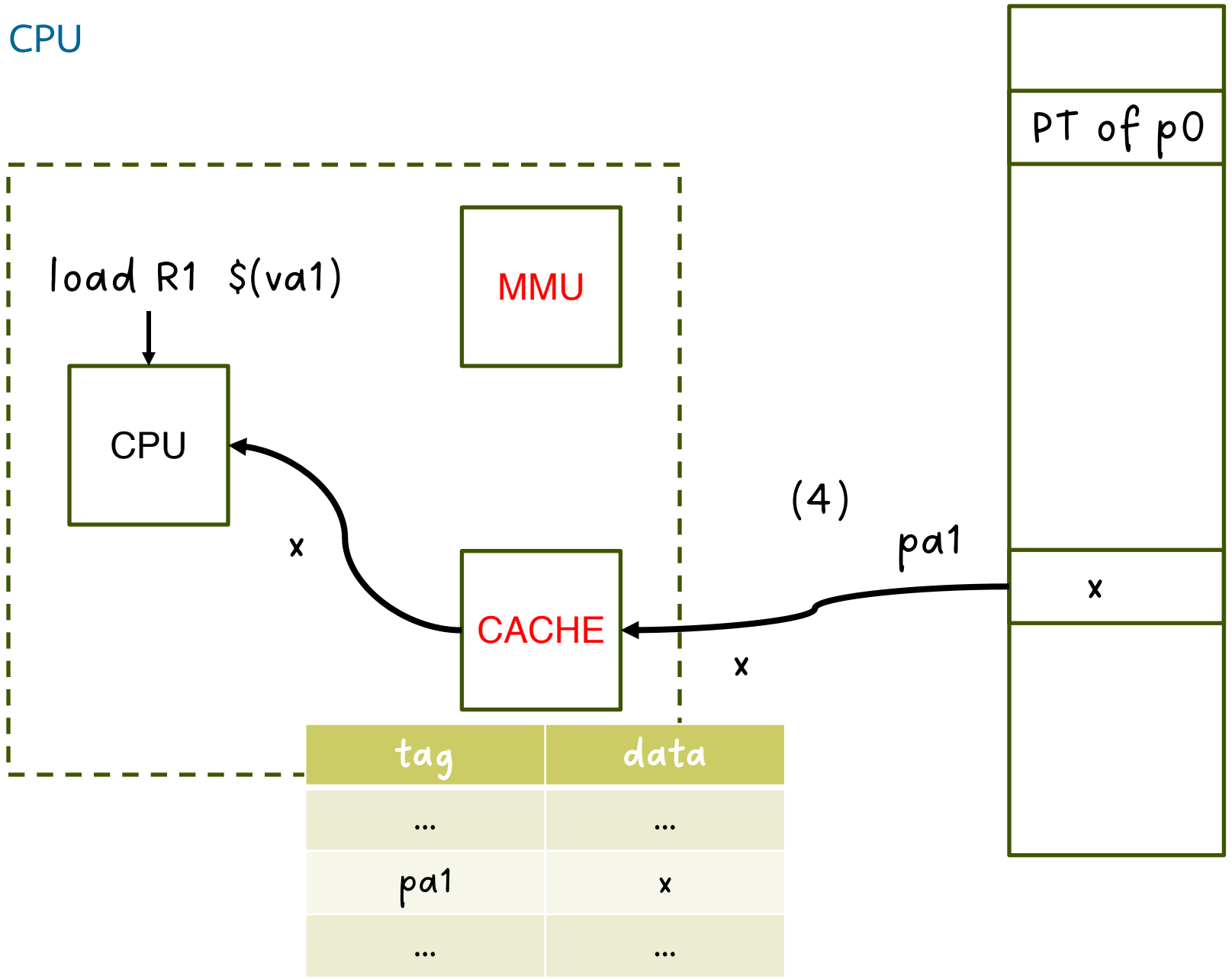
(3) MMU ส่งคำขอที่ memory เพื่อ ขอให้ส่งค่าที่เก็บใน pa1 มายัง CPU หรือ R1 = Memory [pa1]

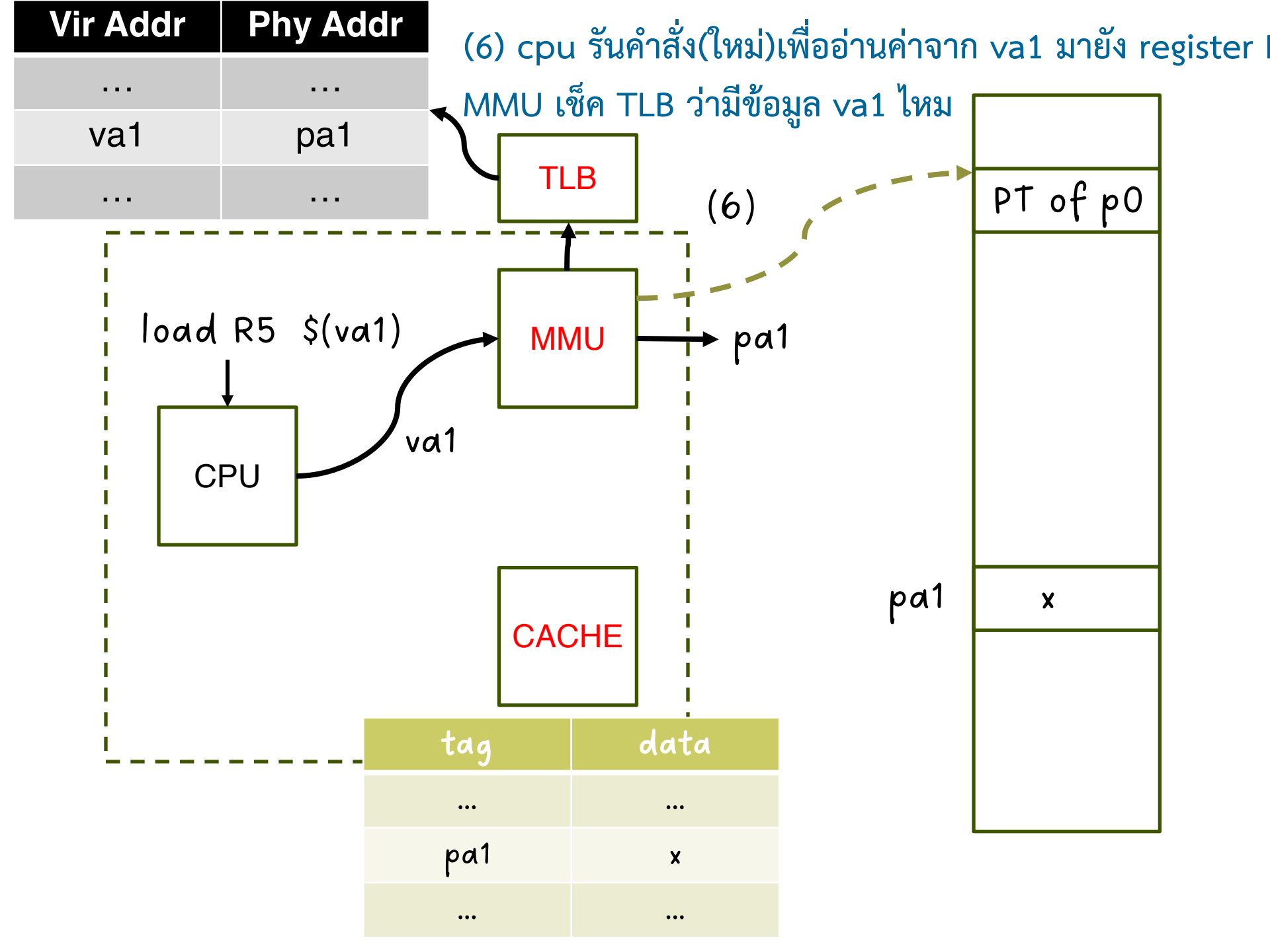


(4) Memory ส่งค่า x มาให้กับ CPU ผ่านระบบ CACHE ซึ่งเก็บค่า คู่ (tag และ data) คือ (pa1 และ x) ไว้ใน CACHE

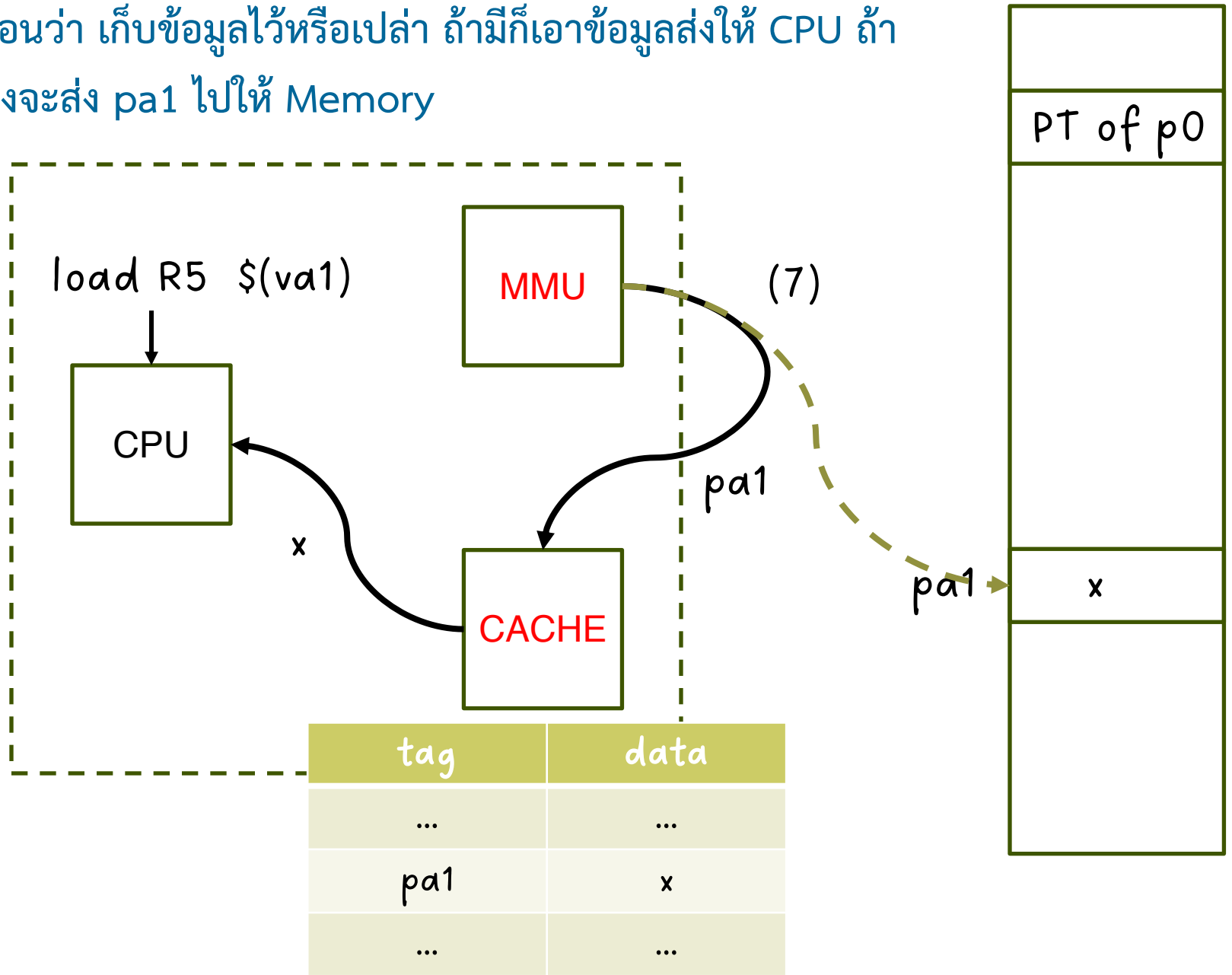


(5) MMU ส่งคำขอที่ memory เพื่อ ขอให้ส่งค่าที่เก็บใน pa1 มายัง CPU




(6) cpu รันคำสั่ง(ใหม่)เพื่ออ่านค่าจาก va1 มายัง register
MMU เช็ค TLB ว่ามีข้อมูล va1 ไหม
Vir Addr
Phy Addr
...
...
va1
pa1
...
...
TLB
(6)
PT of p0
load R5 $(va1)
MMU
pa1
CPU
va1
pa1
x
CACHE
tag
data
...
...
pa1
x
...
...

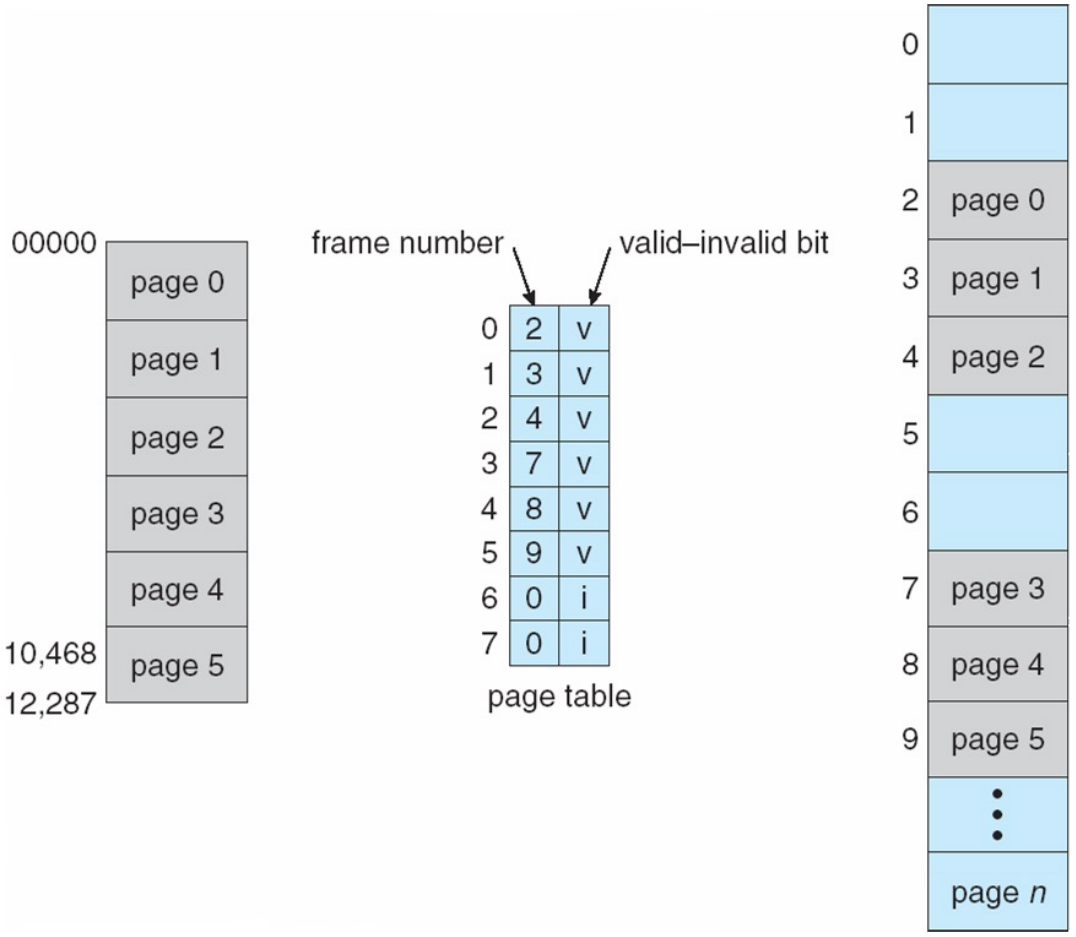
(7) MMU จะส่ง physical address pa1 ไปให้ระบบ CACHE เช็คก่อนว่า เก็บข้อมูลไว้หรือเปล่า ถ้ามีก็เอาข้อมูลส่งให้ CPU ถ้าไม่มีถึงจะส่ง pa1 ไปให้ Memory



Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table

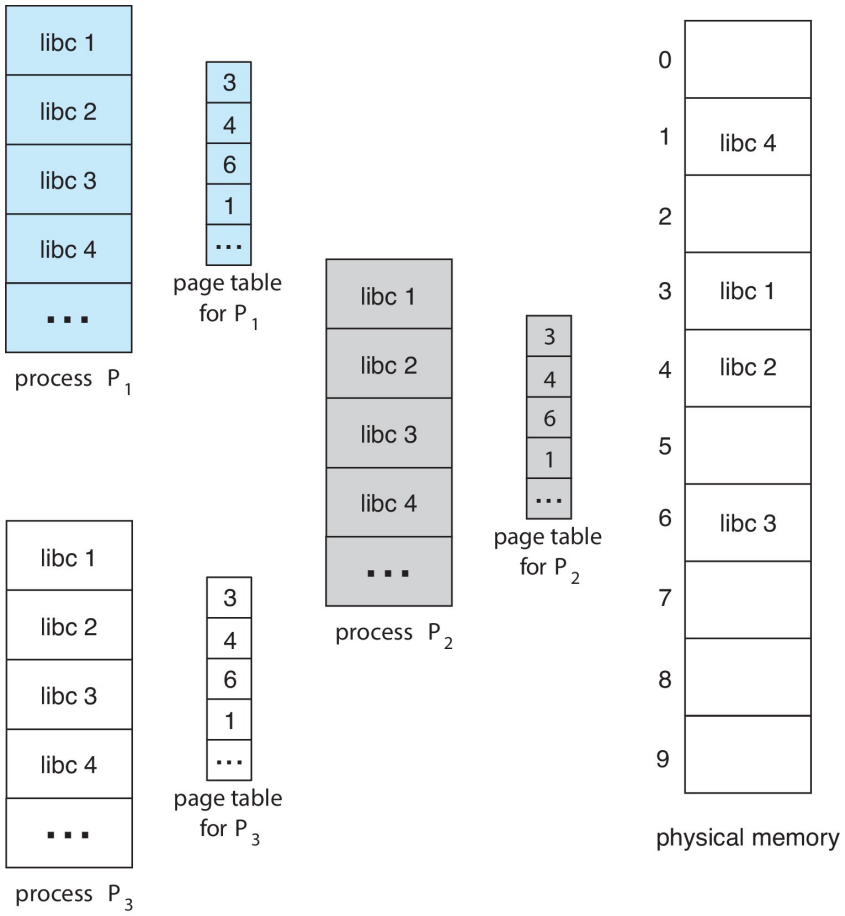


Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - *Reentrant code is non-self-modifying code: it never changes during execution.*
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

If a system has 40 user processes, and the libc library is 2 MB, this would require 80 MB of memory.

Shared Pages Example

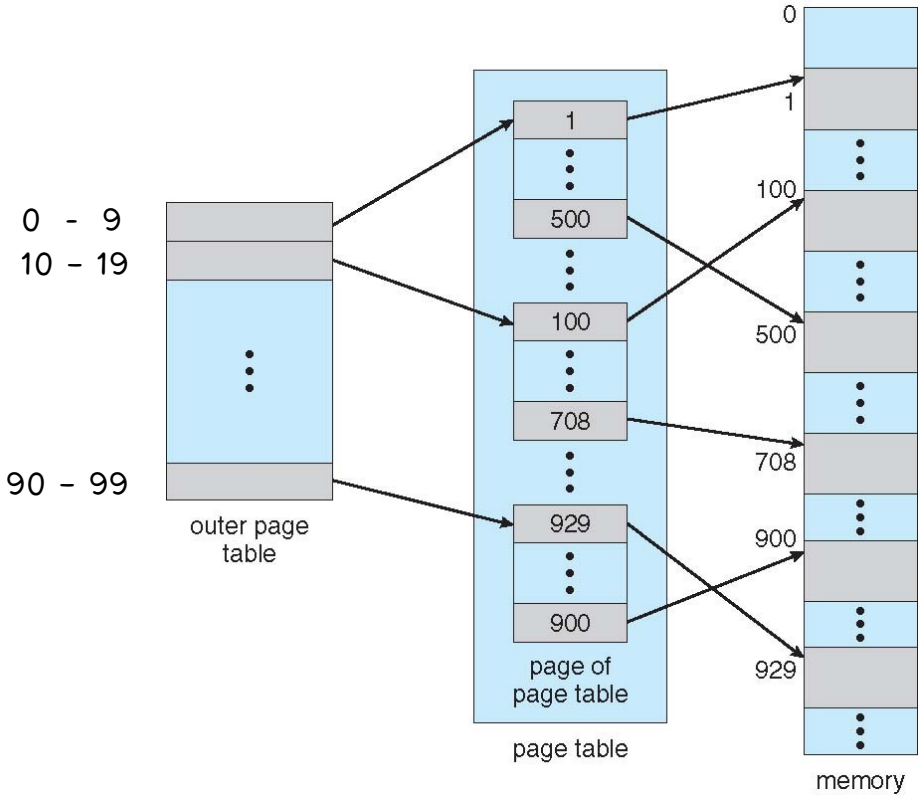


Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes ➔ each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

Hierarchical Page Tables

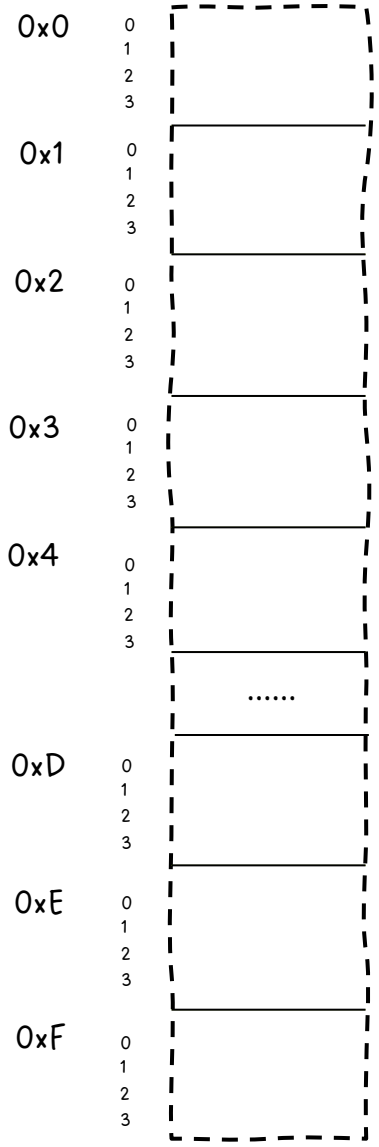
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



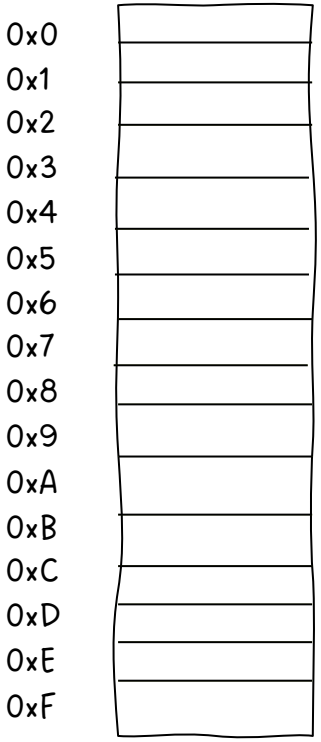
Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)

Page id (p) : offset (d)

4 bits : 2 bits

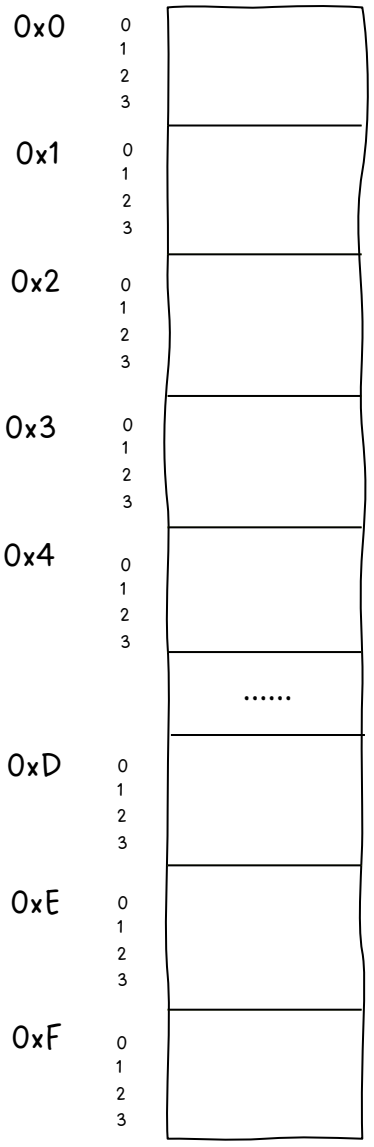


Page id (p)

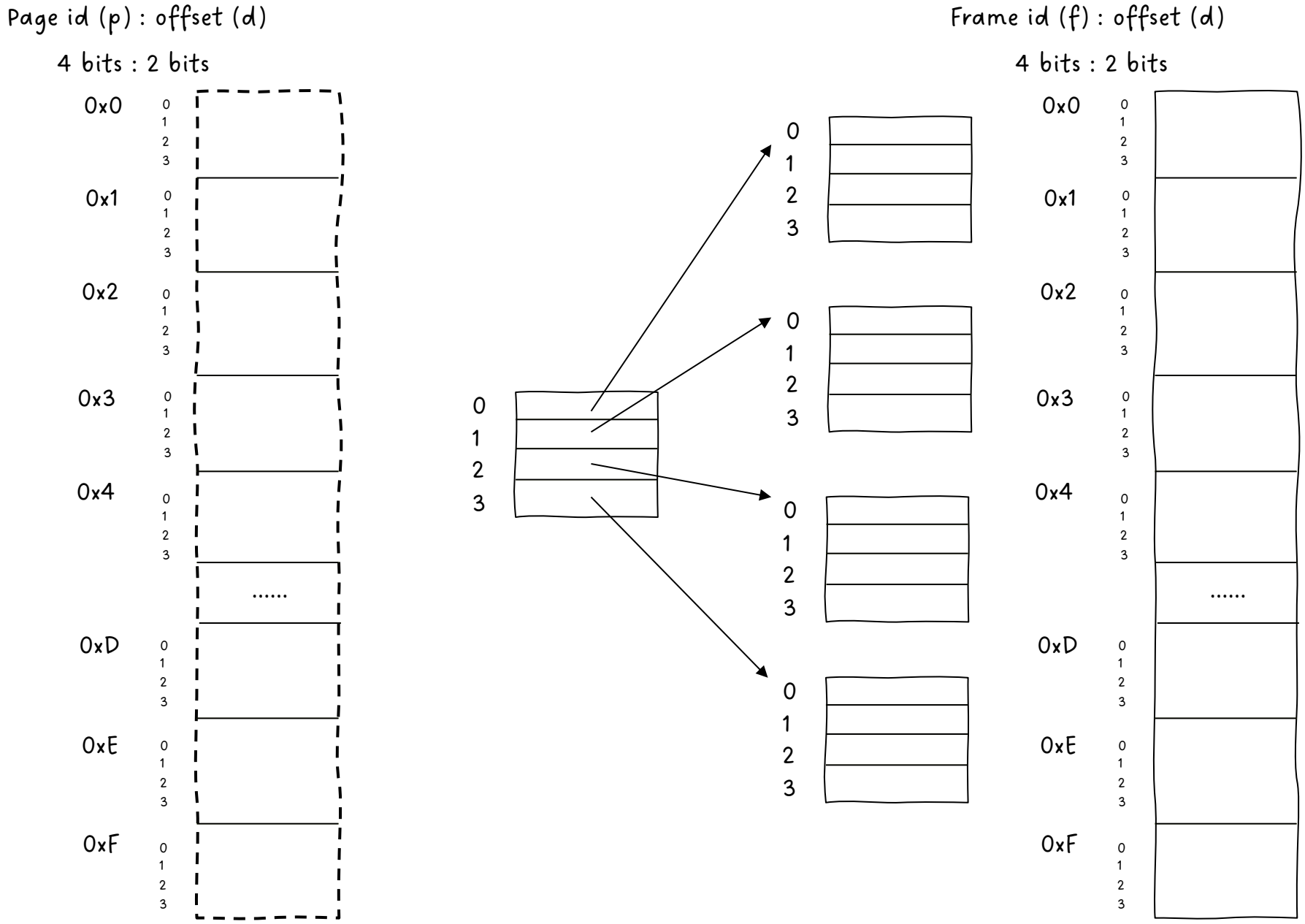


Frame id (f) : offset (d)

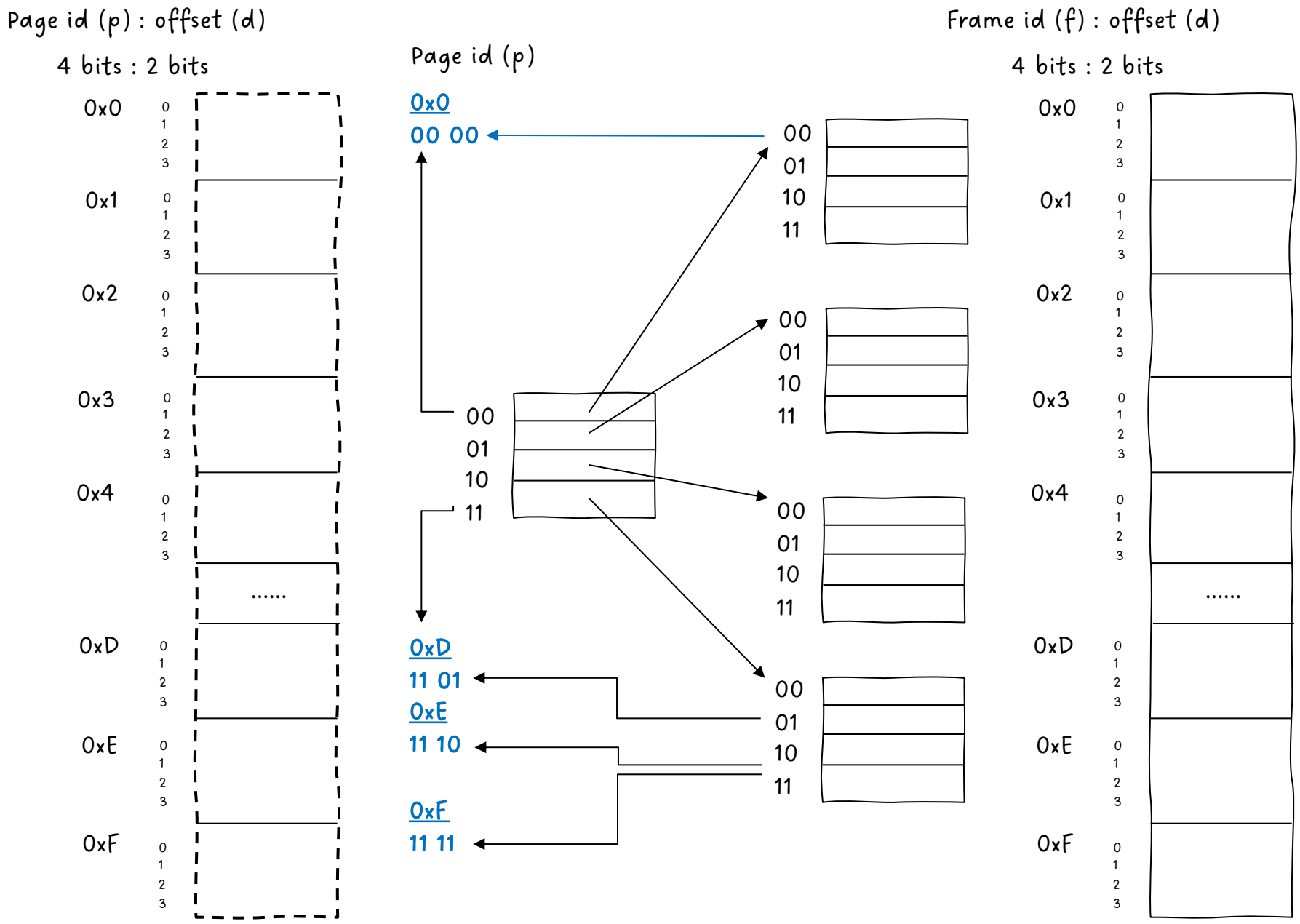
4 bits : 2 bits



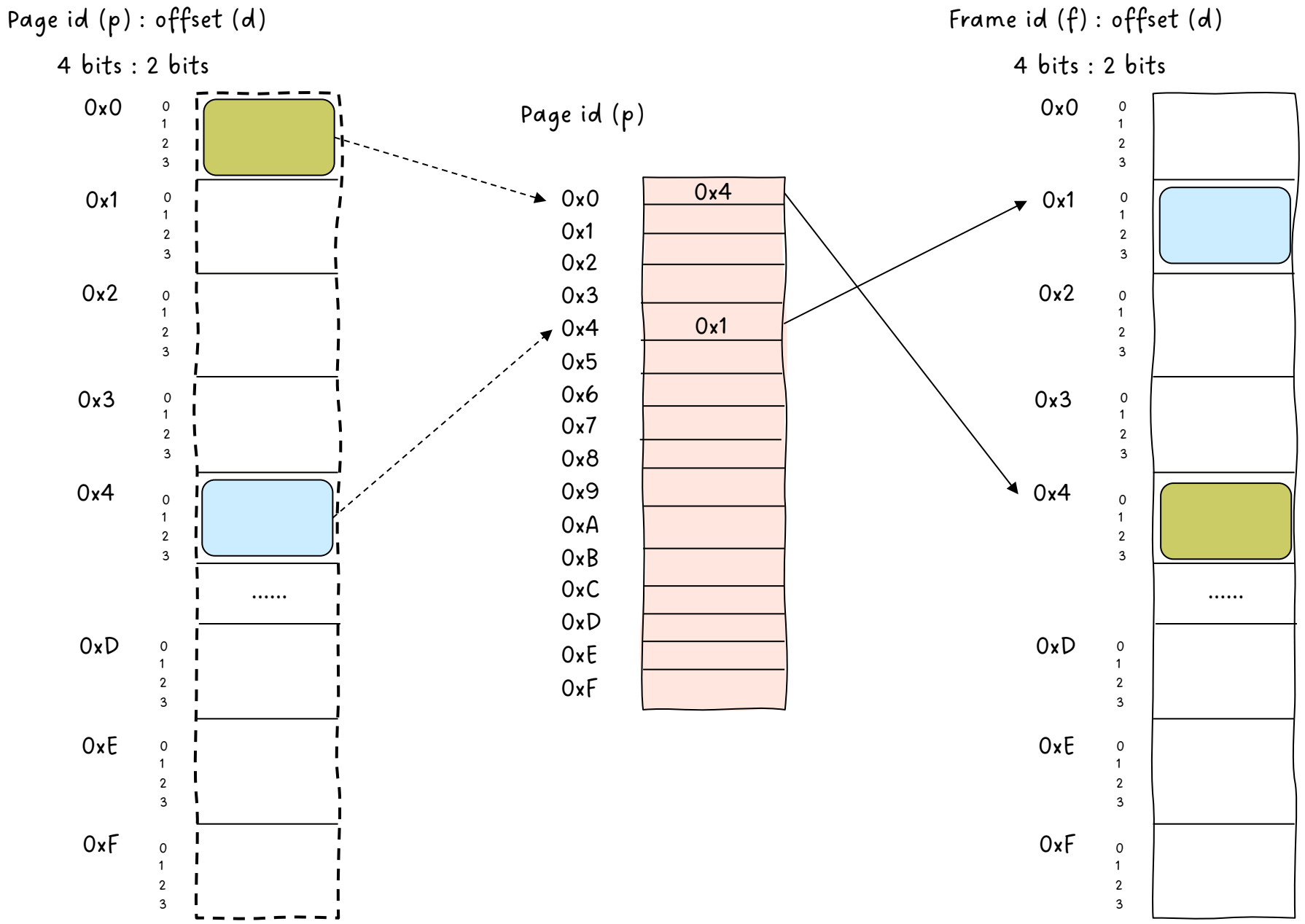
Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)



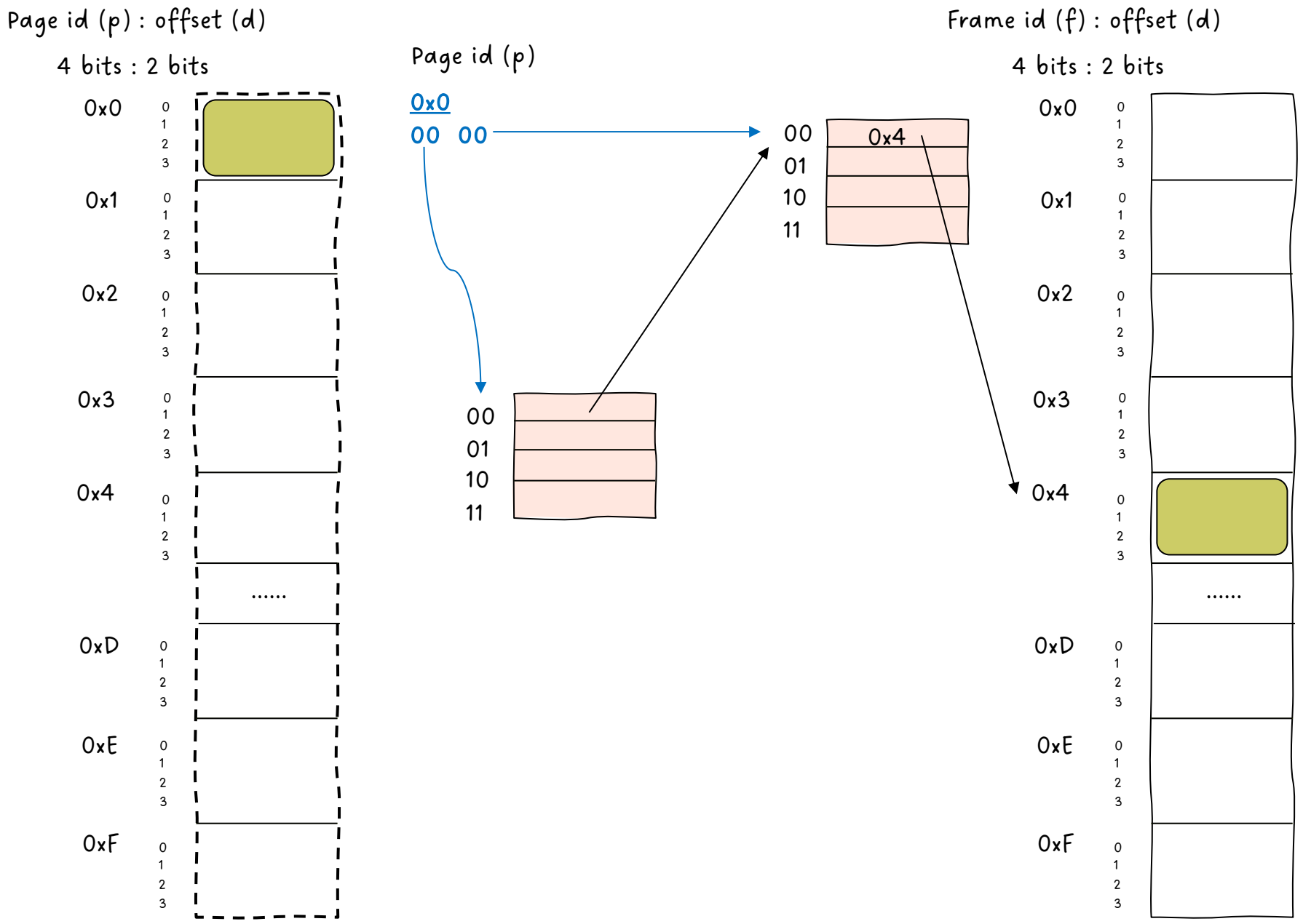
Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)



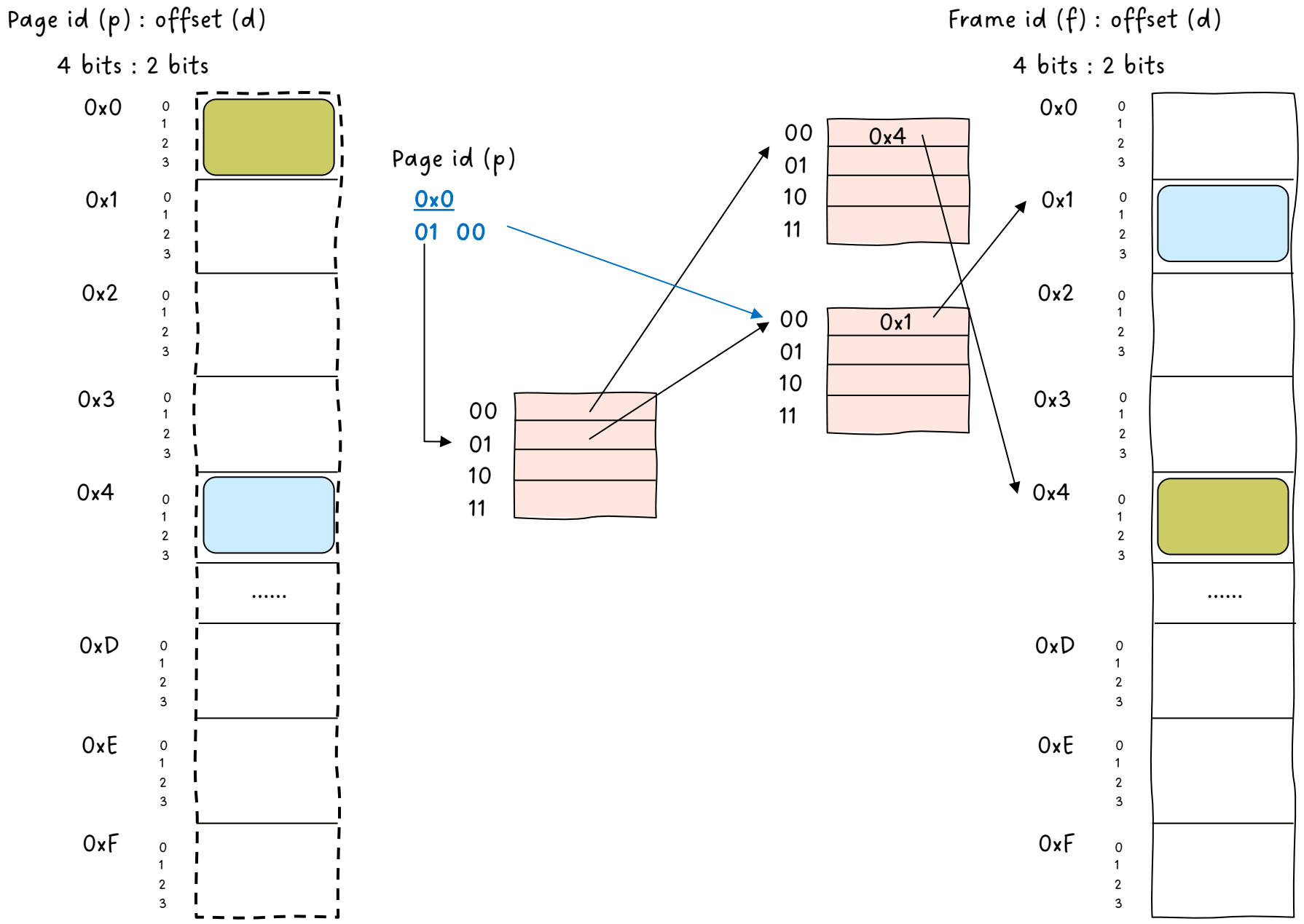
Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)



Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)

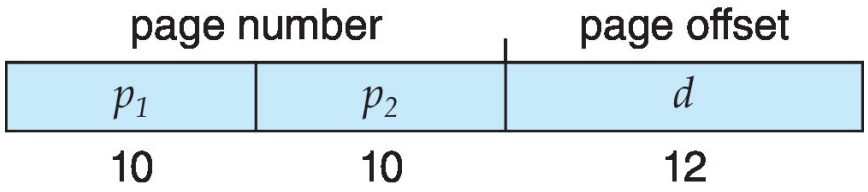


Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)



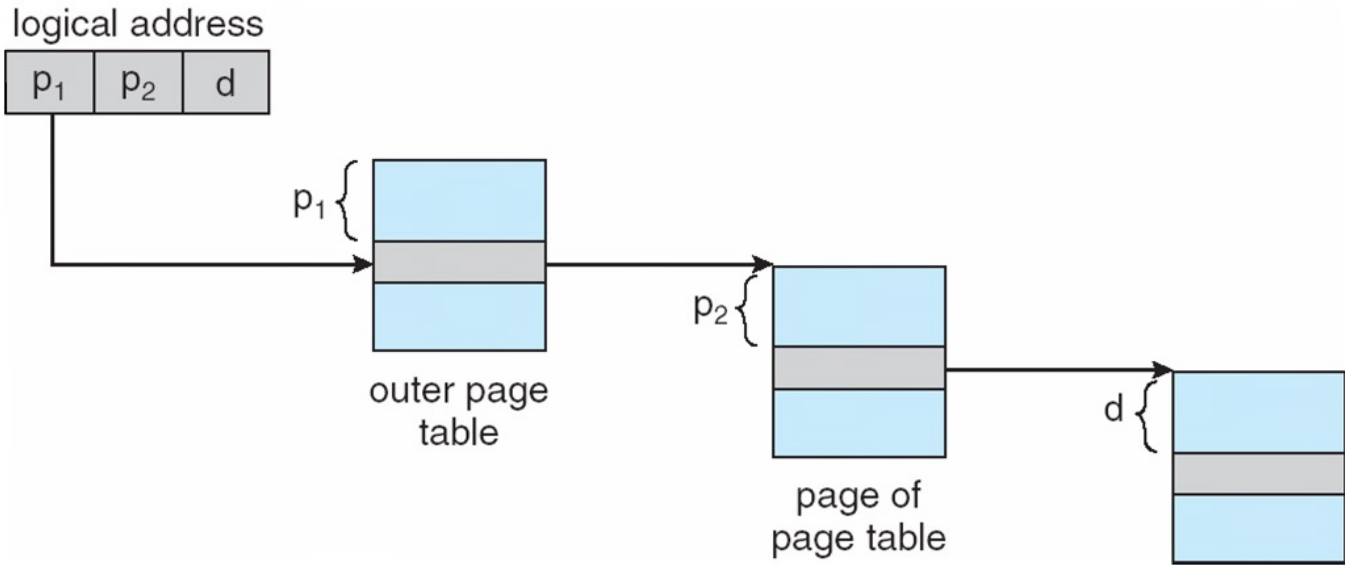
Two-Level Paging Example

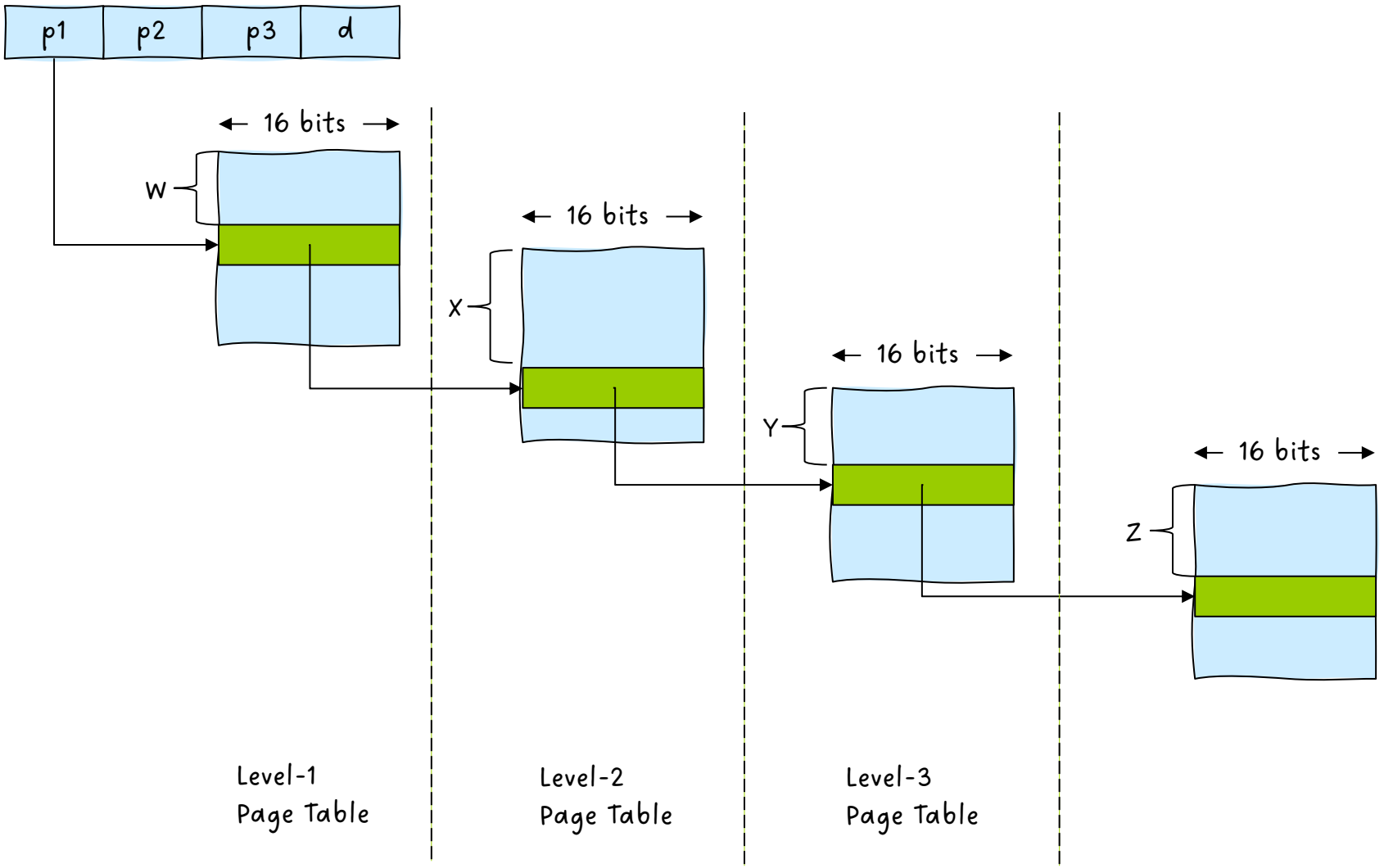
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



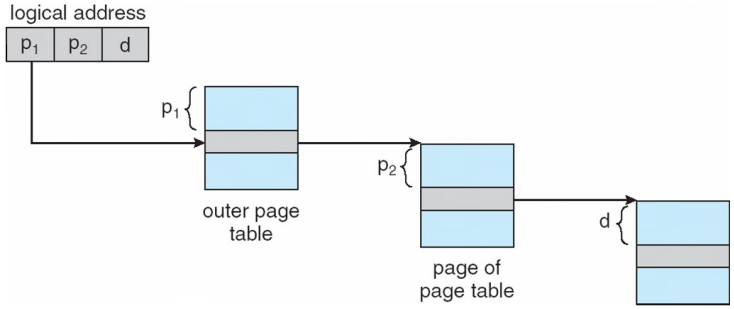
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



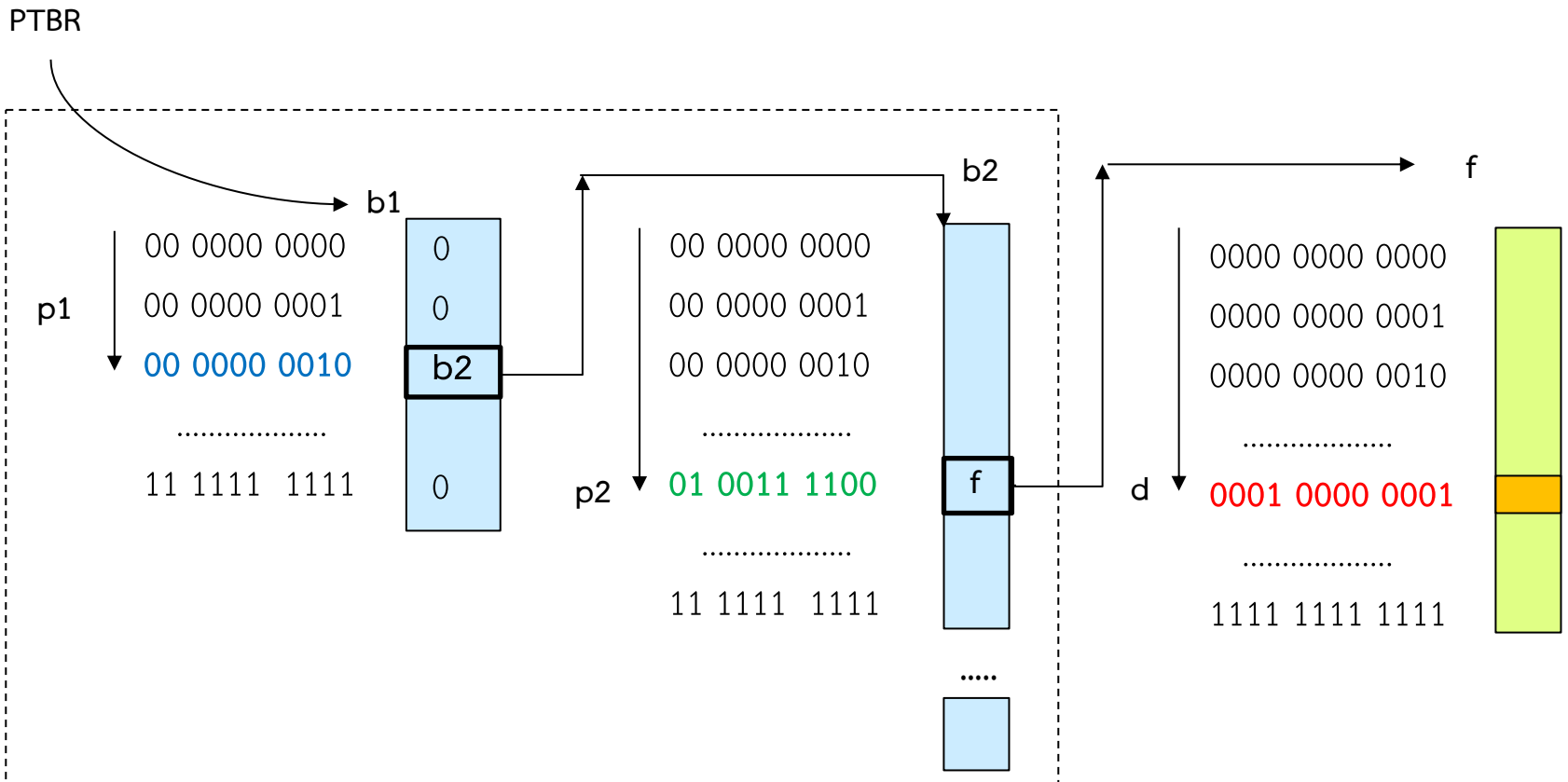

p1
p2
p3
d
16 bits
W
16 bits
X
16 bits
Y
16 bits
Z
Level-1
Page Table
Level-2
Page Table
Level-3
Page Table

Address-Translation Scheme



A logical address (32)

p1 (10)	p2 (10)	d (12)
00 0000 0010	01 0011 1100	0001 0000 0001



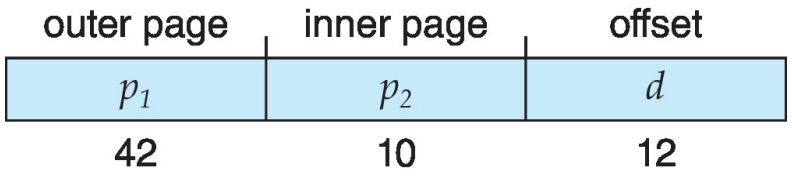
64-bit Logical Address Space

UNIT	ABBREVIATION	STORAGE
Bit	B	Binary Digit, Single 1 or 0
Nibble	-	4 bits
Byte/Octet	B	8 bits
Kilobyte	KB	1024 bytes
Megabyte	MB	1024 KB
Gigabyte	GB	1024 MB
Terabyte	TB	1024 GB
Petabyte	PB	1024 TB
Exabyte	EB	1024 PB
Zettabyte	ZB	1024 EB
Yottabyte	YB	1024 ZB

Storage units (www.byte-notes.com

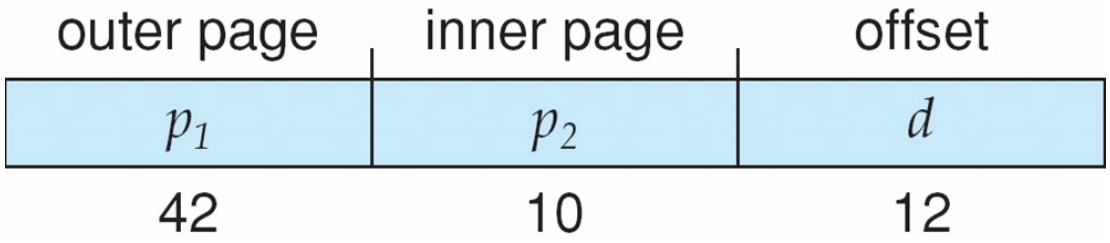
https://lecture481.rssing.com/chan-14834568/all_p1.html

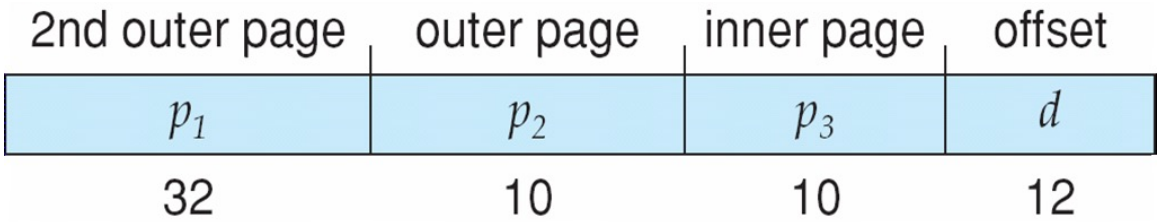
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries.
    - ($4,503,599,627,370,496$) = 4K T หรือ 4P entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes = 17,592,186,044,416 bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme





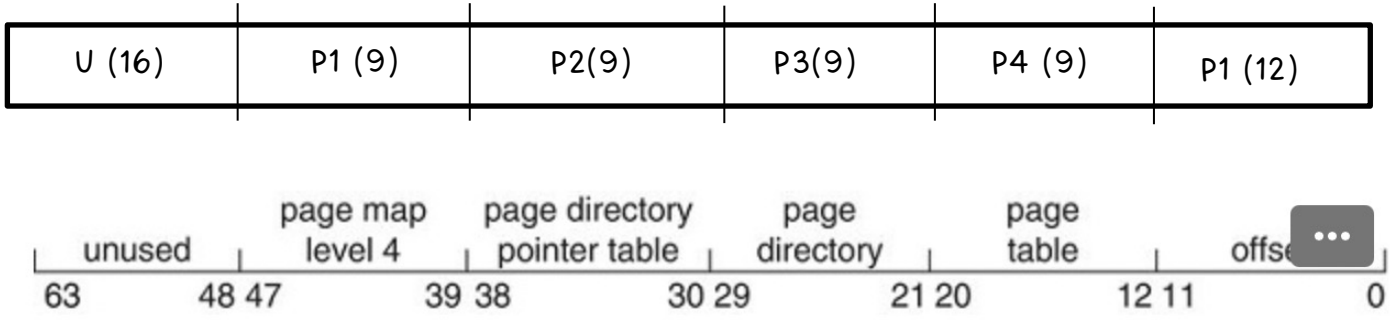
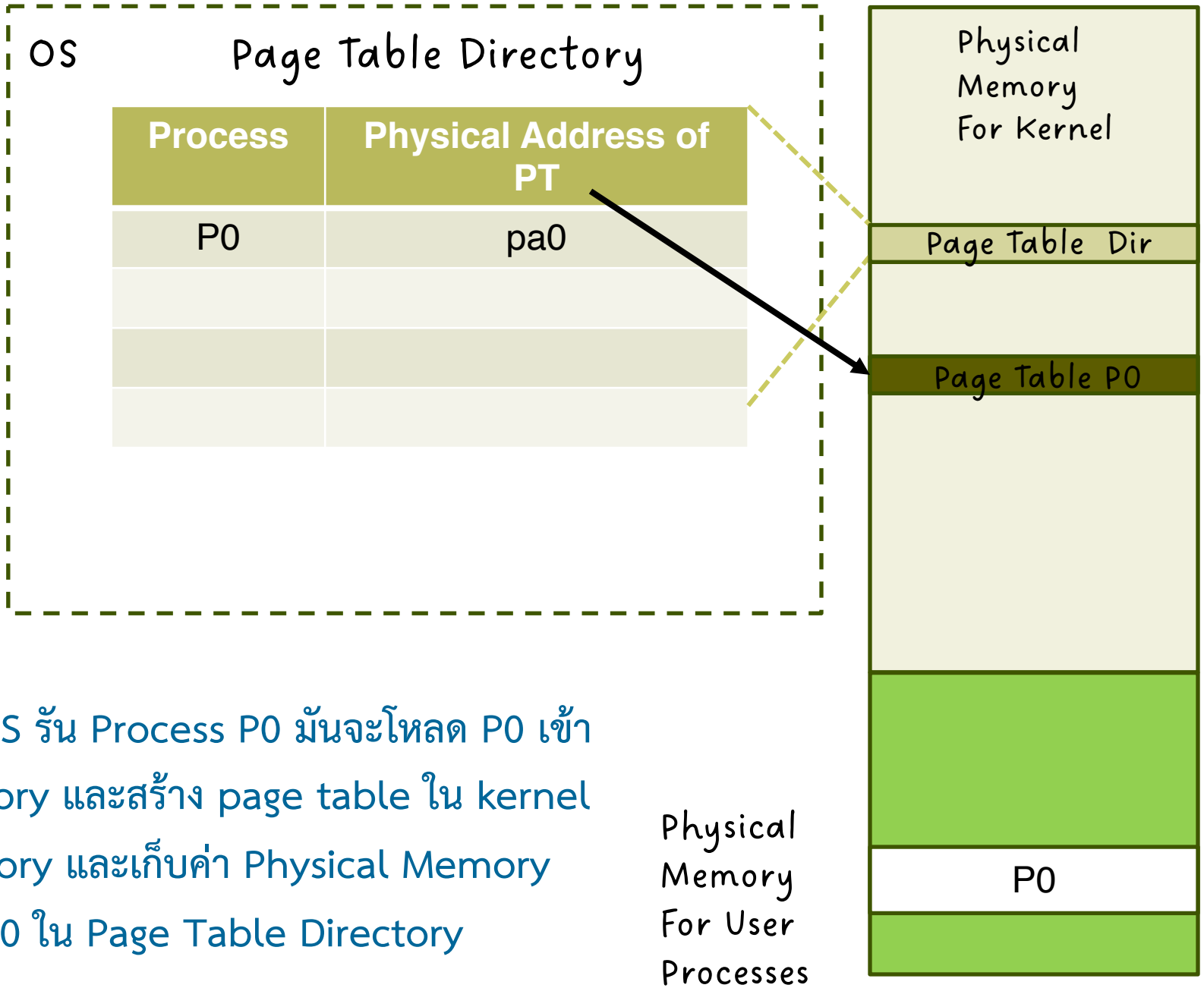
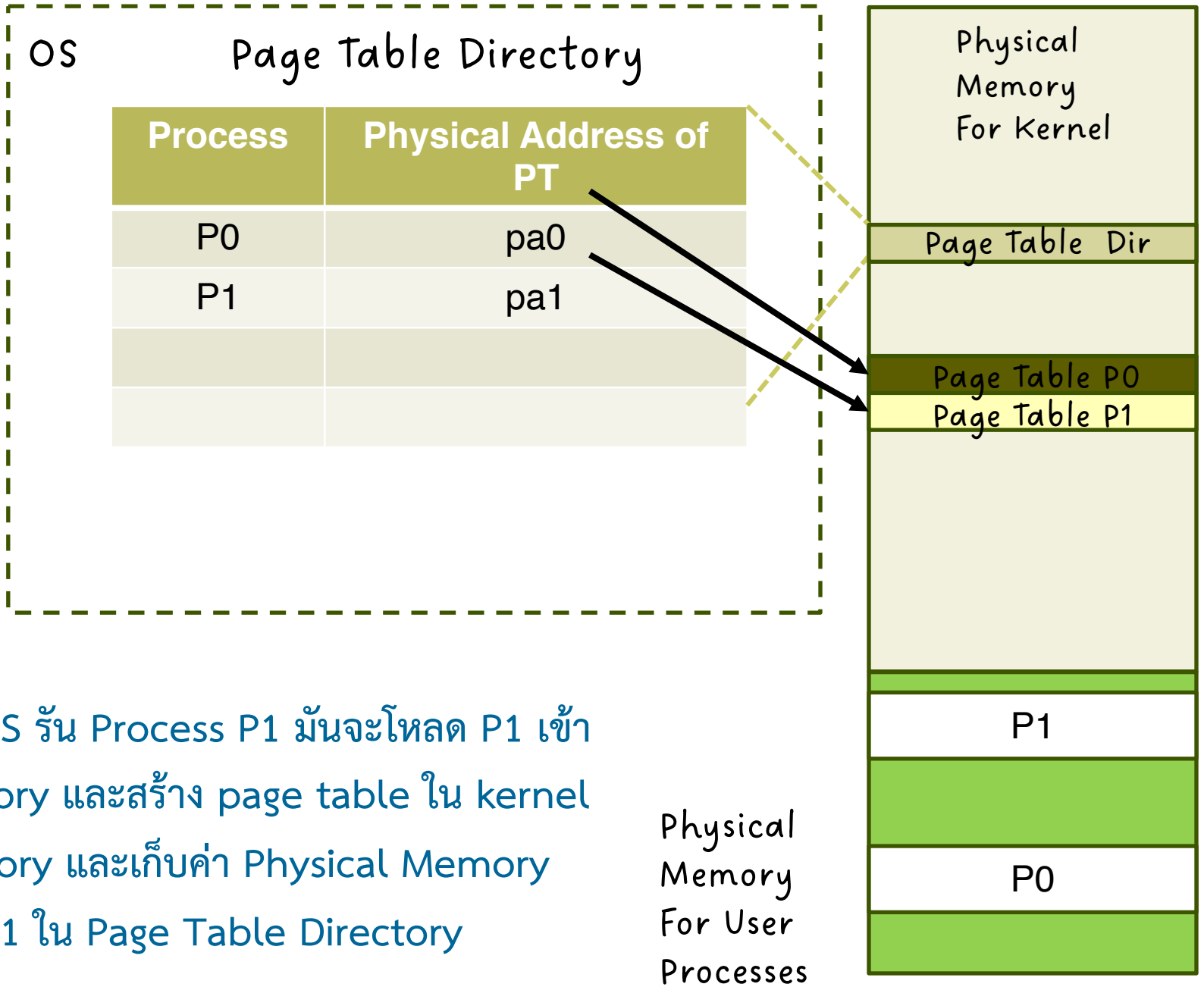


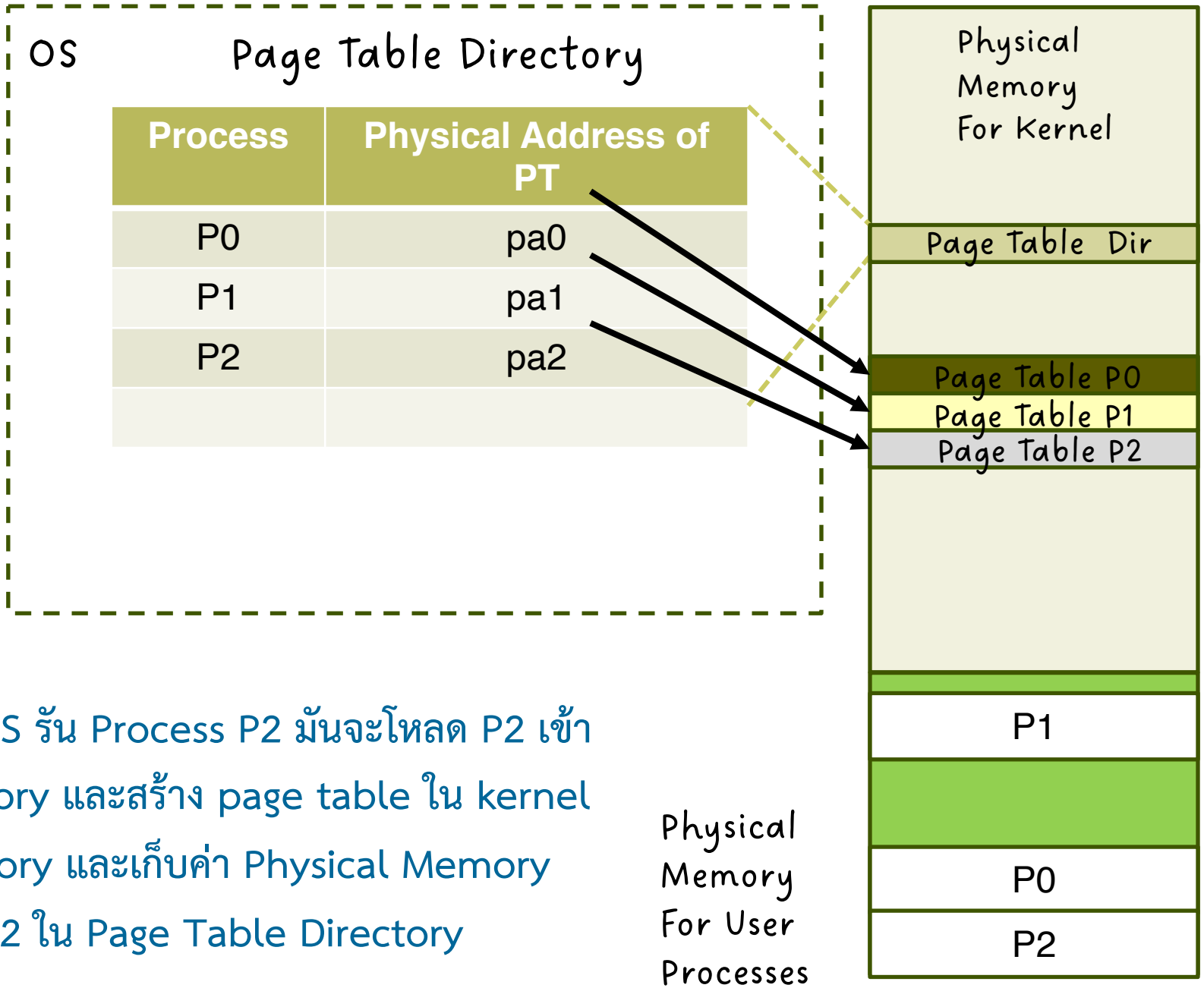
Figure 9.25 x86-64 linear address.



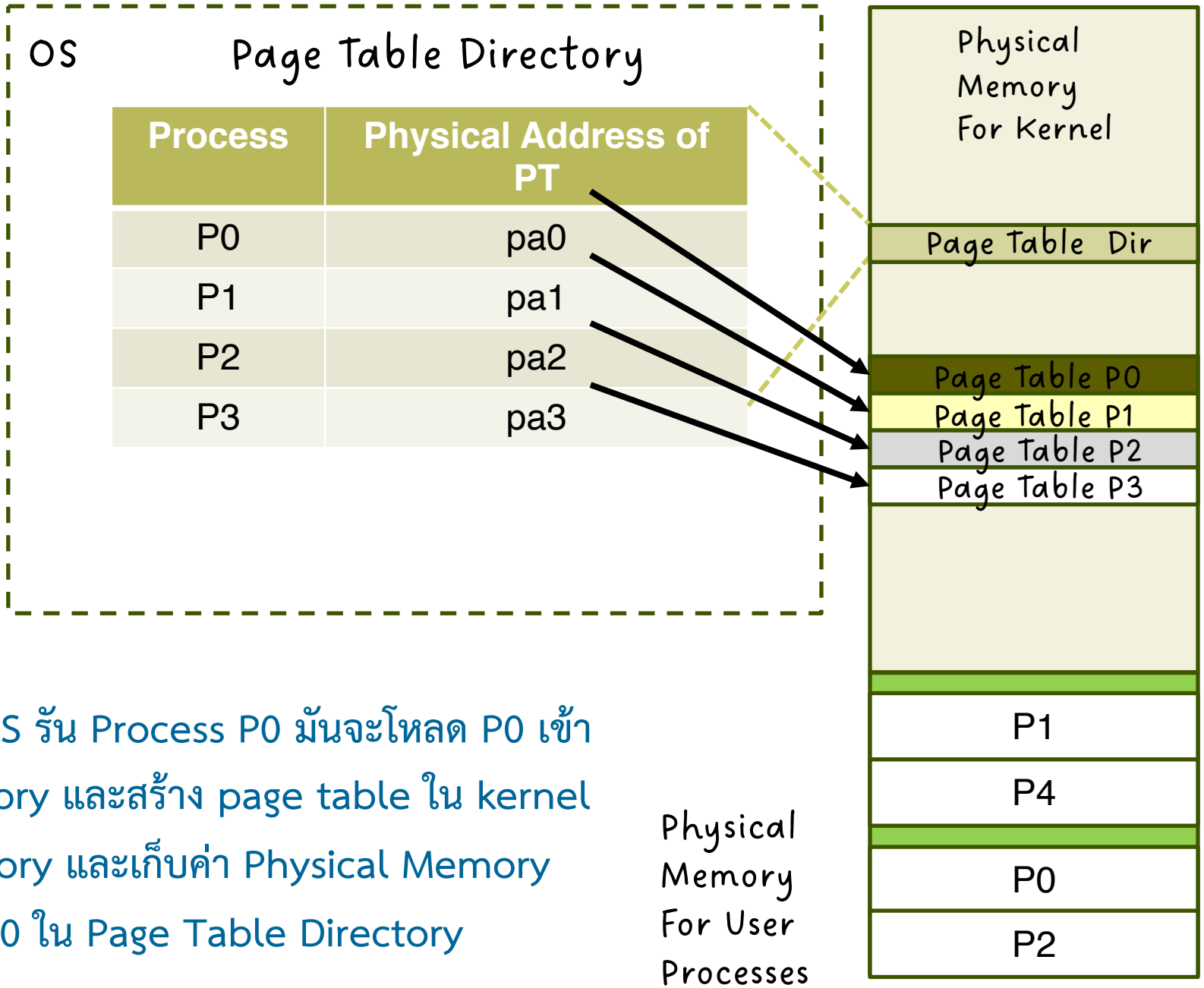
เมื่อ OS รัน Process P0 มันจะโหลด P0 เข้า Memory และสร้าง page table ใน kernel memory และเก็บค่า Physical Memory ของ P0 ใน Page Table Directory



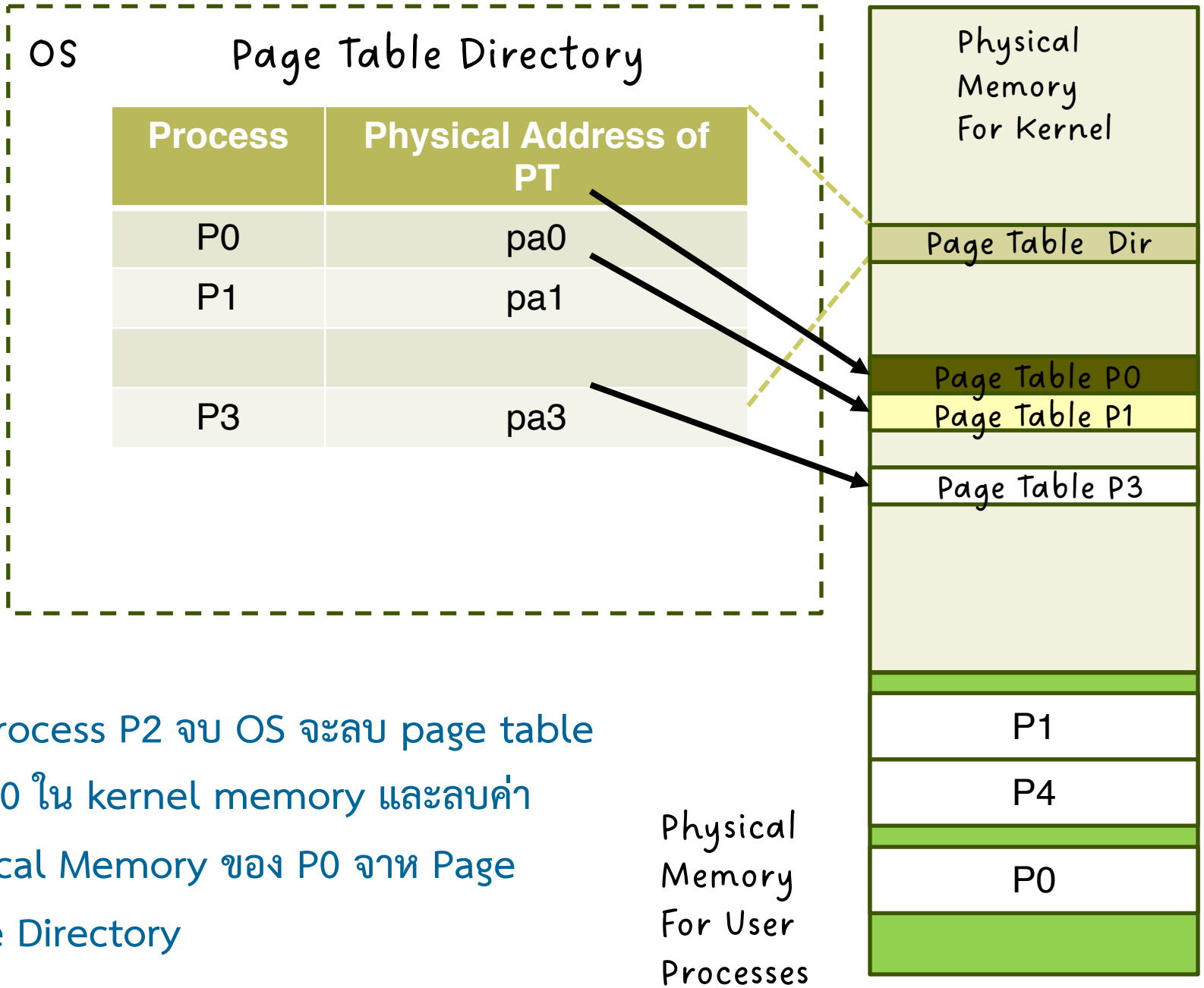
เมื่อ OS รัน Process P1 มันจะโหลด P1 เข้า Memory และสร้าง page table ใน kernel memory และเก็บค่า Physical Memory ของ P1 ใน Page Table Directory



เมื่อ OS รัน Process P2 มันจะโหลด P2 เข้า Memory และสร้าง page table ใน kernel memory และเก็บค่า Physical Memory ของ P2 ใน Page Table Directory

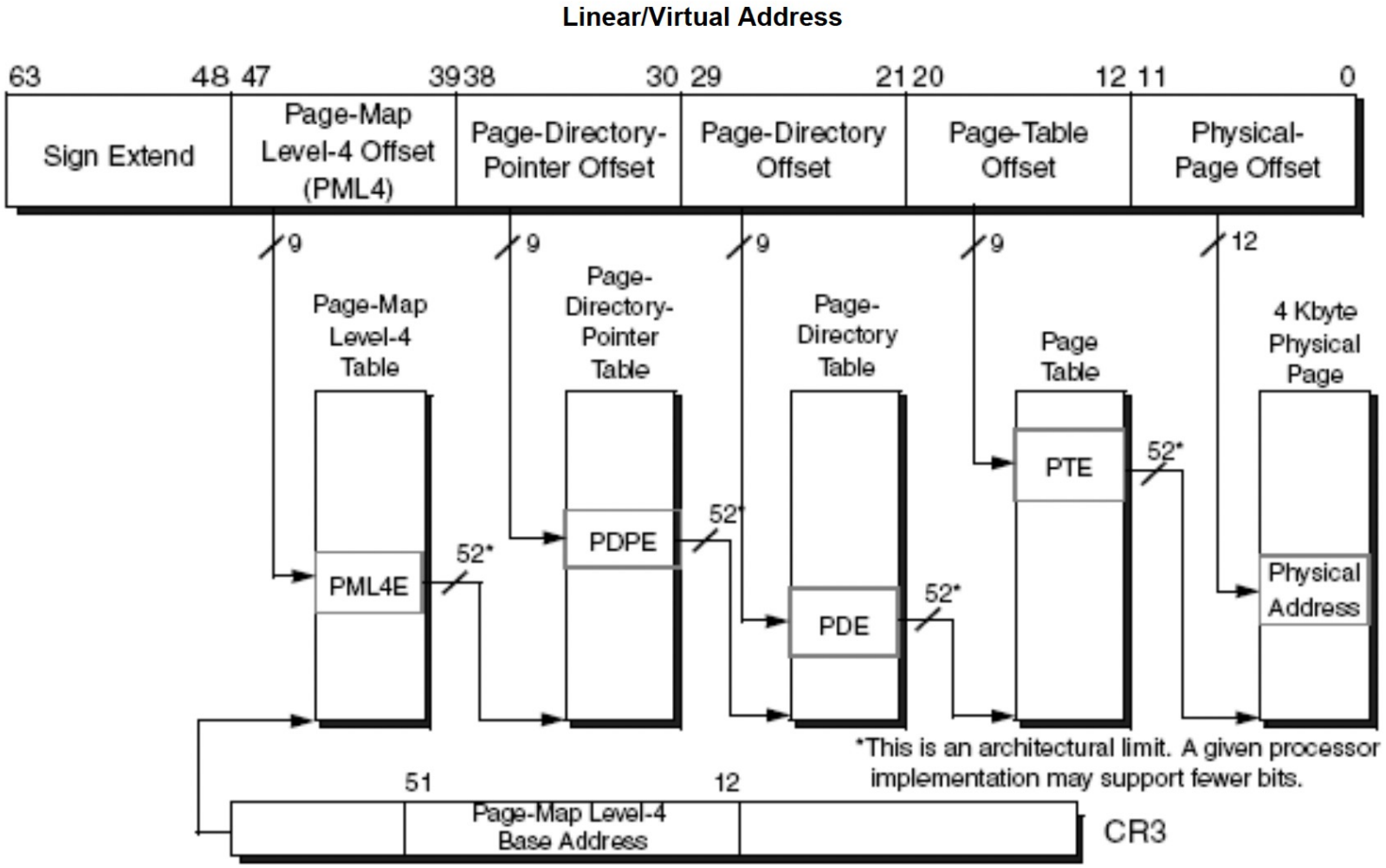


เมื่อ OS รัน Process P0 มันจะโหลด P0 เข้า Memory และสร้าง page table ใน kernel memory และเก็บค่า Physical Memory ของ P0 ใน Page Table Directory



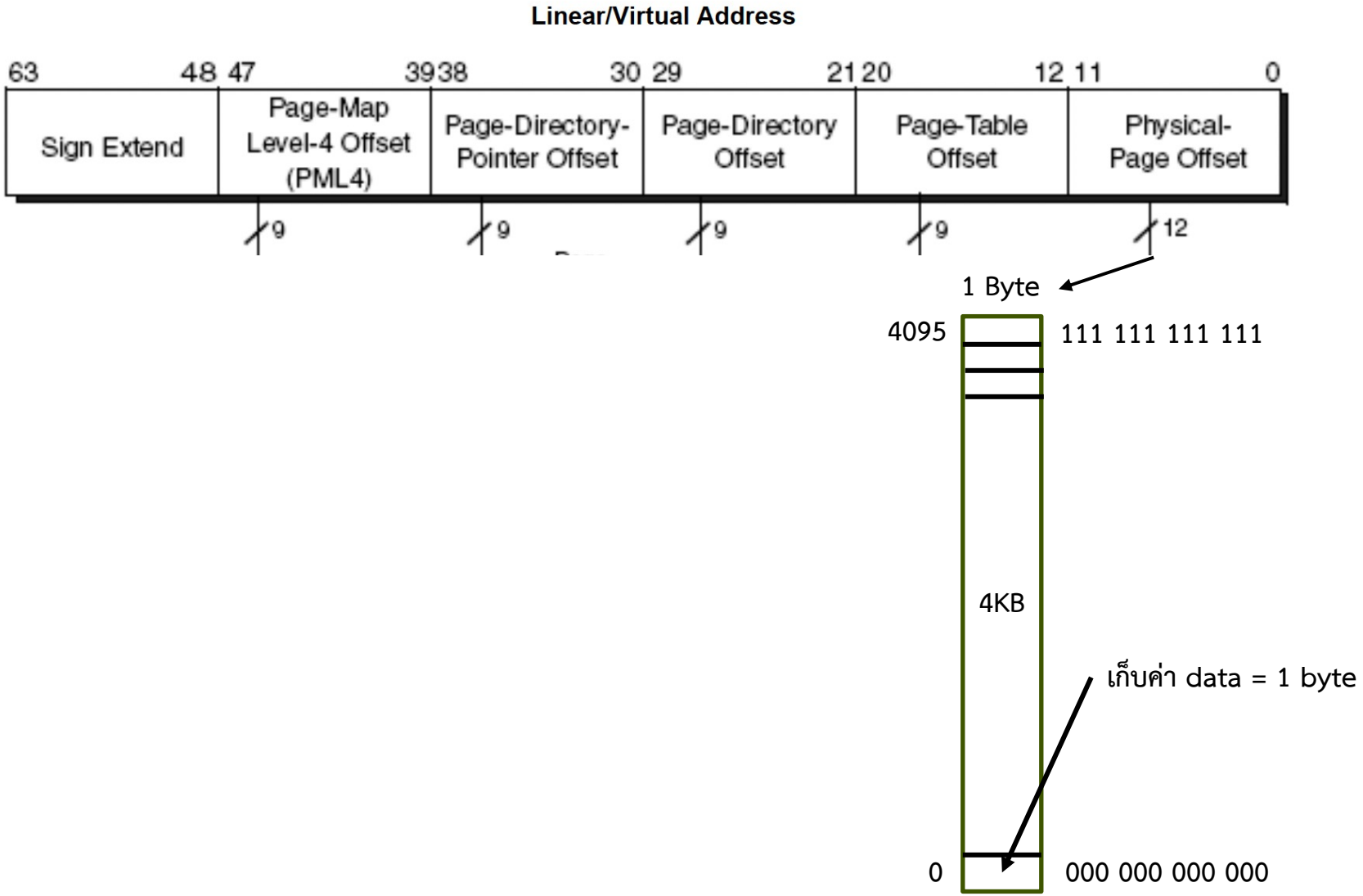
เมื่อ Process P2 จบ OS จะลบ page table ของ P0 ใน kernel memory และลบค่า Physical Memory ของ P0 จาห Page Table Directory

Hierarchical Page table



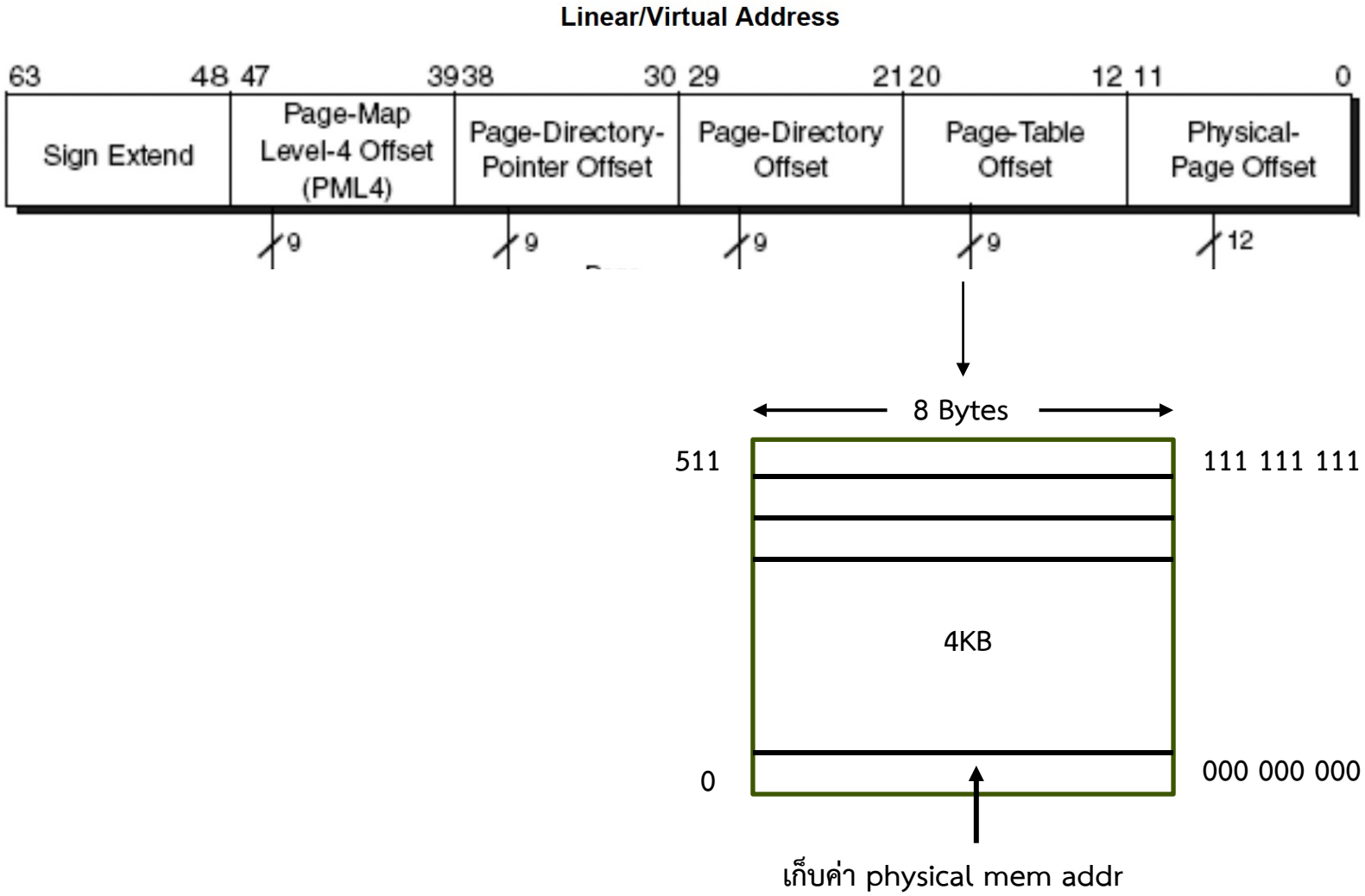
อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table



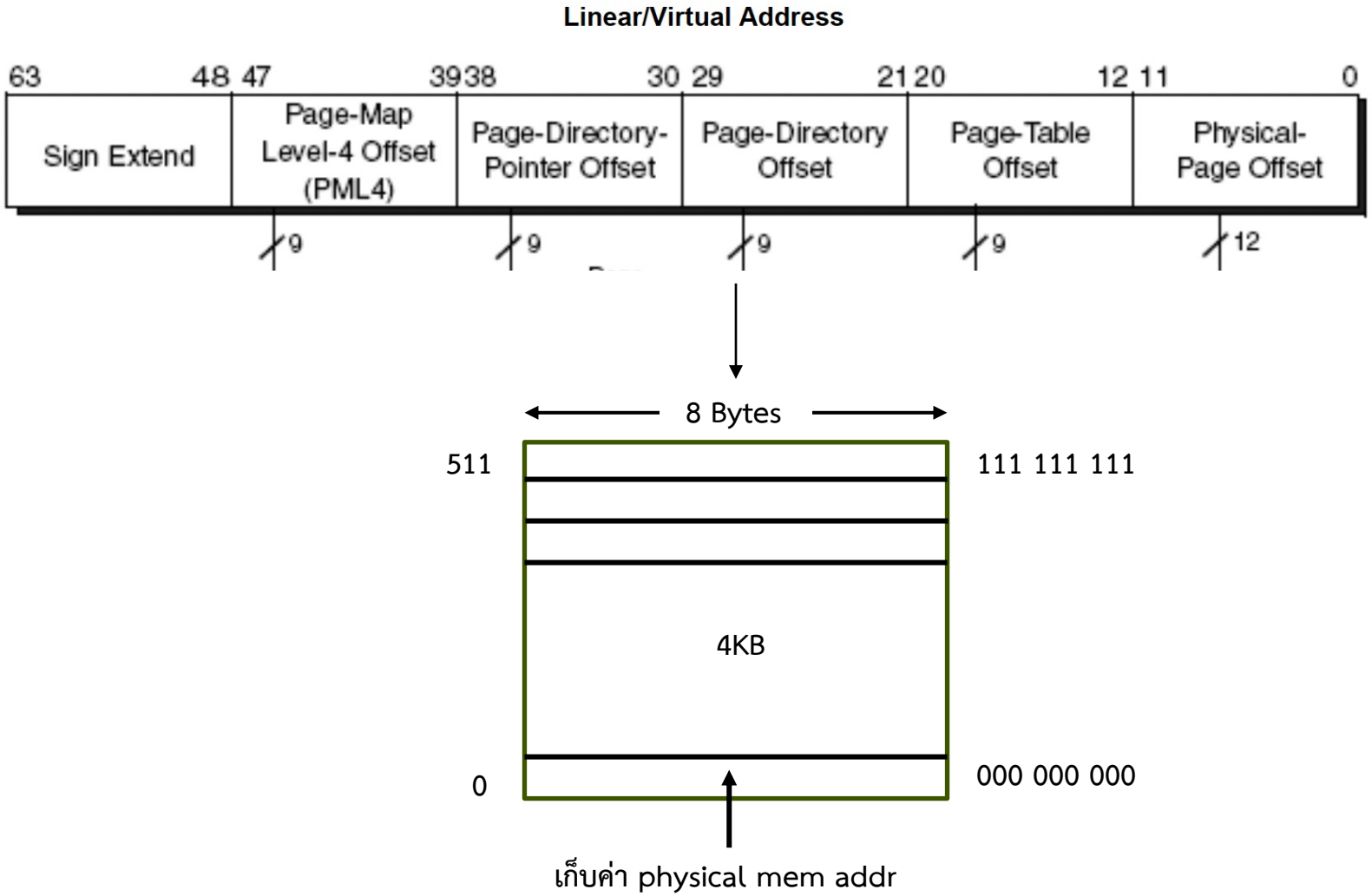
อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table



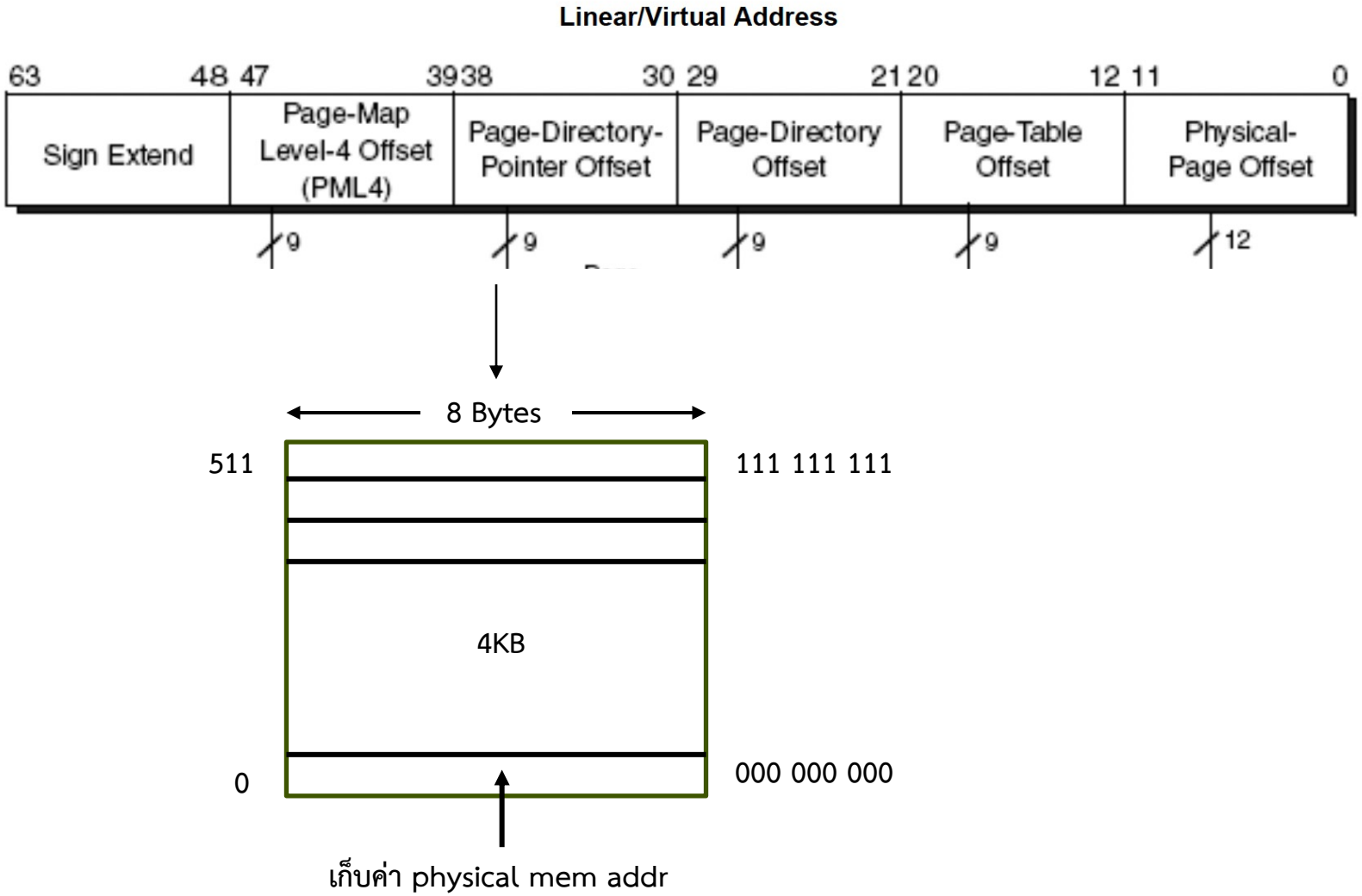
อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table



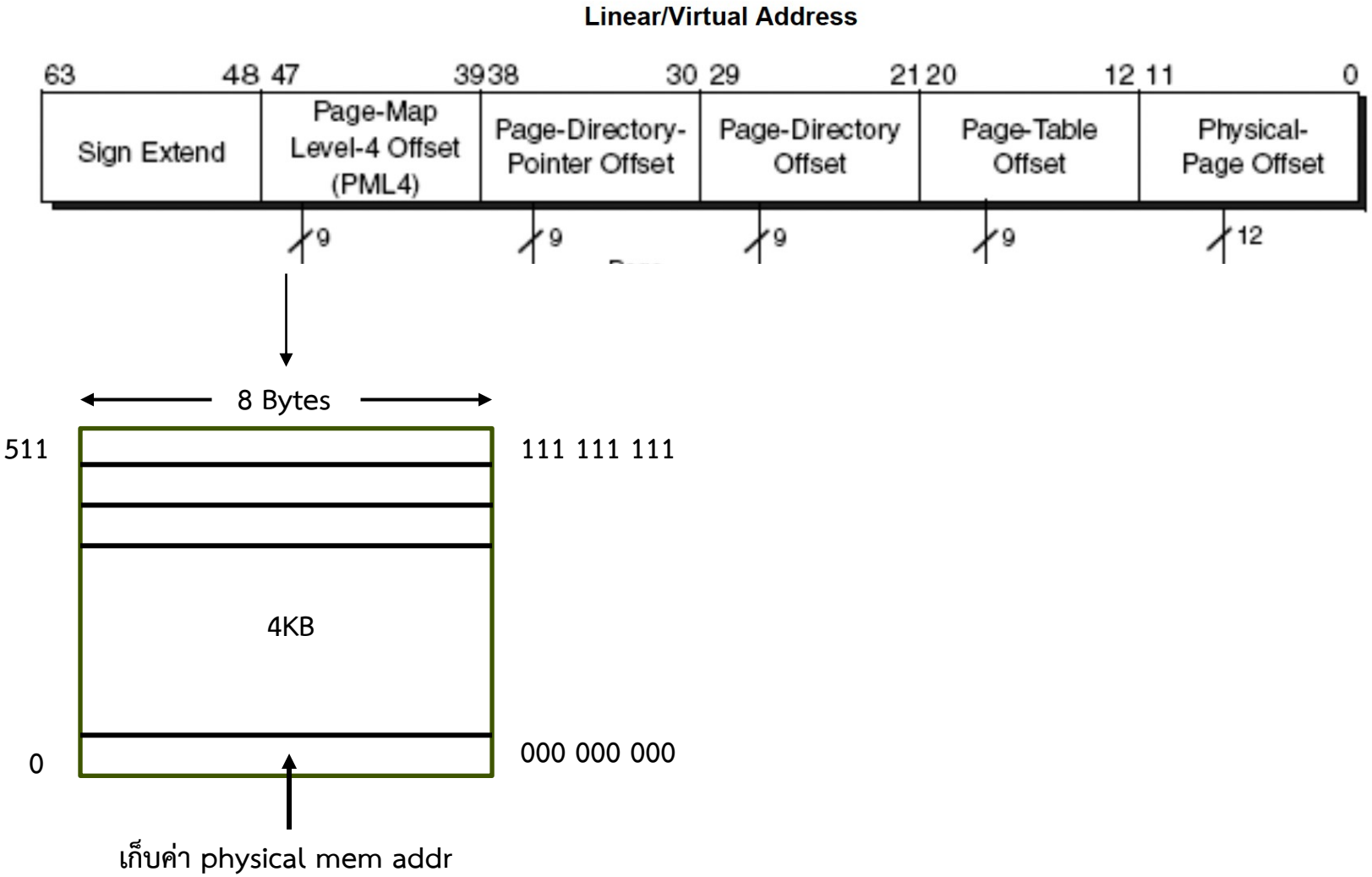
อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table



อ้างอิง AMD, AMD-V Nested Paging, 2008

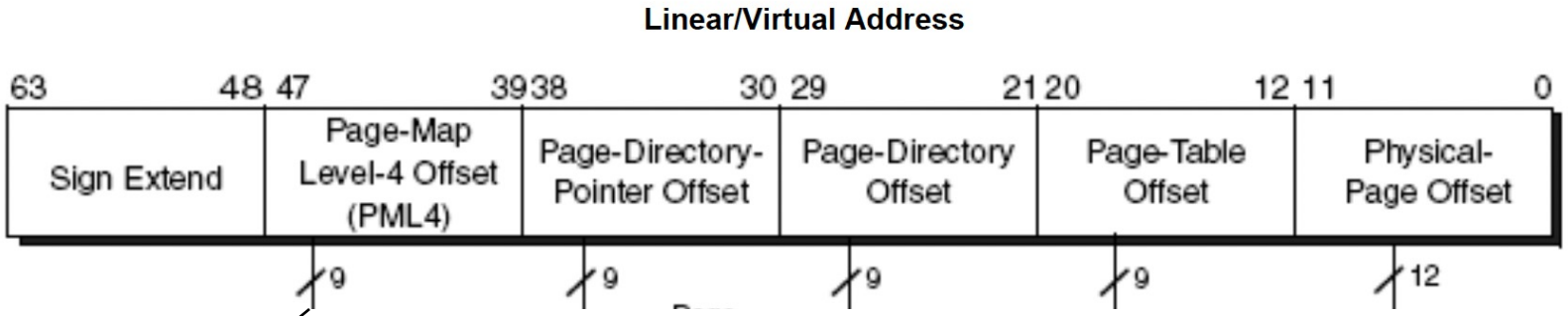
Hierarchical Page table

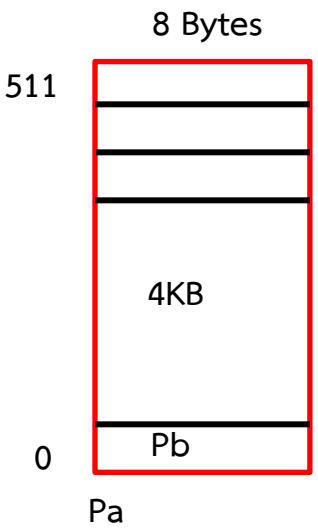


อ้างอิง AMD, AMD-V Nested Paging, 2008

VA = 0x 10 = 0b 0001 0000

000_8 000_8 000_8 000_8 0020_8



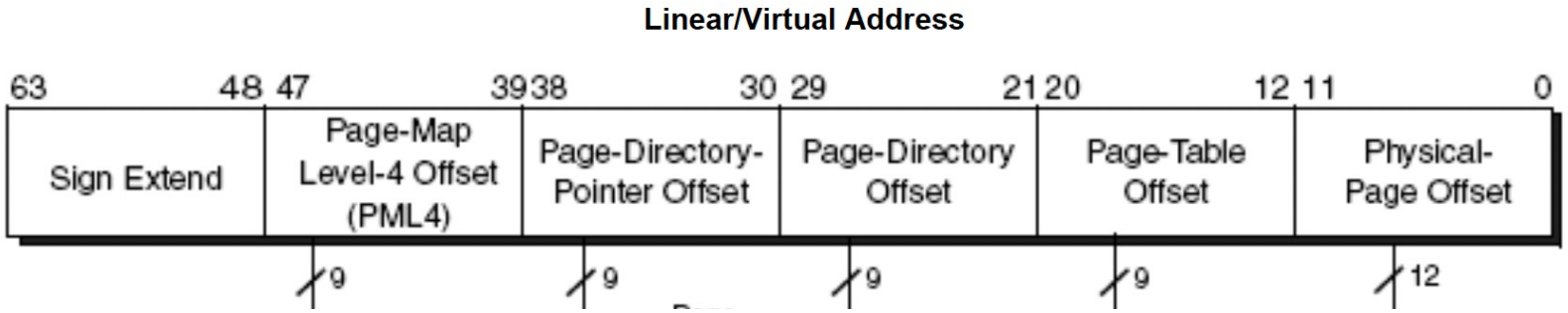


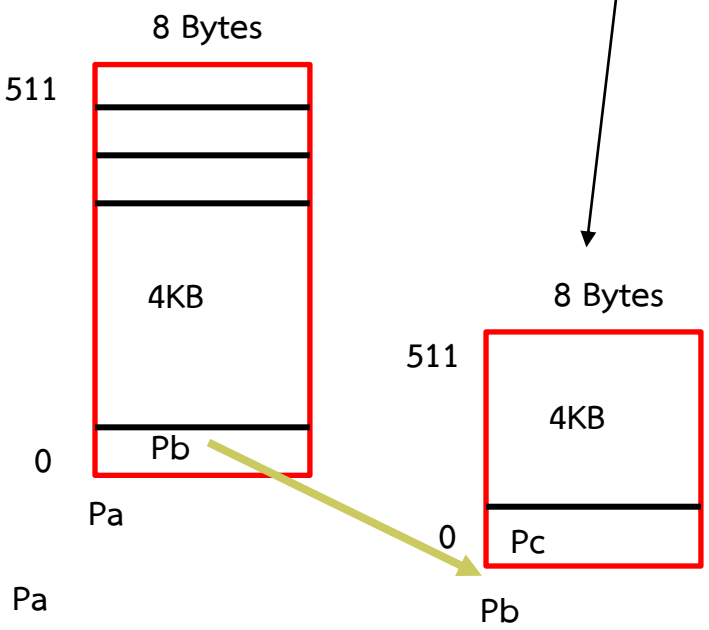
CR3 = Pa

อ้างอิง AMD, AMD-V Nested Paging, 2008

VA = 0x 10 = 0b 0001 0000

000_8 000_8 000_8 000_8 0020_8



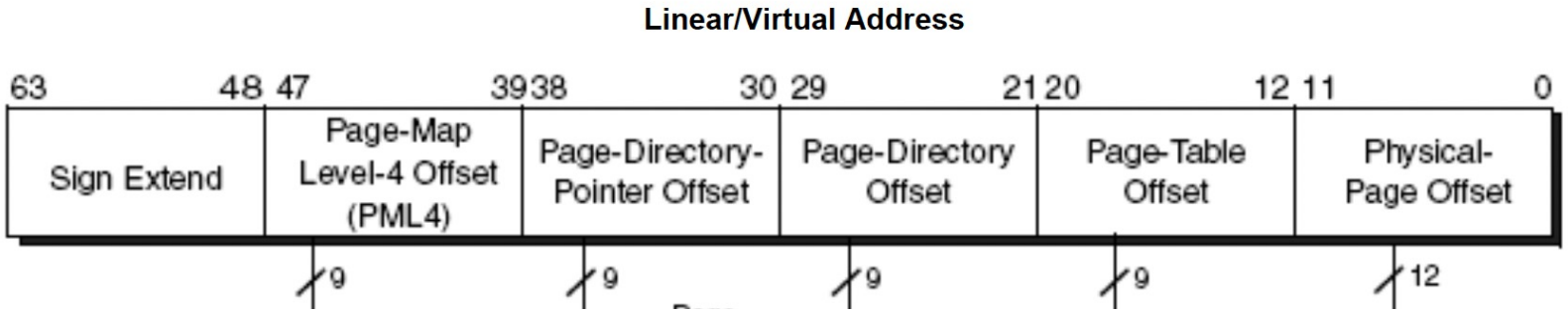


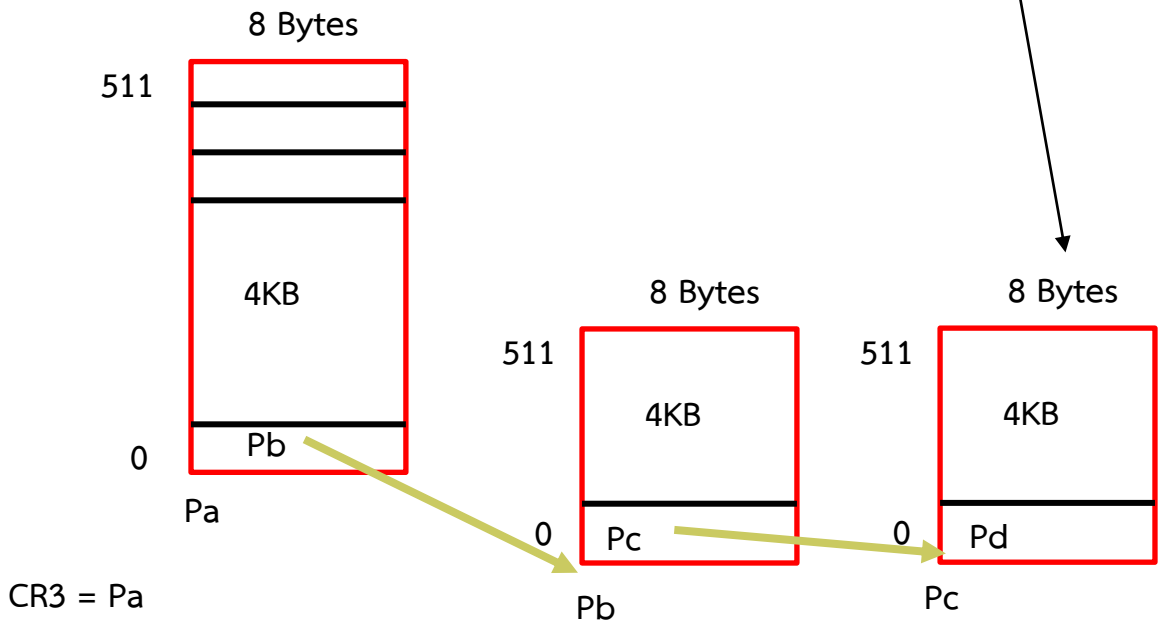
CR3 = Pa

อ้างอิง AMD, AMD-V Nested Paging, 2008

VA = 0x 10 = 0b 0001 0000 = 0o 20

000_8 000_8 000_8 000_8 0020_8

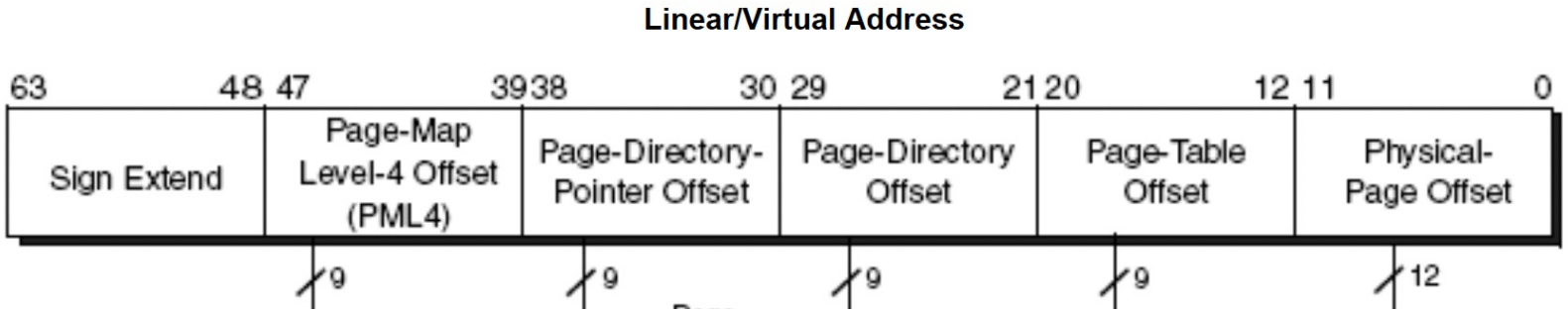


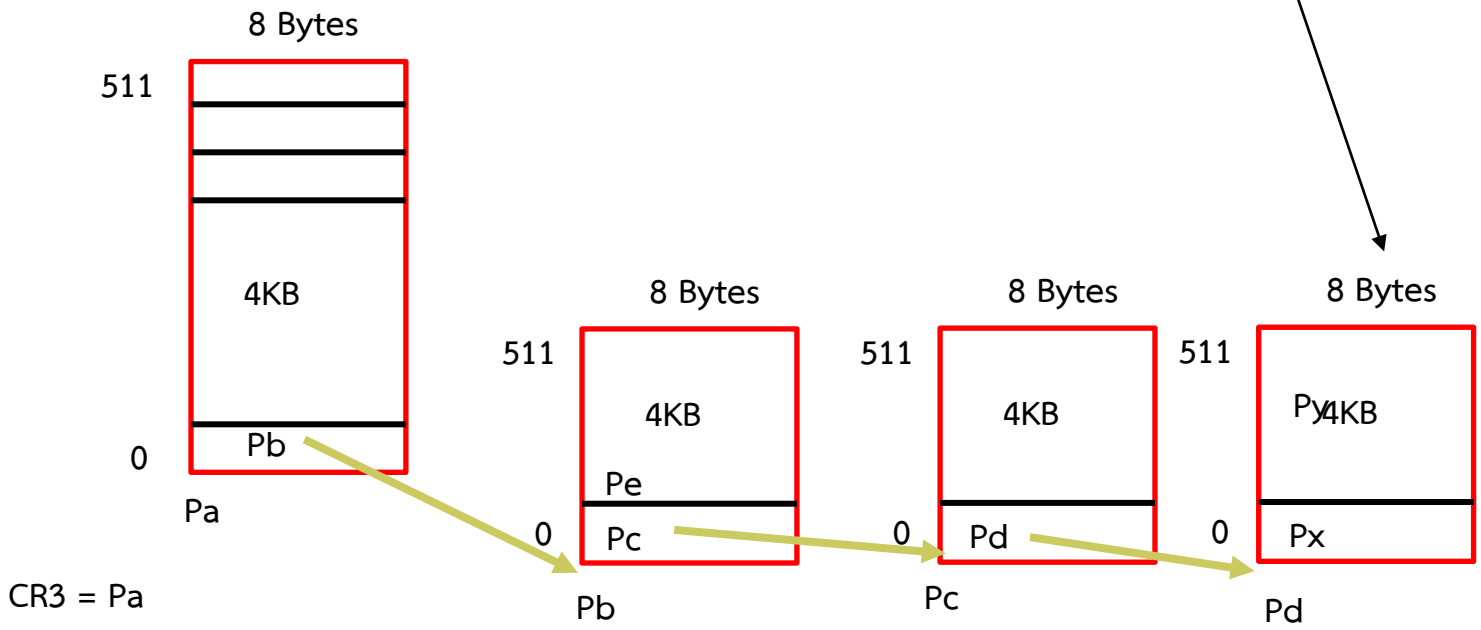


อ้างอิง AMD, AMD-V Nested Paging, 2008

VA = 0x 10 = 0b 0001 0000 = 0o 20

000_8 000_8 000_8 000_8 0020_8

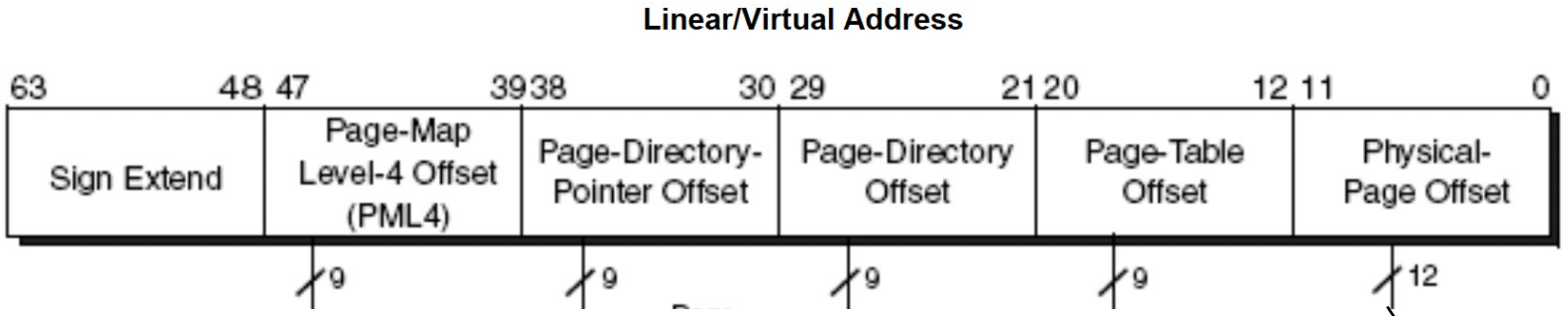


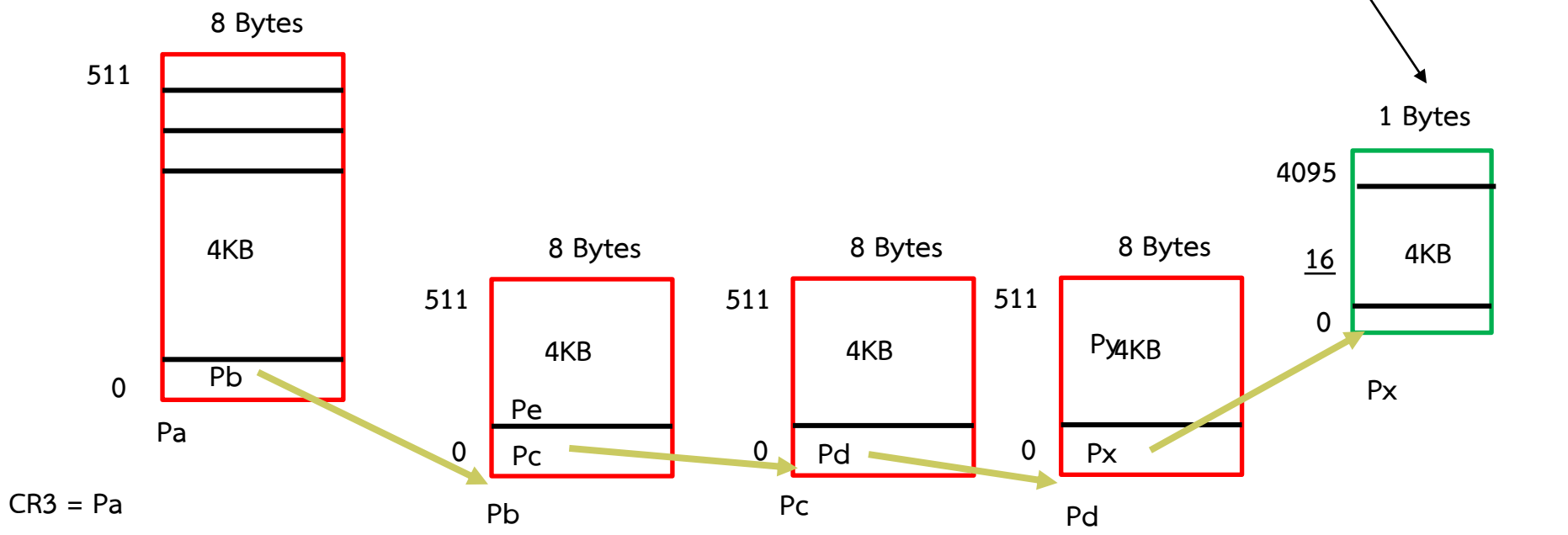


อ้างอิง AMD, AMD-V Nested Paging, 2008

VA = 0x 10 = 0b 0001 0000 = 0o 20

000_8 000_8 000_8 000_8 0020_8

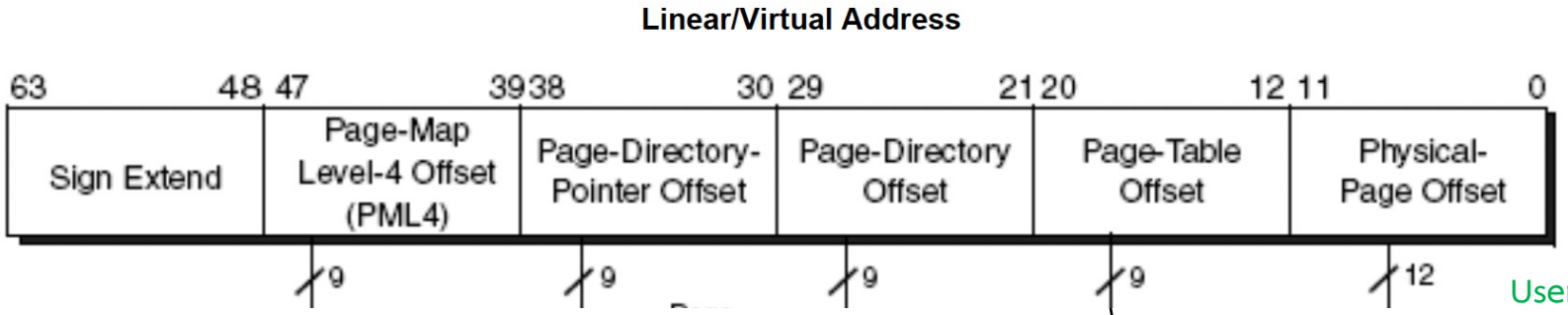


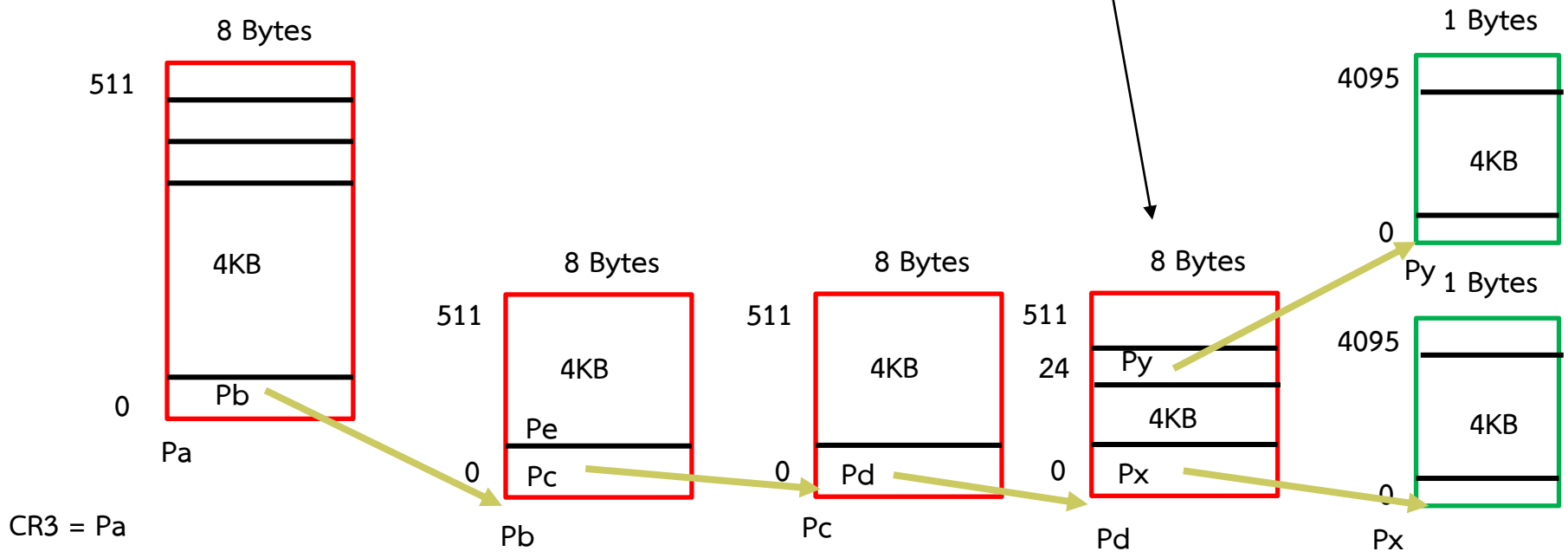


อ้างอิง AMD, AMD-V Nested Paging, 2008

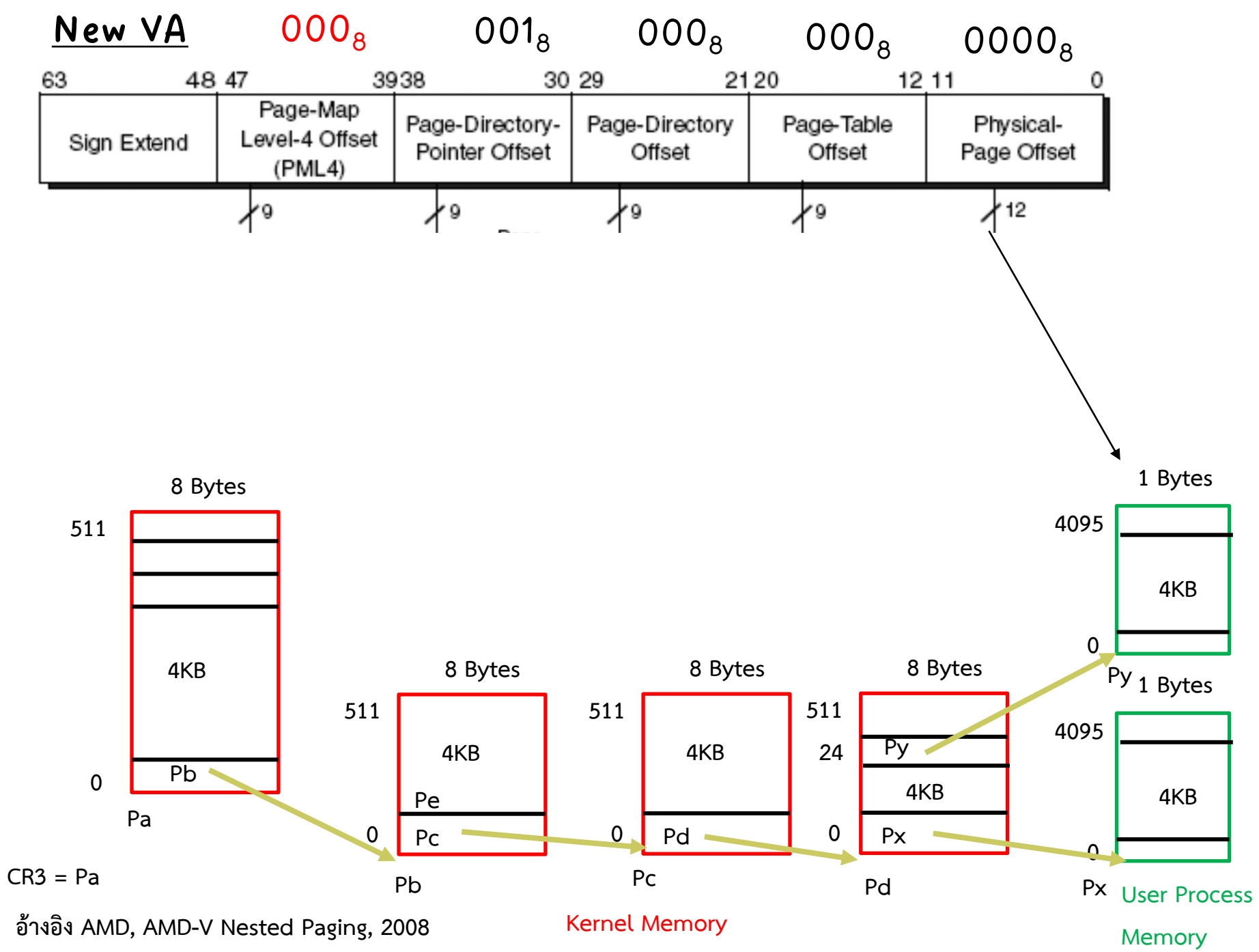
VA = 0x 3000 = 0b 0011 0000 0000 0000 = 0o 30000

000_8 000_8 000_8 003_8 0000_8

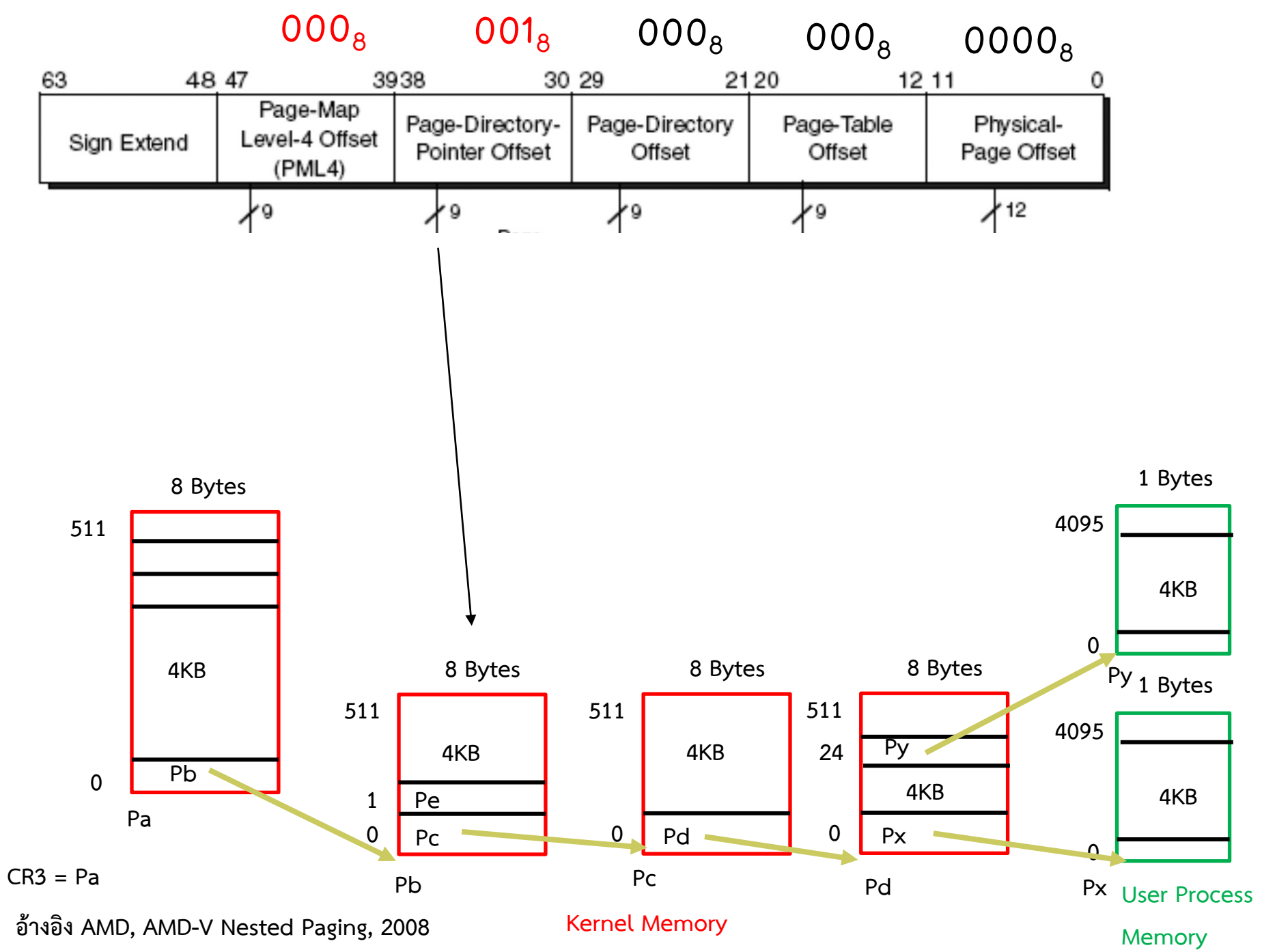




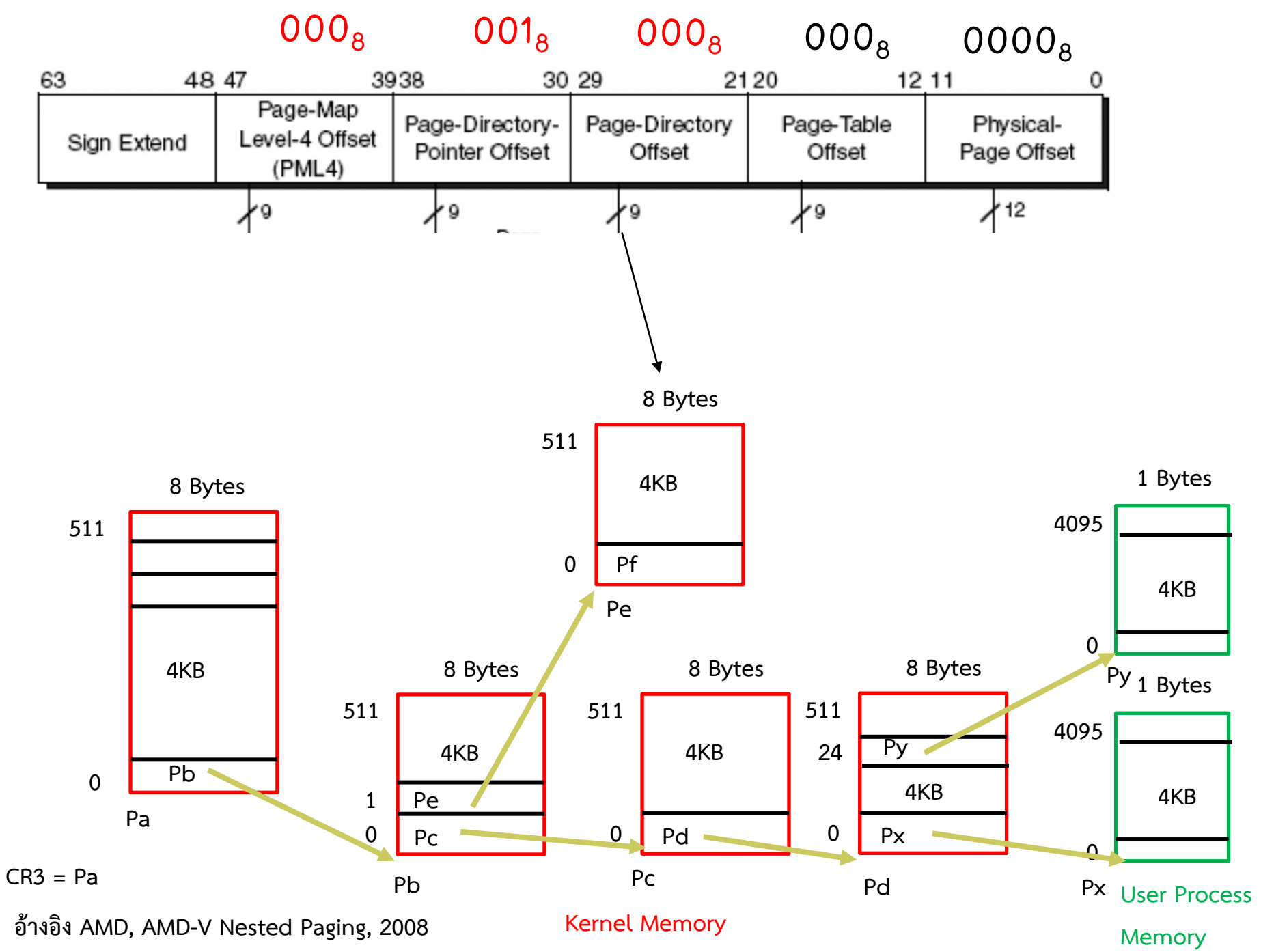
อ้างอิง AMD, AMD-V Nested Paging, 2008



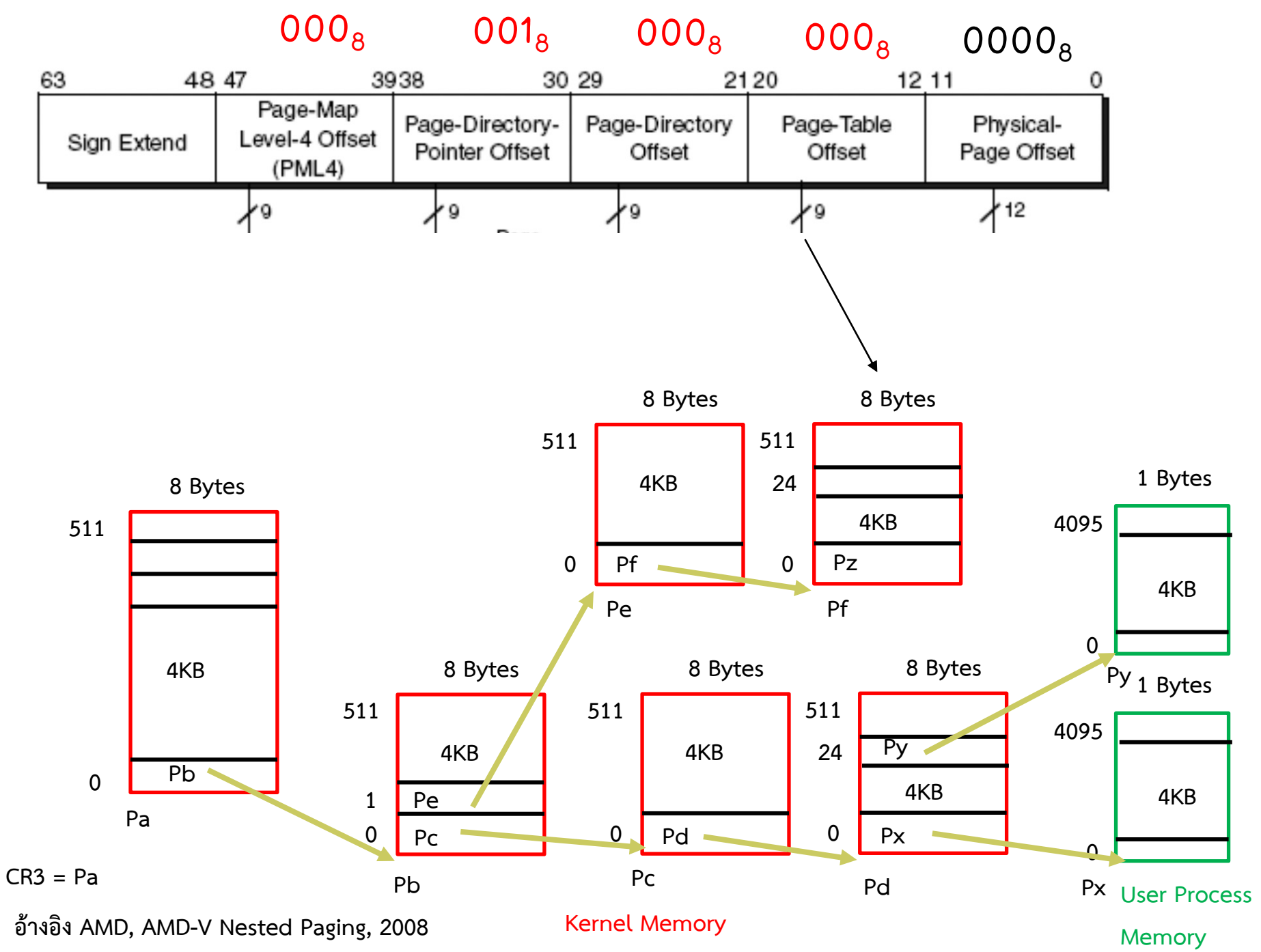
อ้างอิง AMD, AMD-V Nested Paging, 2008



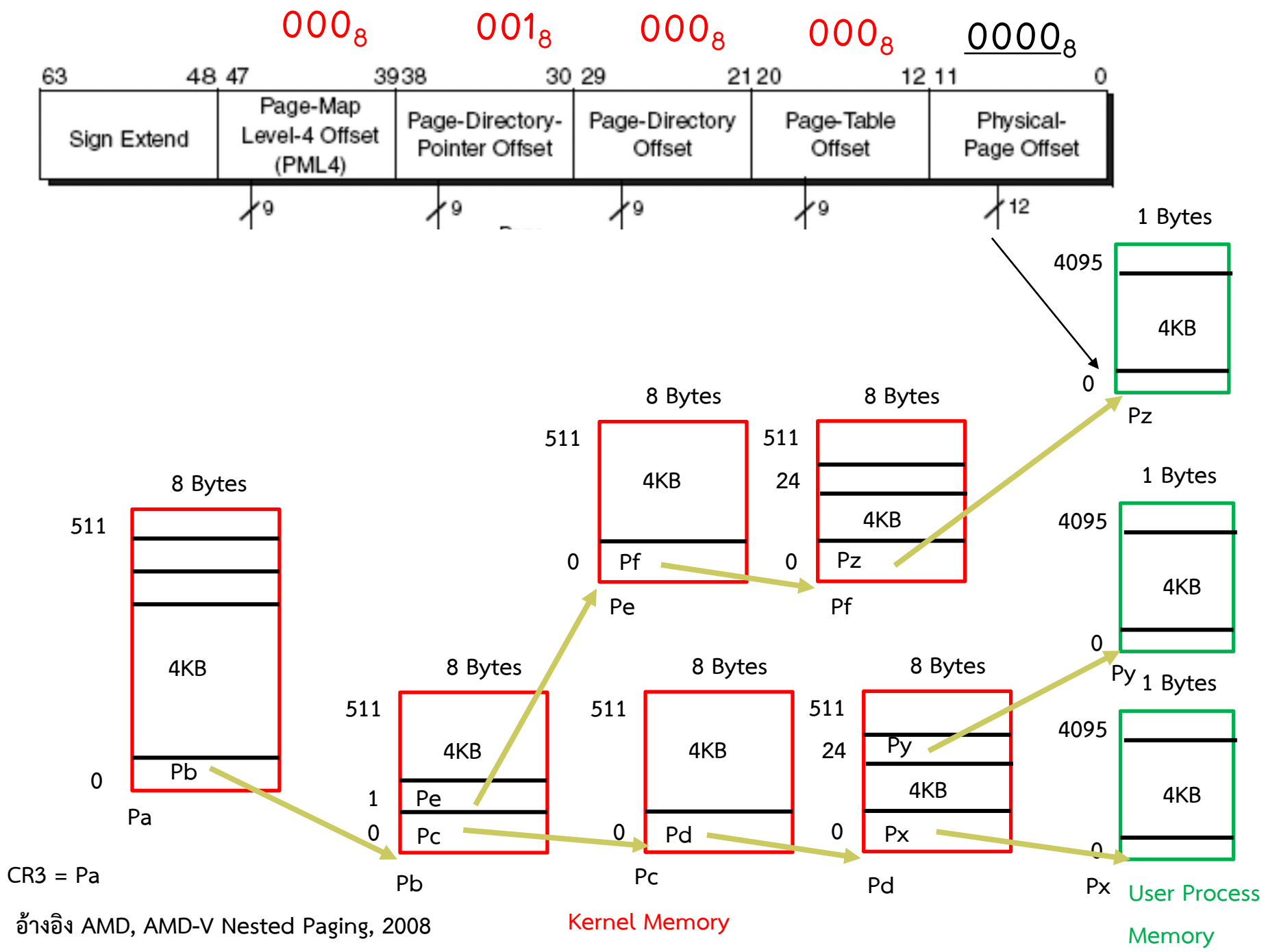
อ้างอิง AMD, AMD-V Nested Paging, 2008



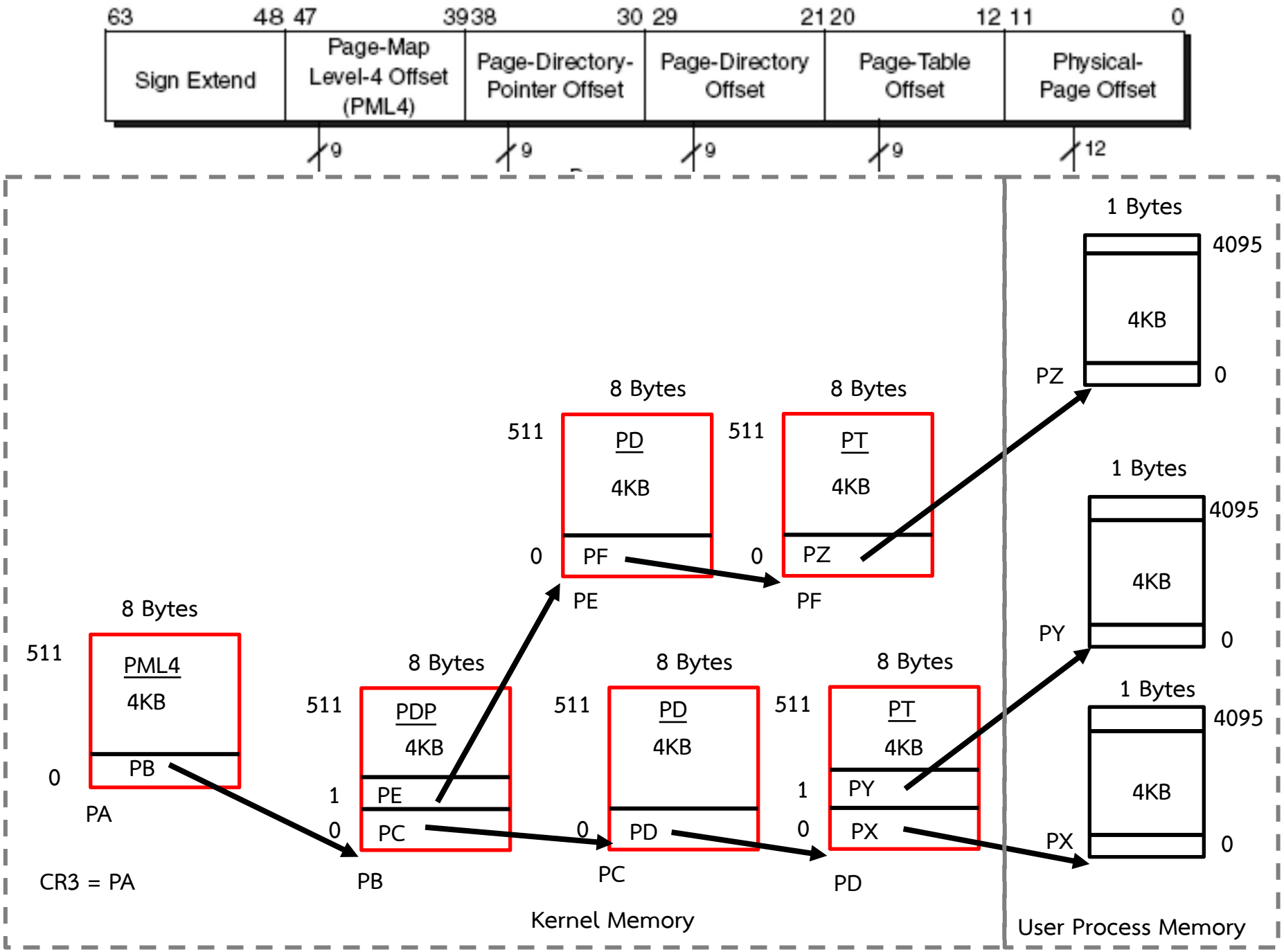
อ้างอิง AMD, AMD-V Nested Paging, 2008

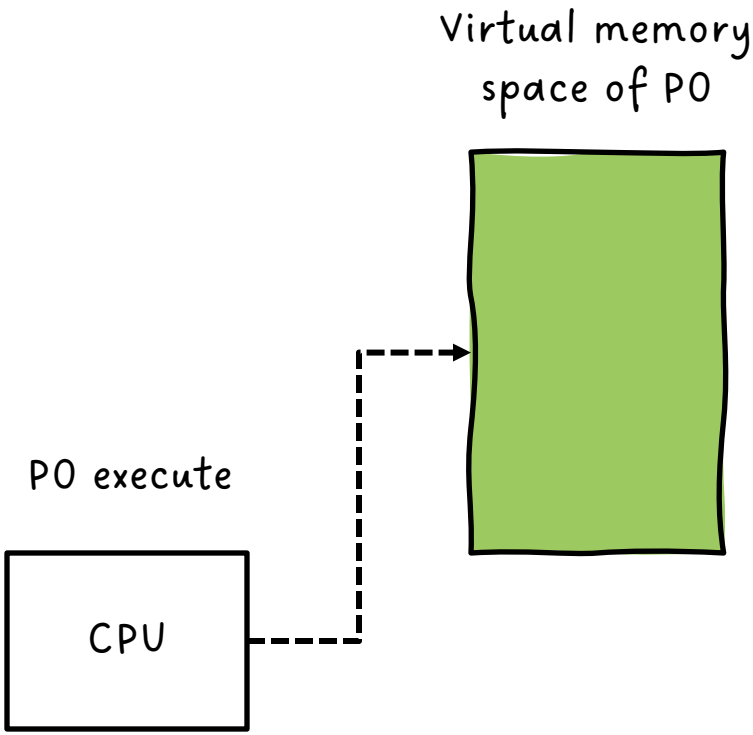


อ้างอิง AMD, AMD-V Nested Paging, 2008

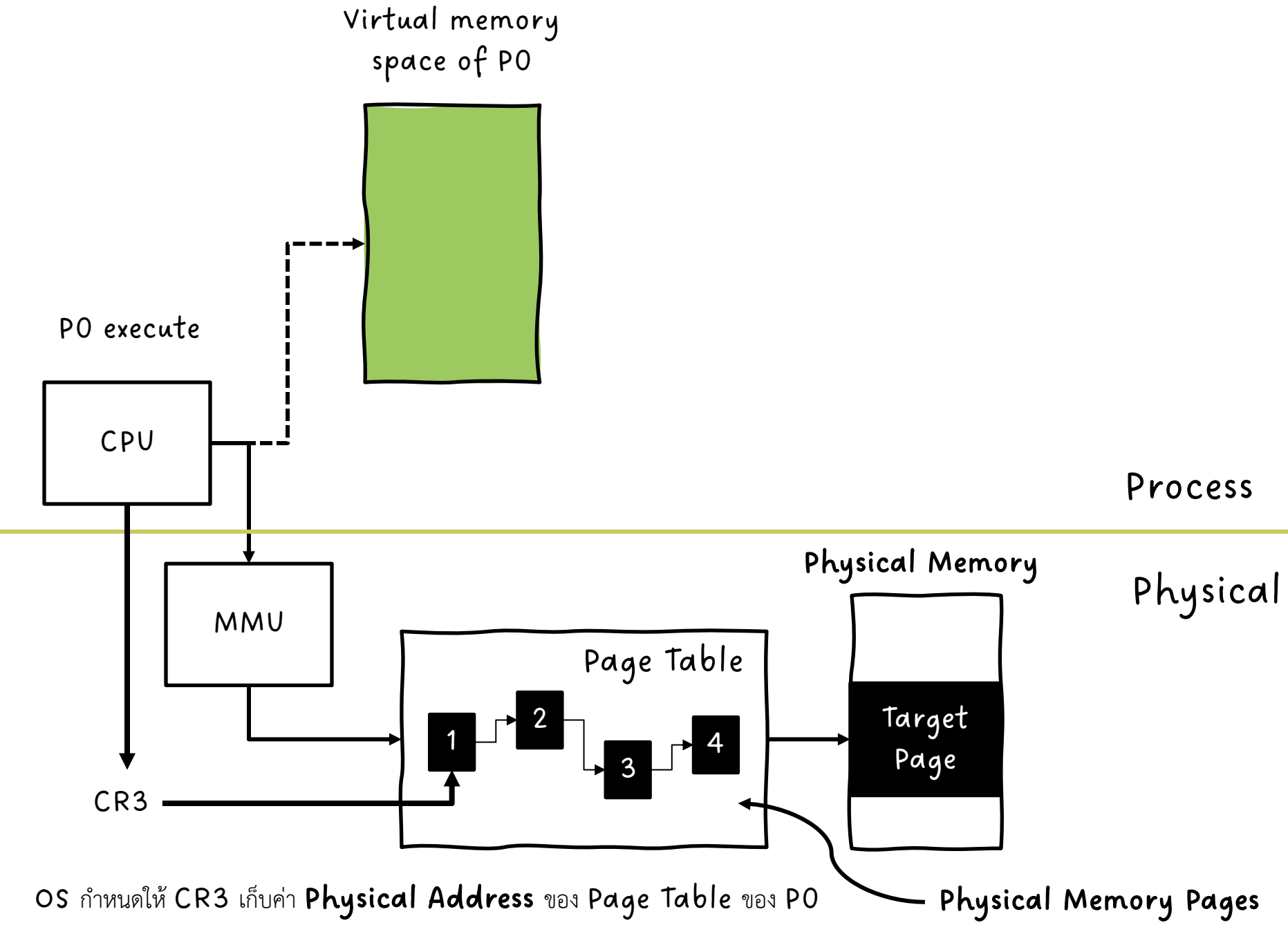


อ้างอิง AMD, AMD-V Nested Paging, 2008


63
48 47
39 38
30 29
21 20
12 11
0
Sign Extend
Page-Map Level-4 Offset (PML4)
Page-Directory-Pointer Offset
Page-Directory Offset
Page-Table Offset
Physical-Page Offset
9
9
9
9
12
8 Bytes
511
PML4
4KB
0
PB
PA
CR3 = PA
8 Bytes
511
PDP
4KB
1
PE
0
PC
PB
8 Bytes
511
PD
4KB
0
PF
PE
8 Bytes
511
PD
4KB
0
PD
PC
8 Bytes
511
PT
4KB
1
PY
0
PX
PD
8 Bytes
511
PT
4KB
0
PZ
PF
Kernel Memory
1 Bytes
4095
4KB
PZ
0
1 Bytes
4095
4KB
PY
0
1 Bytes
4095
4KB
PX
0
User Process Memory



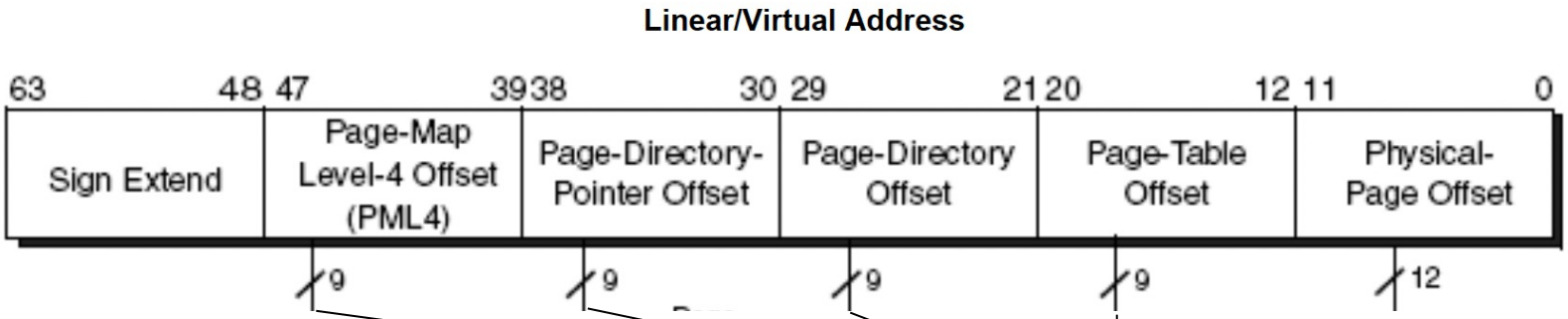
Process


Virtual memory
space of P0
P0 execute
CPU
Process
Physical
Physical Memory
MMU
Page Table
1
2
3
4
Target
Page
CR3
OS กำหนดให้ CR3 เก็บค่า Physical Address ของ Page Table ของ P0
Physical Memory Pages

ปัญหาและวิธีแก้ปัญหาของ
Heirarchical Page Table

- MMU ต้องเข้าถึง Memory 4 ครั้งเพื่อหาค่า Physical Mem Address (PA) ของ Virtual Mem Address (VA)
- ต้องใช้ TLB ช่วยเก็บค่า VA to PA mapping ทำให้ Access Time น้อยลง
- TLB มีหลายชั้น เหมือน Cache มี L1 L2 L3 L4
- ในระหว่างที่ Walk Page Table ข้อมูลของ Page Table ระดับ 1 ถึง 4 ที่ถูกใช้บ่อย จะถูกเก็บไว้ใน Cache level ต่างๆของ CPU ทำให้เมื่อเกิด TLB Miss การ Walk Page Table จะทำได้เร็วขึ้น เพราะข้อมูลของ Page Table บางส่วนอยู่ใน Cache แล้ว

Hierarchical Page table

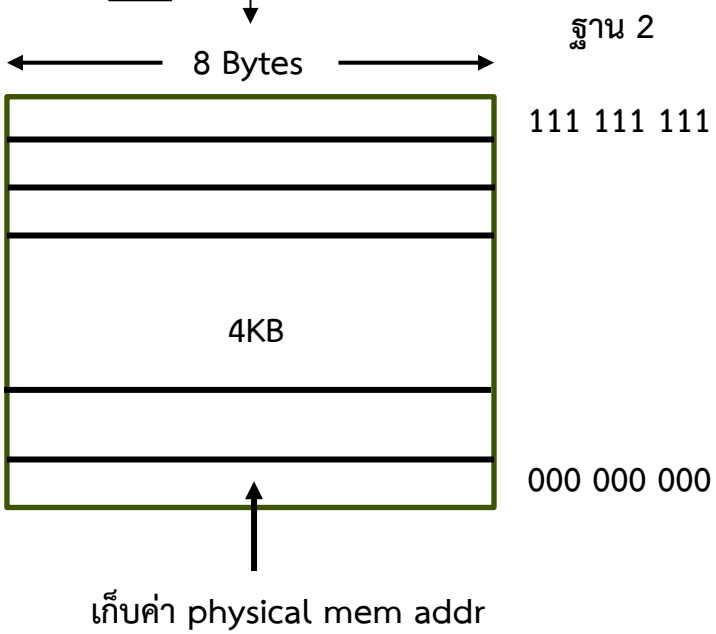


$F(x) = index * 8 + Base$

$F(000_8) = 000_8 * 8 + Base$
$= 000_8 * 2^3 + Base$
$= (000_8 << 3) + Base$
$= 0000_8 + Base$
$= Base$

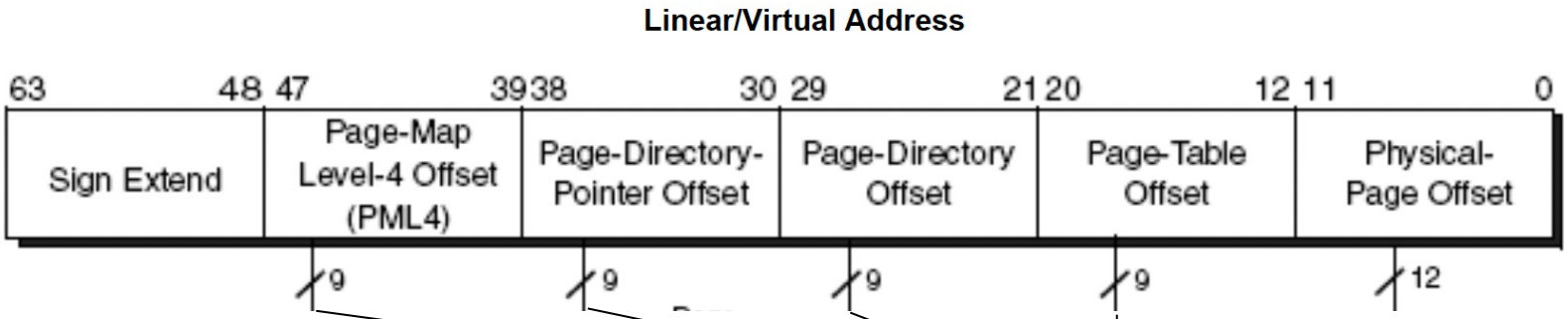
index	
ฐาน 8	ฐาน 10
777_8	511
...	...
010_8	8
007_8	7
...	...
001_8	1
000_8	0

Base



อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table

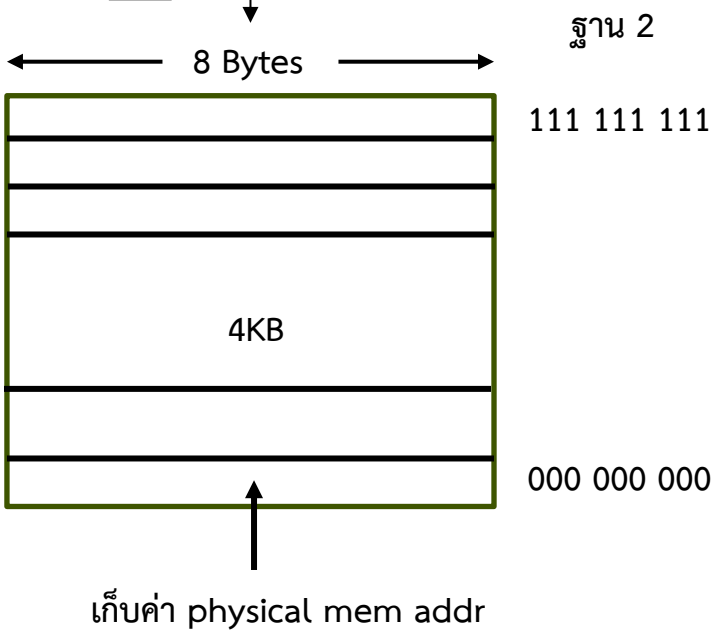


$F(x) = index * 8 + Base$

$F(000_8) = 001_8 * 8 + Base$
$= 001_8 * 2^3 + Base$
$= (001_8 << 3) + Base$
$= 0010_8 + Base$

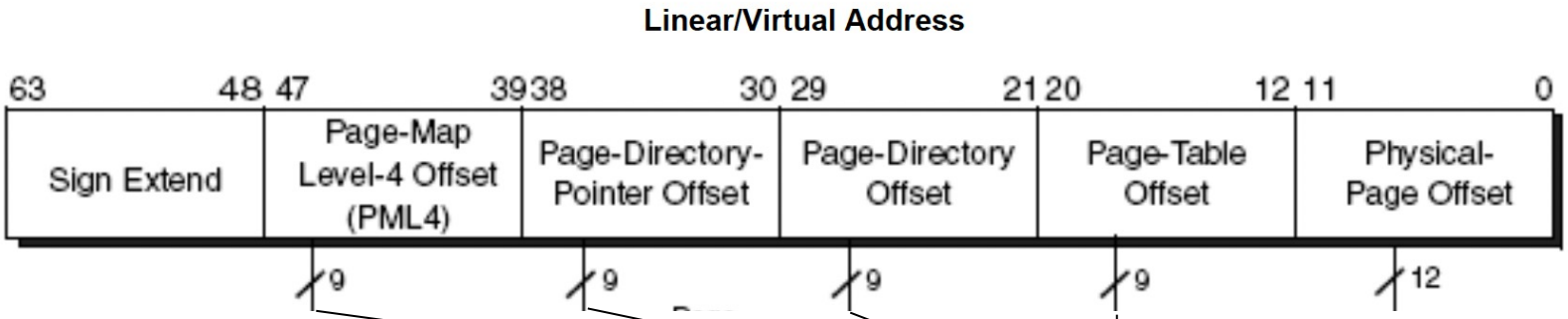
index	
ฐาน 8	ฐาน 10
777_8	511
...	...
010_8	8
007_8	7
...	...
001_8	1
000_8	0

Base



อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table

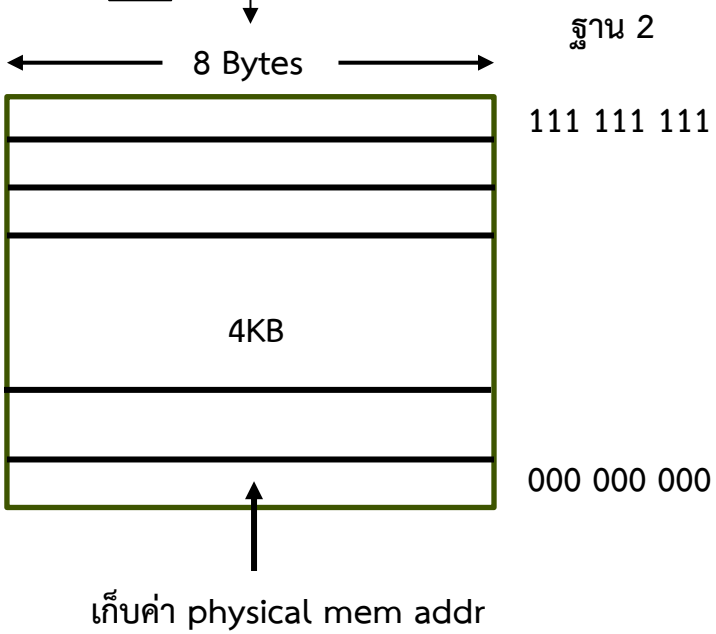


$F(x) = index * 8 + Base$

$F(000_8) = 007_8 * 8 + Base$
$= 007_8 * 2^3 + Base$
$= (007_8 << 3) + Base$
$= 0070_8 + Base$

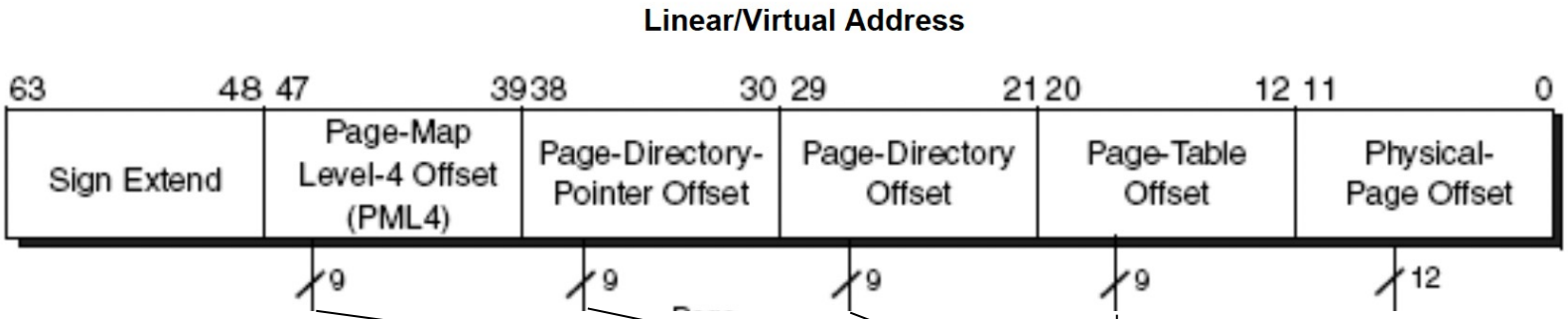
index	
ฐาน 8	ฐาน 10
777_8	511
...	...
010_8	8
007_8	7
...	...
001_8	1
000_8	0

Base



อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table

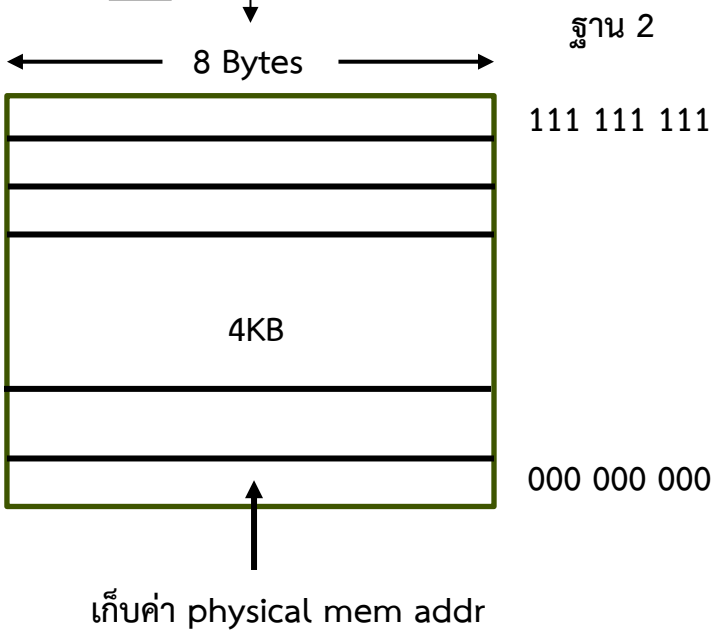


$F(x) = index * 8 + Base$

$F(000_8) = 010_8 * 8 + Base$
$= 010_8 * 2^3 + Base$
$= (010_8 << 3) + Base$
$= 0100_8 + Base$

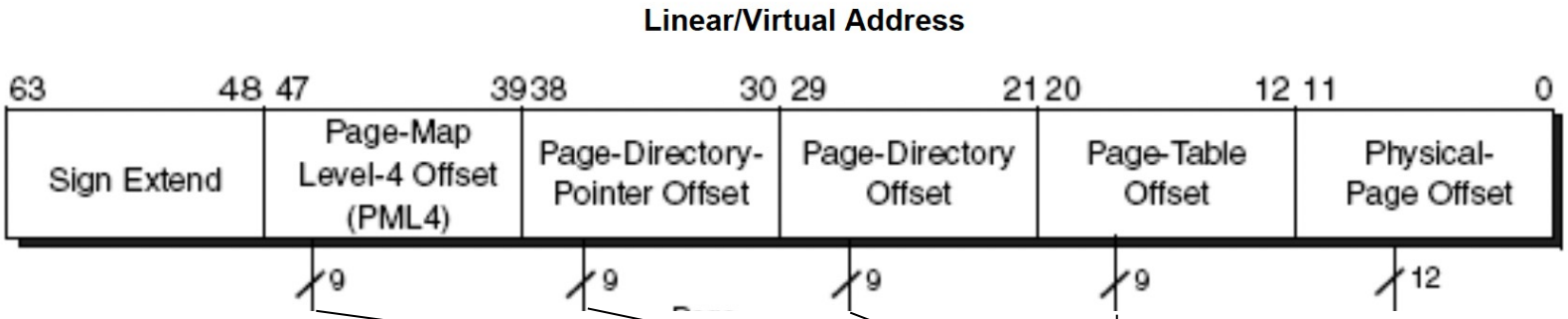
index	
ฐาน 8	ฐาน 10
777_8	511
...	...
010_8	8
007_8	7
...	...
001_8	1
000_8	0

Base



อ้างอิง AMD, AMD-V Nested Paging, 2008

Hierarchical Page table

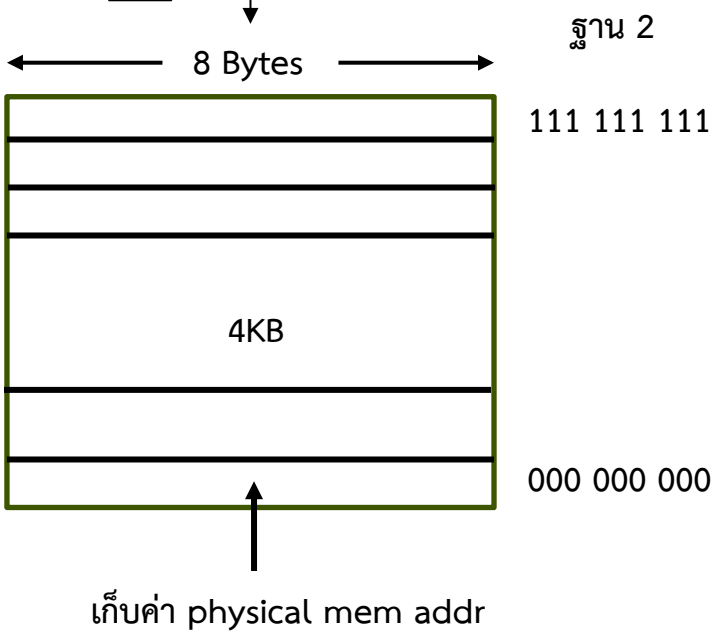


$F(x) = index * 8 + Base$

$F(000_8) = 777_8 * 8 + Base$
$= 777_8 * 2^3 + Base$
$= (777_8 << 3) + Base$
$= 7770_8 + Base$

index	
ฐาน 8	ฐาน 10
777_8	511
...	...
010_8	8
007_8	7
...	...
001_8	1
000_8	0

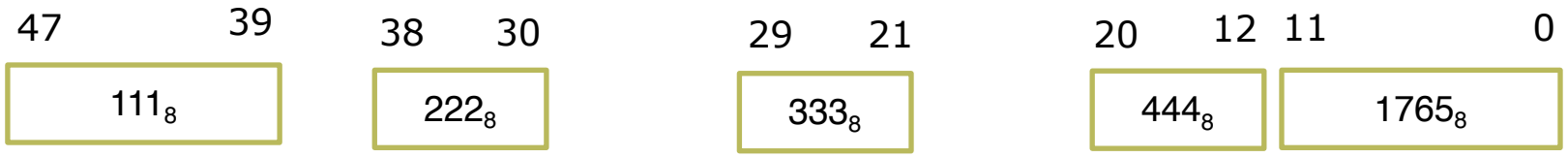
Base

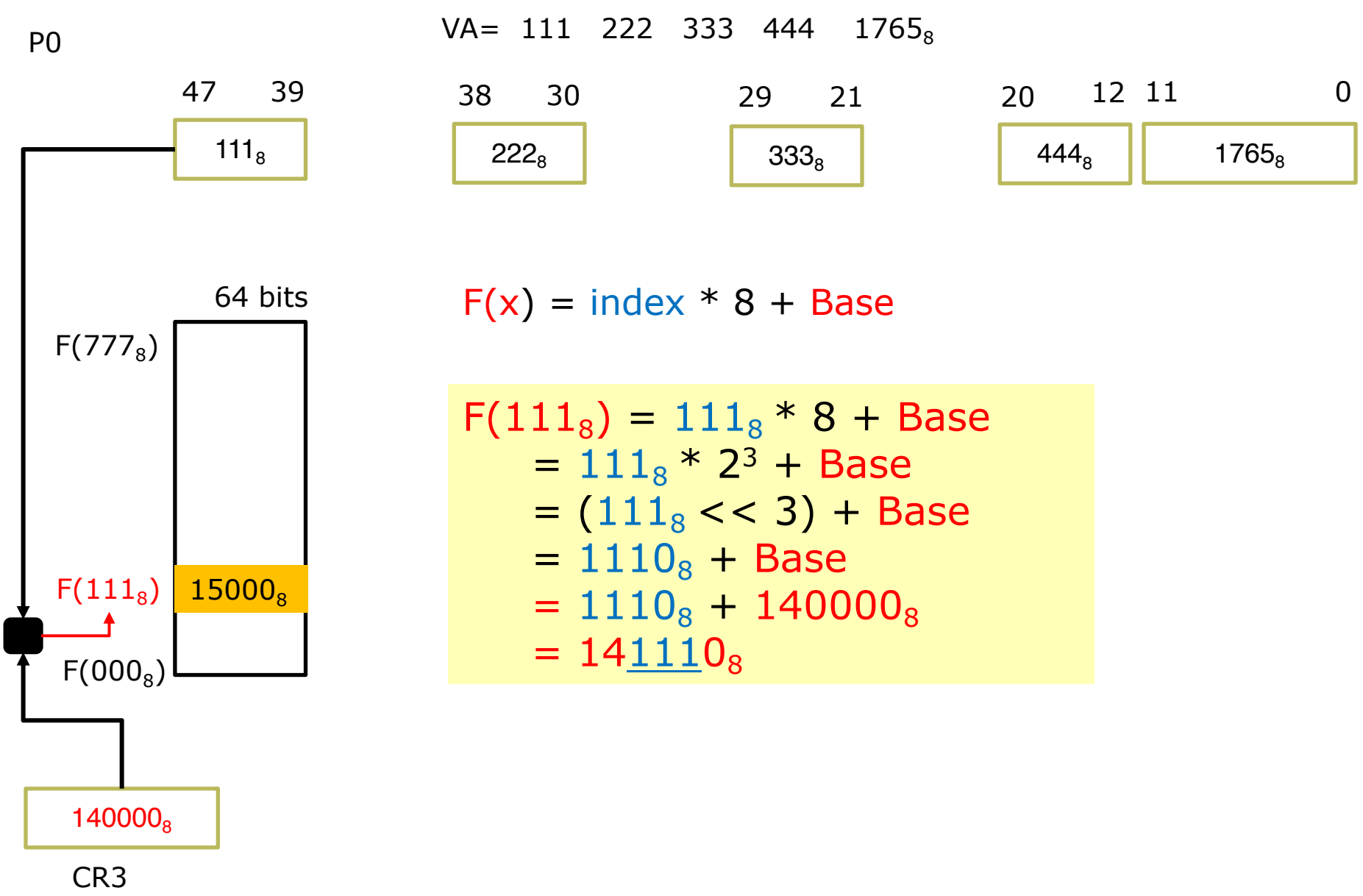


อ้างอิง AMD, AMD-V Nested Paging, 2008

P0

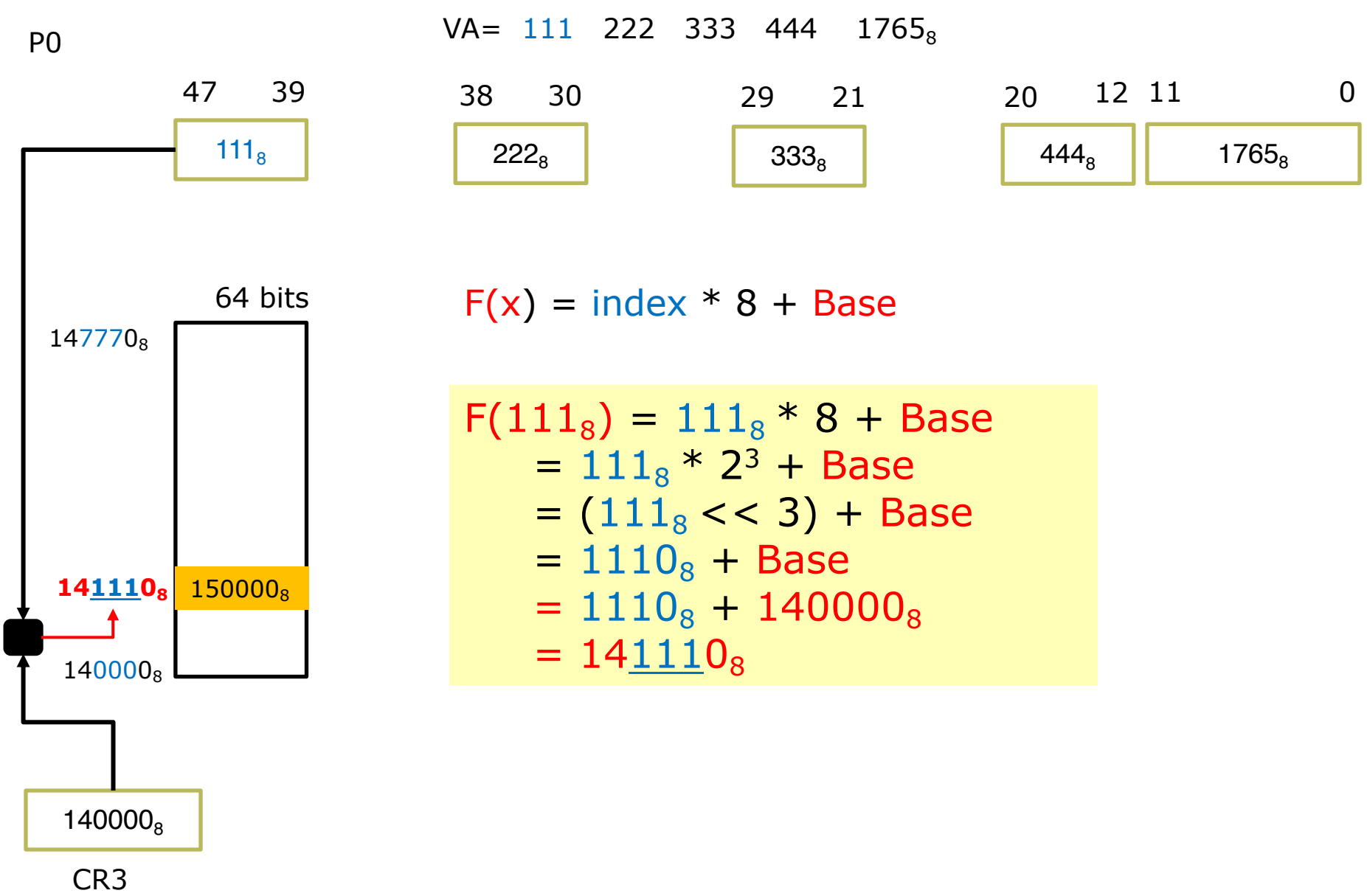
VA= 111 222 333 444 1765_8





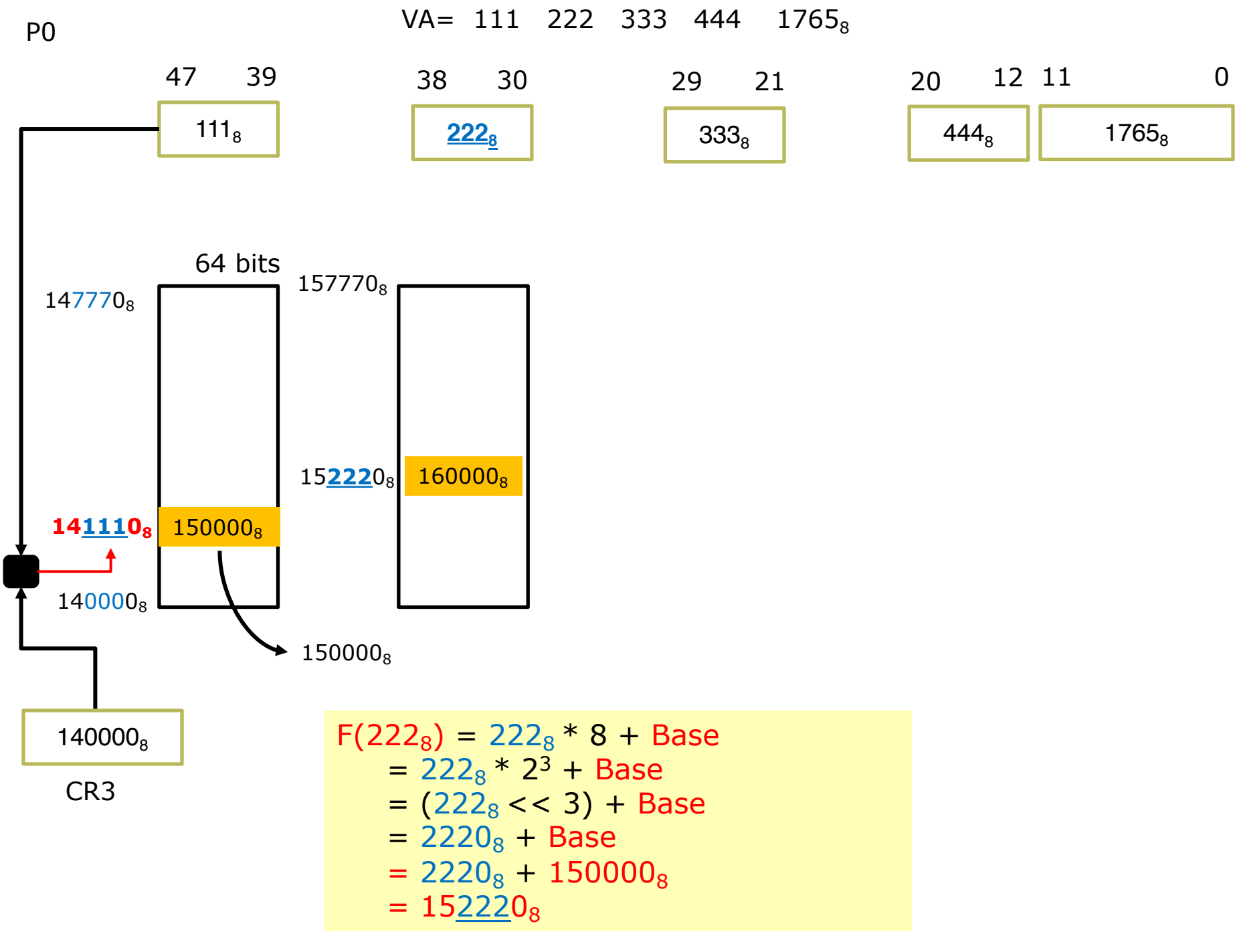
$F(x) = index * 8 + Base$

$$
\begin{aligned}
F(111_8) &= 111_8 * 8 + Base \\
&= 111_8 * 2^3 + Base \\
&= (111_8 << 3) + Base \\
&= 1110_8 + Base \\
&= 1110_8 + 140000_8 \\
&= 14\underline{1110}_8
\end{aligned}
$$

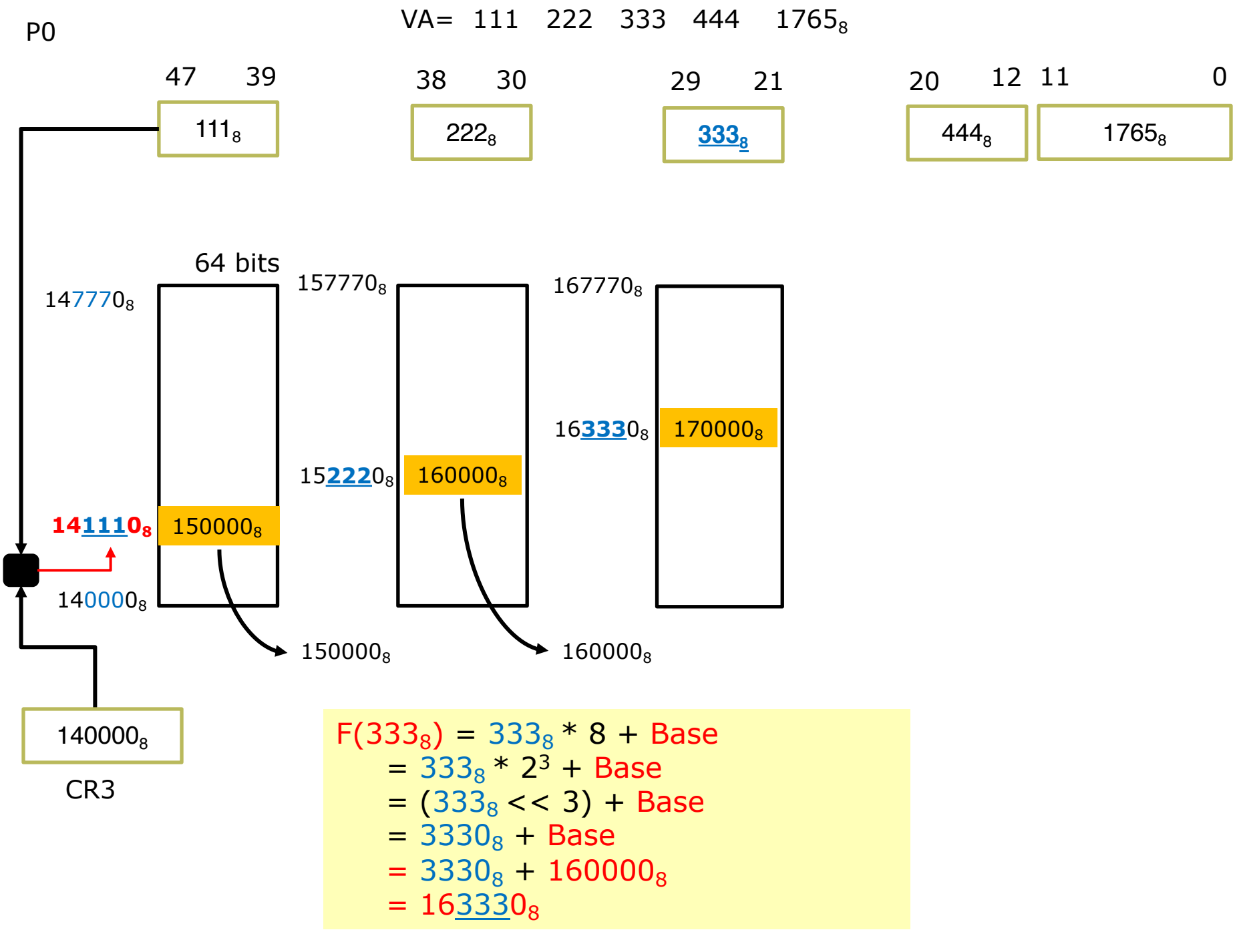


$F(x) = index * 8 + Base$

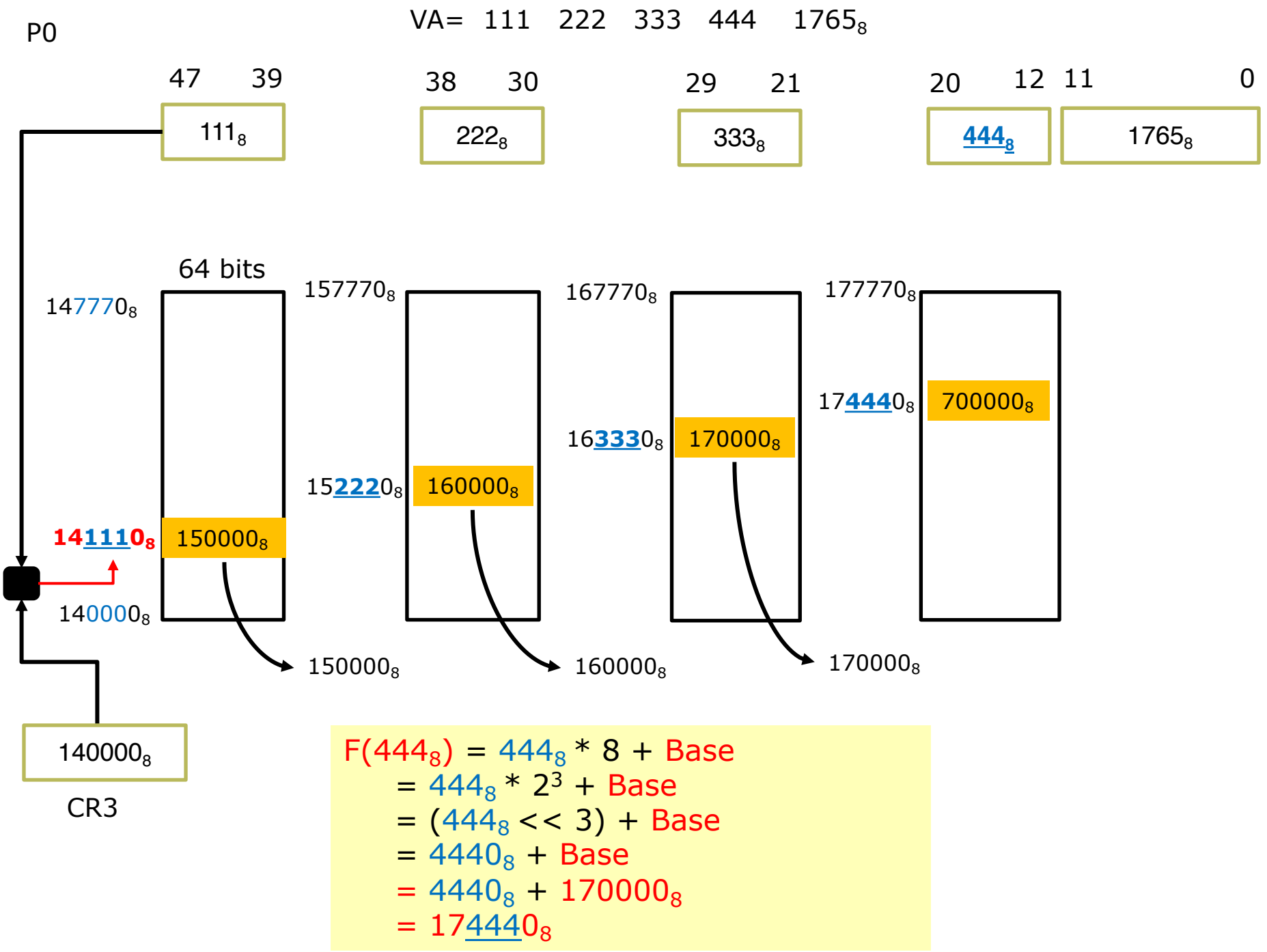
$$
\begin{aligned}
F(111_8) &= 111_8 * 8 + Base \\
&= 111_8 * 2^3 + Base \\
&= (111_8 << 3) + Base \\
&= 1110_8 + Base \\
&= 1110_8 + 140000_8 \\
&= 141110_8
\end{aligned}
$$

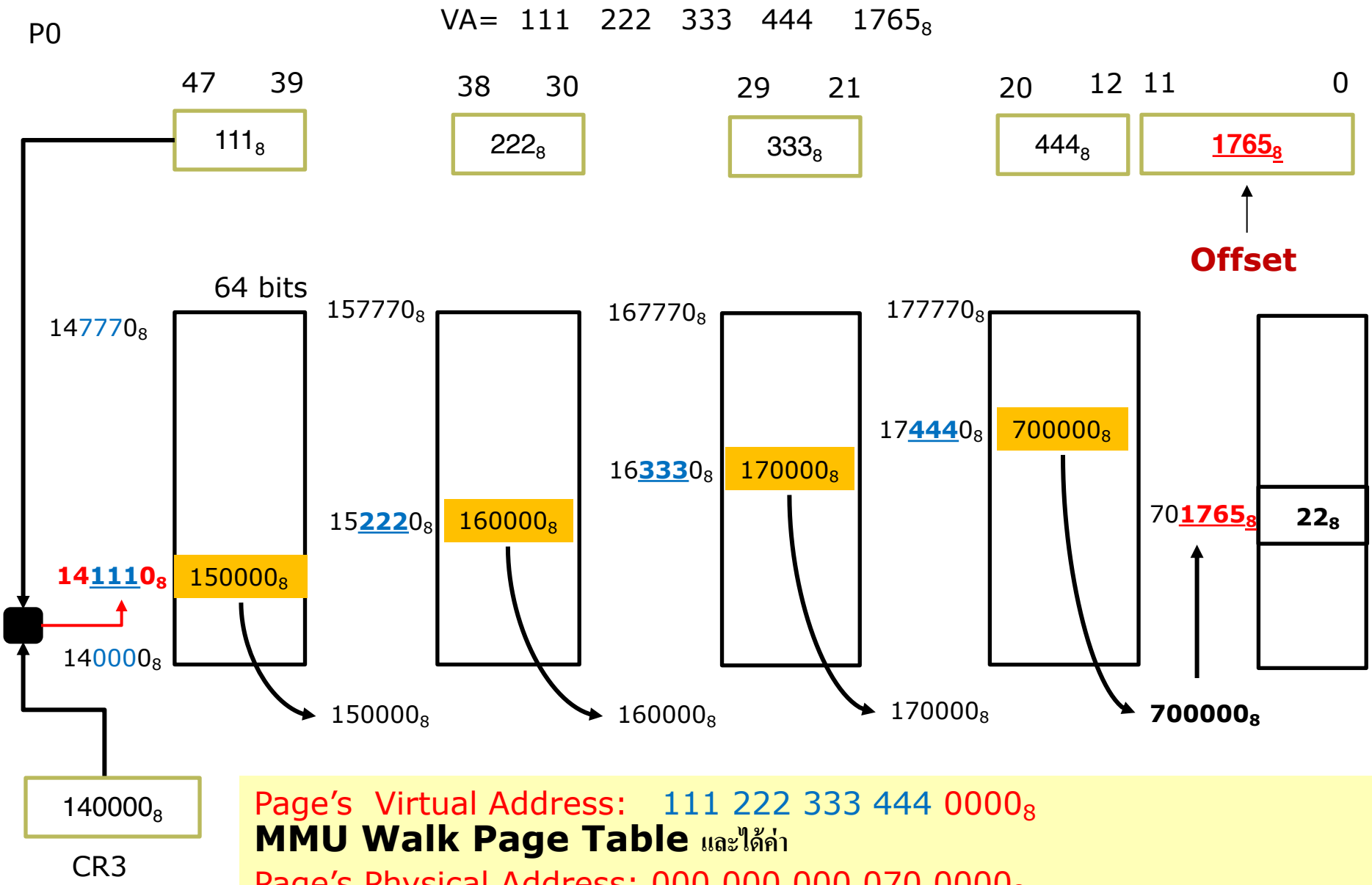


$F(222_8) = 222_8 * 8 + Base$
$= 222_8 * 2^3 + Base$
$= (222_8 << 3) + Base$
$= 2220_8 + Base$
$= 2220_8 + 150000_8$
$= 152220_8$



$F(333_8) = 333_8 * 8 + Base$
$= 333_8 * 2^3 + Base$
$= (333_8 << 3) + Base$
$= 3330_8 + Base$
$= 3330_8 + 160000_8$
$= 163330_8$


P0
VA= 111 222 333 444 1765$_8$
47 39
38 30
29 21
20 12
11 0
111$_8$
222$_8$
333$_8$
444$_8$
1765$_8$
64 bits
147770$_8$
157770$_8$
167770$_8$
177770$_8$
141110$_8$
150000$_8$
140000$_8$
152220$_8$
160000$_8$
163330$_8$
170000$_8$
174440$_8$
700000$_8$
150000$_8$
160000$_8$
170000$_8$
140000$_8$
CR3
F(444$_8$) = 444$_8$ * 8 + Base
= 444$_8$ * 2^3 + Base
= (444$_8$ << 3) + Base
= 4440$_8$ + Base
= 4440$_8$ + 170000$_8$
= 174440$_8$



Page's Virtual Address: 111 222 333 444 0000₈

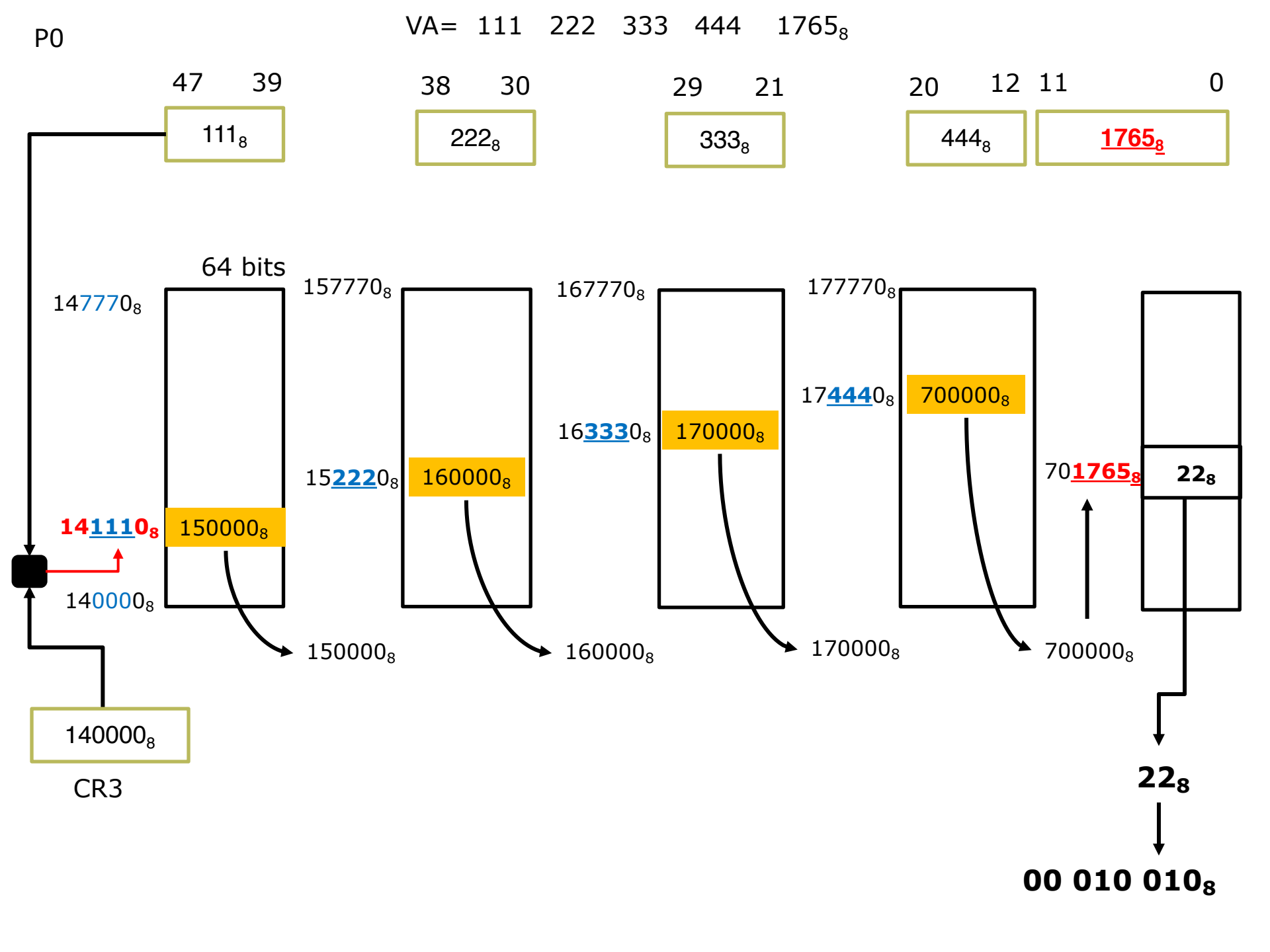
MMU Walk Page Table และได้ค่า

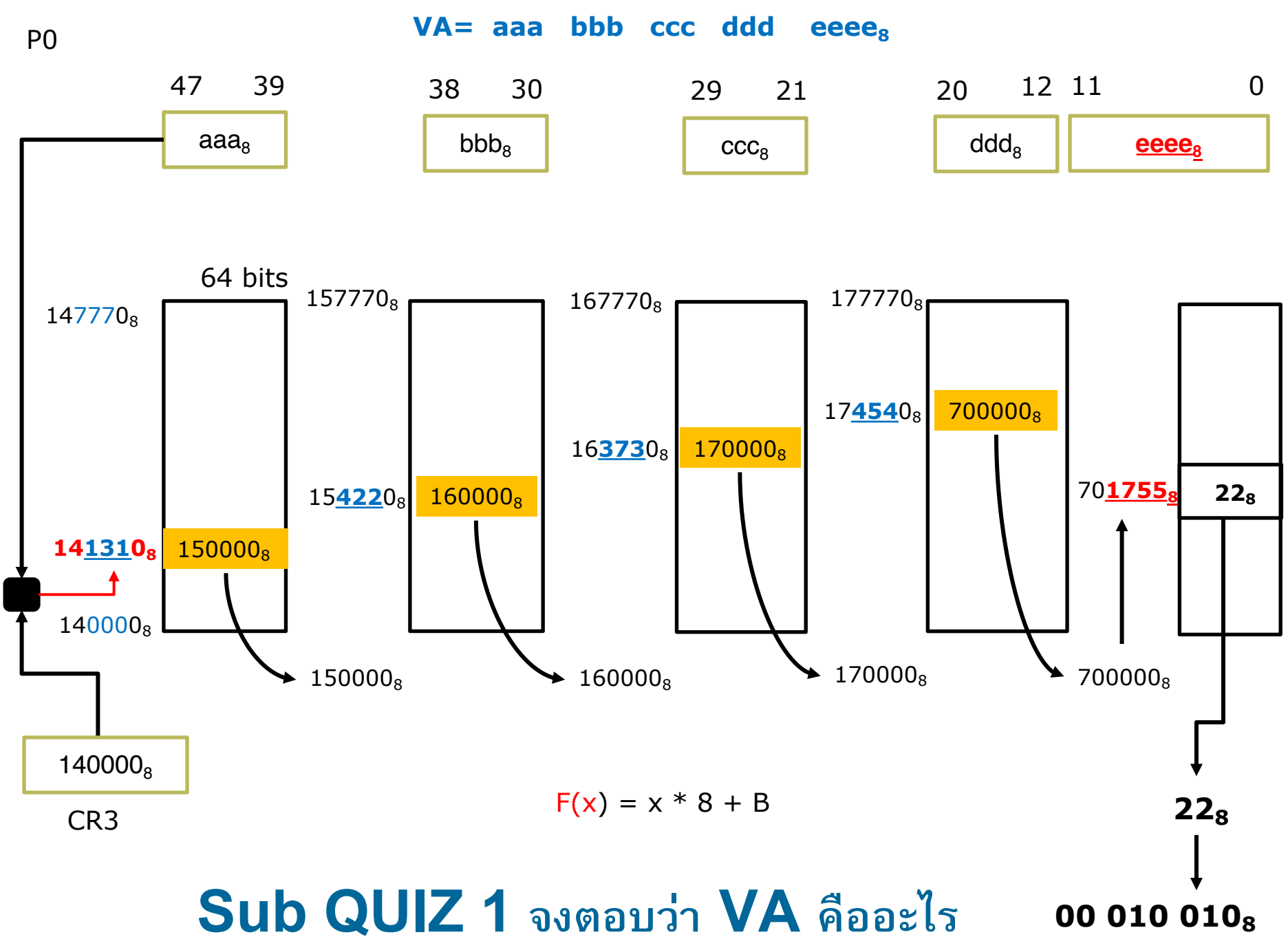
Page's Physical Address: 000 000 000 070 0000₈

ดังนั้น Virtual Address: 111 222 333 444 1765₈

จะได้ Physical Address: 000 000 000 0**70 0000₈** + 1765₈ = **70 1765₈**

Offset

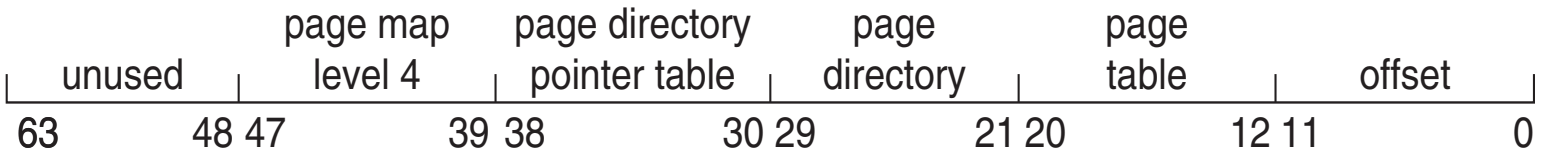

P0
VA= 111 222 333 444 1765₈
47 39
38 30
29 21
20 12
11 0
111₈
222₈
333₈
444₈
1765₈
64 bits
147770₈
157770₈
167770₈
177770₈
141110₈
150000₈
152220₈
160000₈
163330₈
170000₈
174440₈
700000₈
701765₈
22₈
140000₈
150000₈
160000₈
170000₈
700000₈
140000₈
CR3
22₈
00 010 010₈

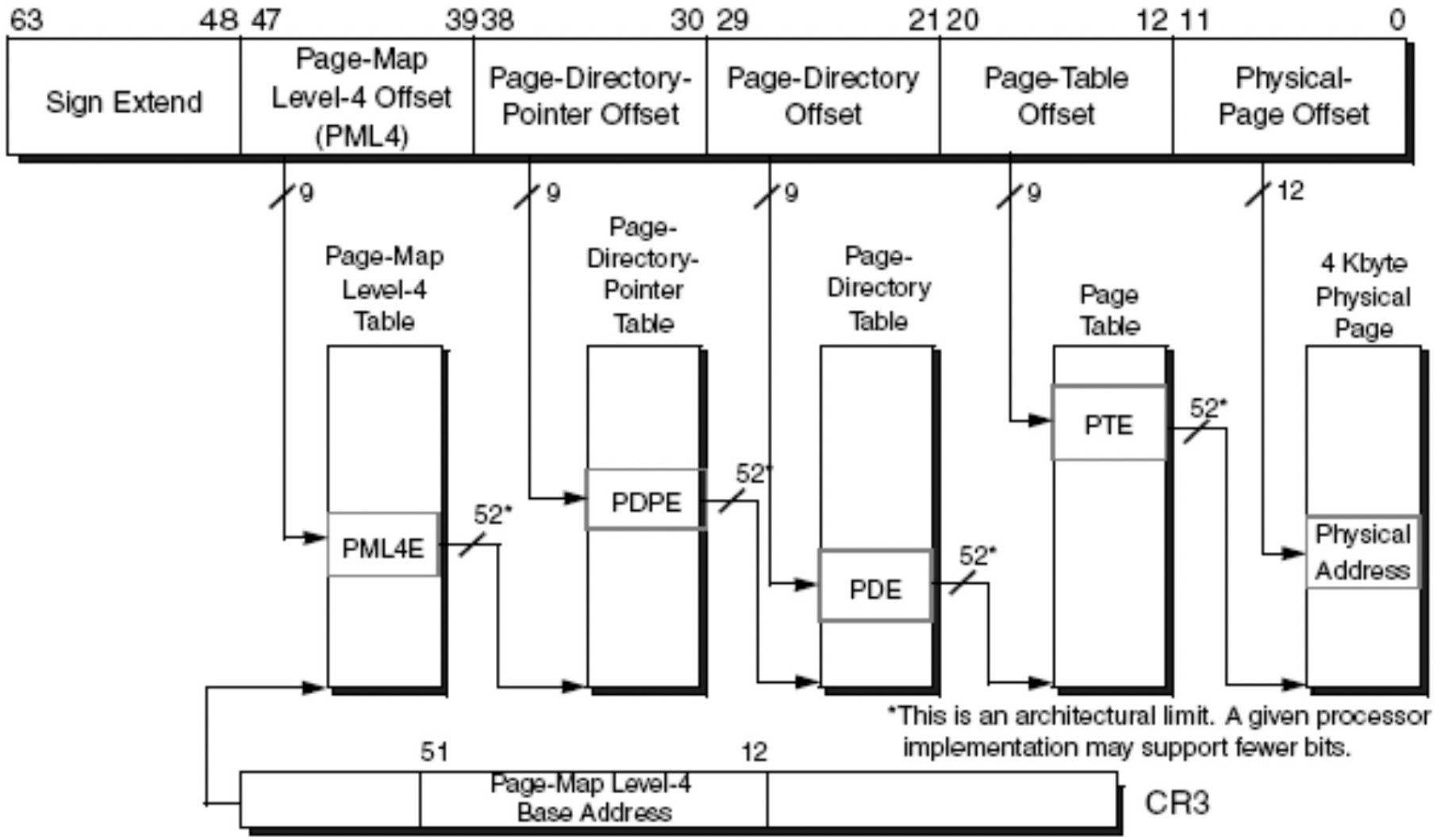


Sub QUIZ 1 จงตอบว่า VA คืออะไร

Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits (เราไม่พิจารณาเรื่อง PAE ในวิชานี้)

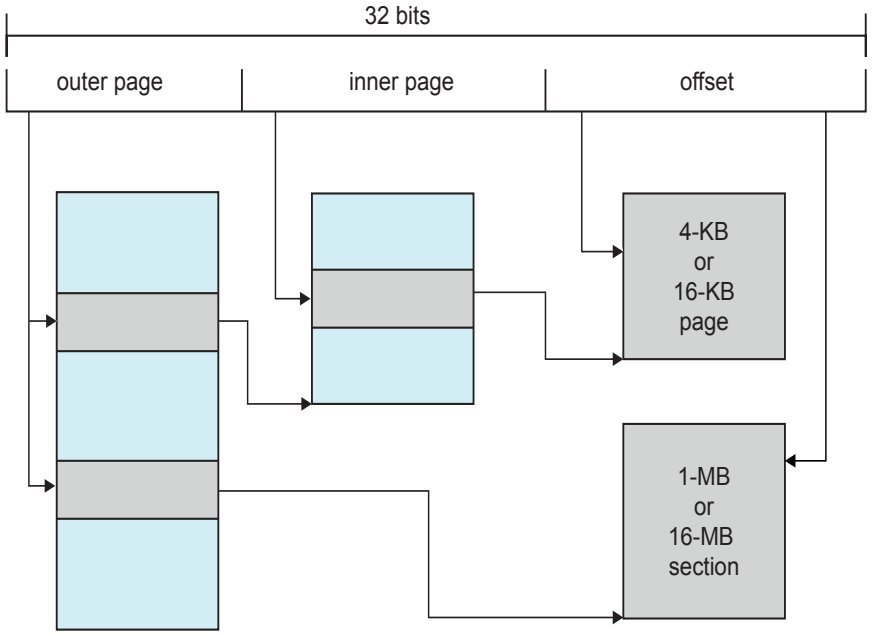




อ้างอิง AMD, AMD-V Nested Paging, 2008

Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



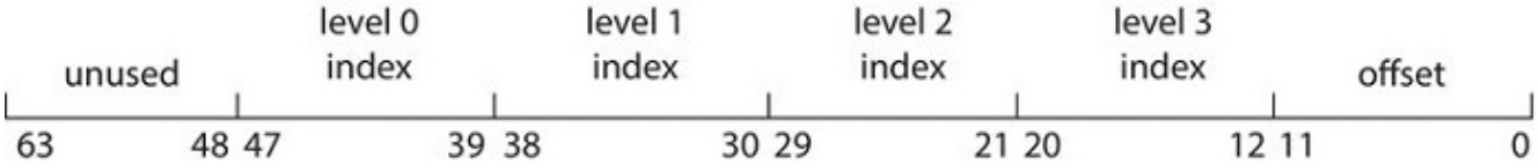


Figure 9.26 ARM 4-KB translation granule.

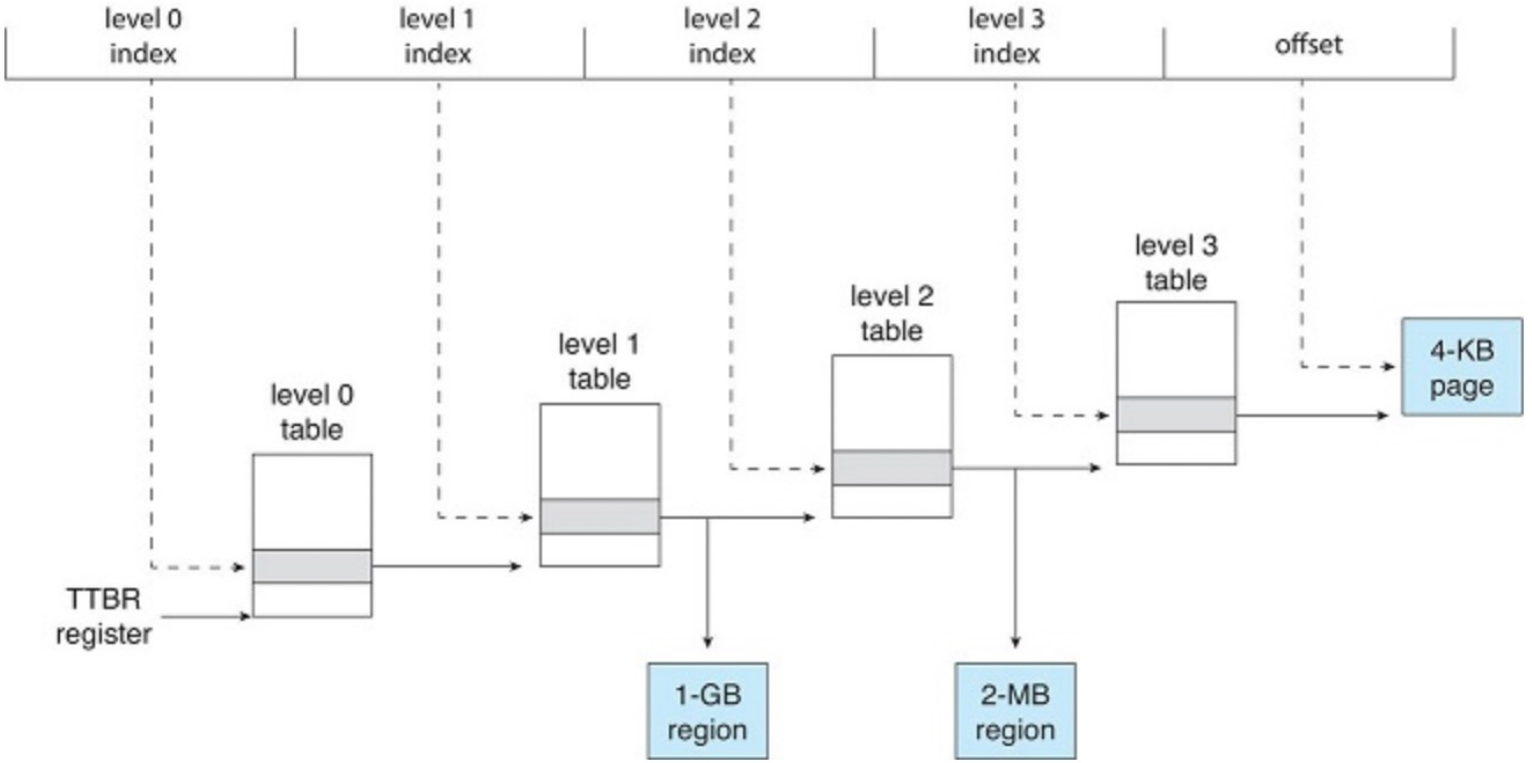
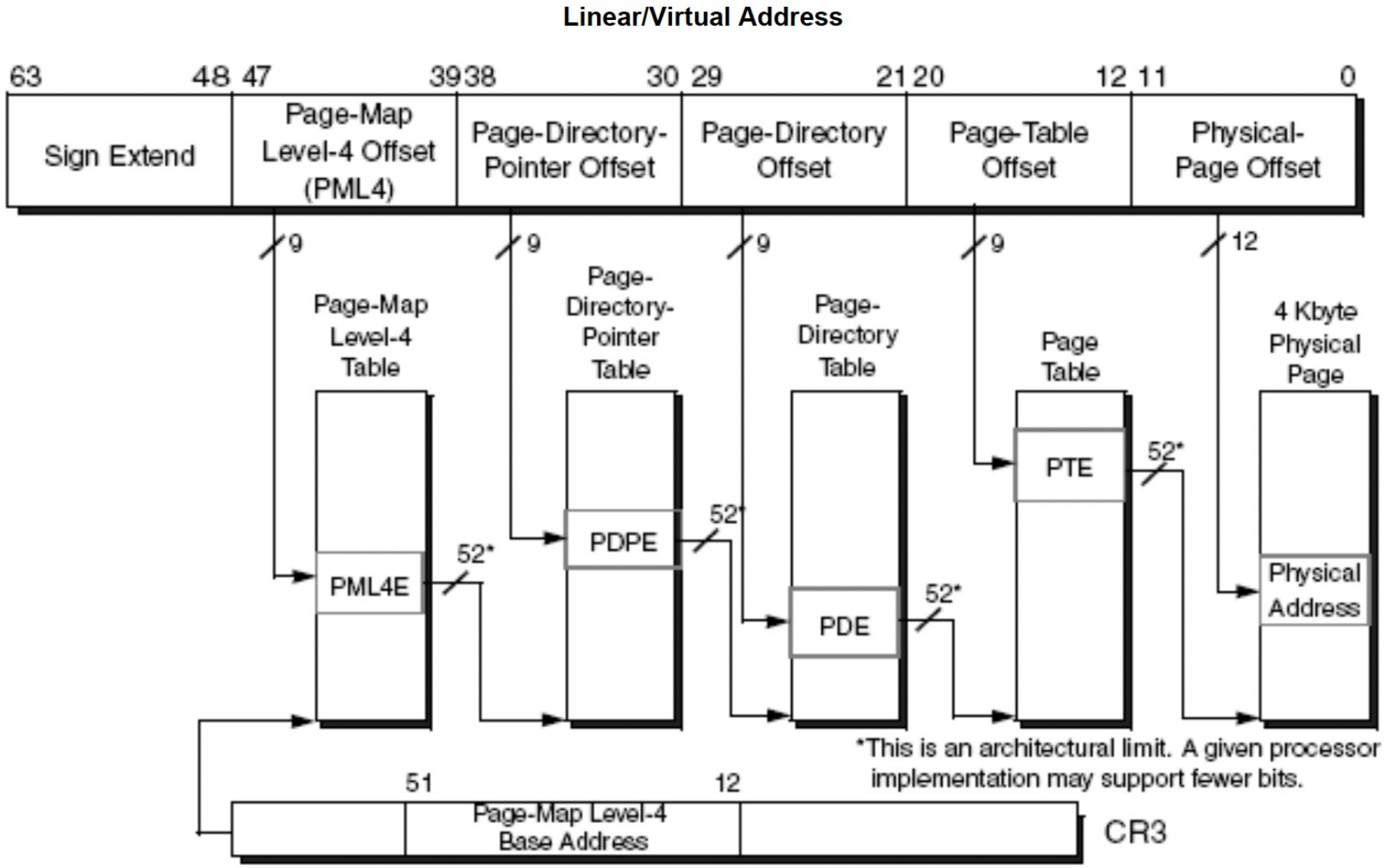


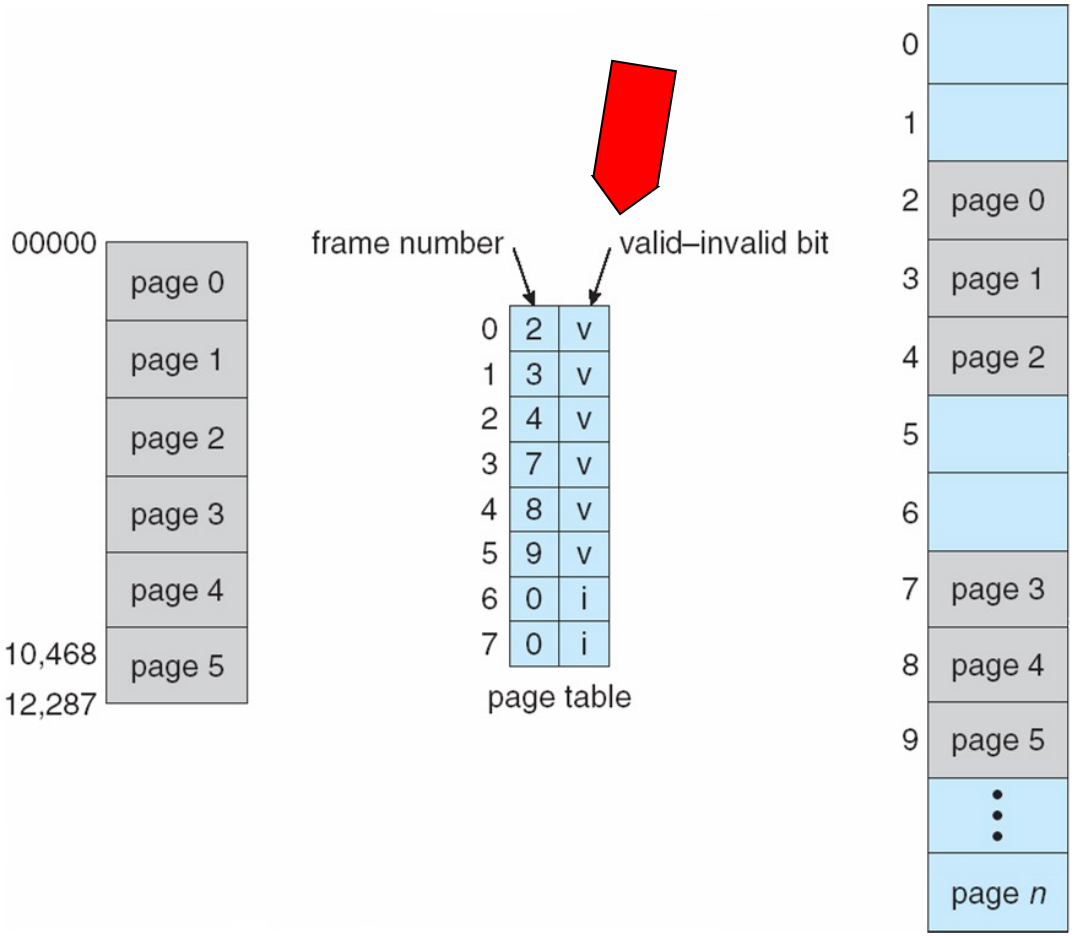
Figure 9.27 ARM four-level hierarchical paging.

Revisit: x86_64 Architecture

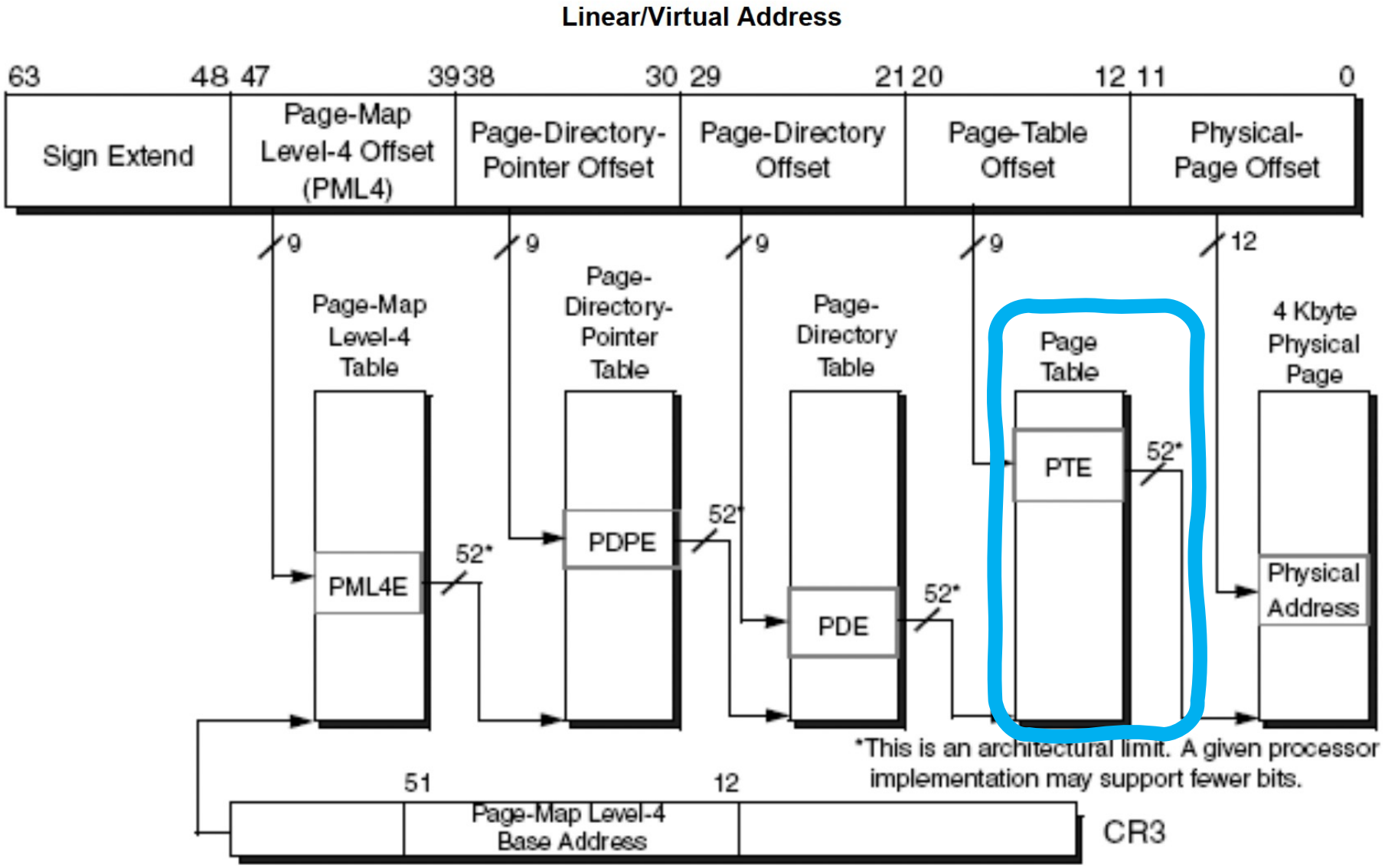


อ้างอิง AMD, AMD-V Nested Paging, 2008

So, where Linux store those Valid (v) or Invalid (i) Bit In A Page Table?

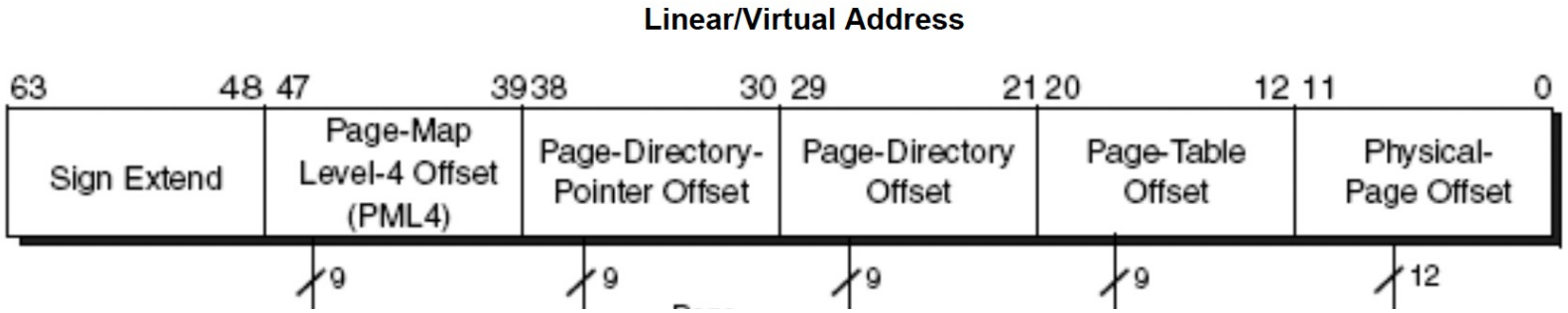


มันอยู่ใน Page Table Entry (PTE)

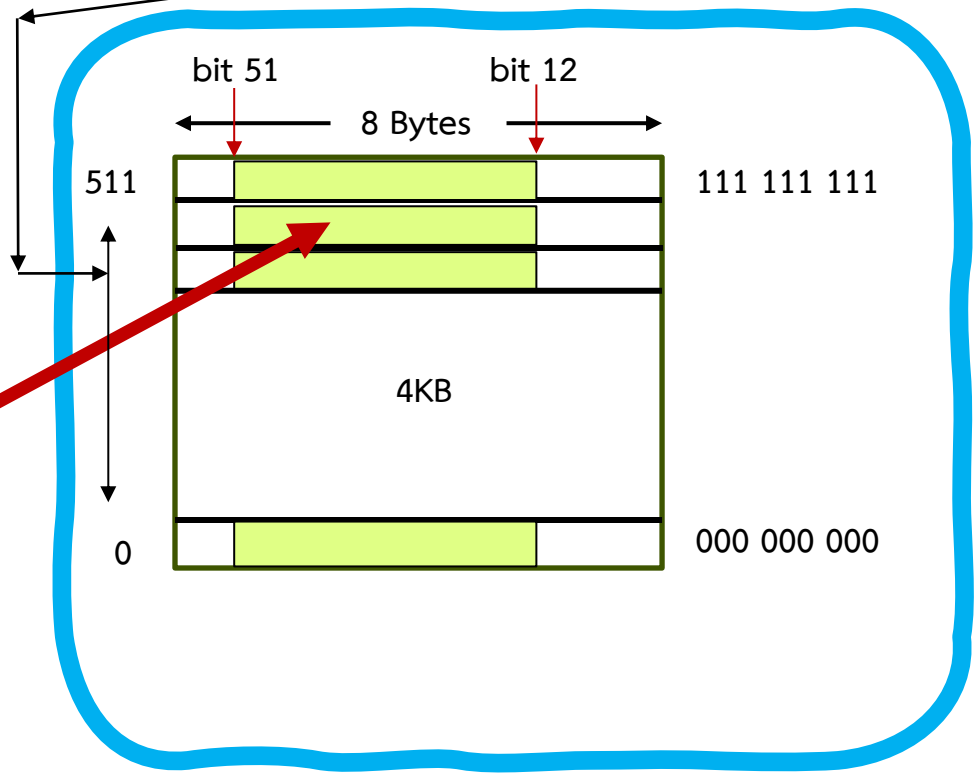


อ้างอิง AMD, AMD-V Nested Paging, 2008

Page table Entry

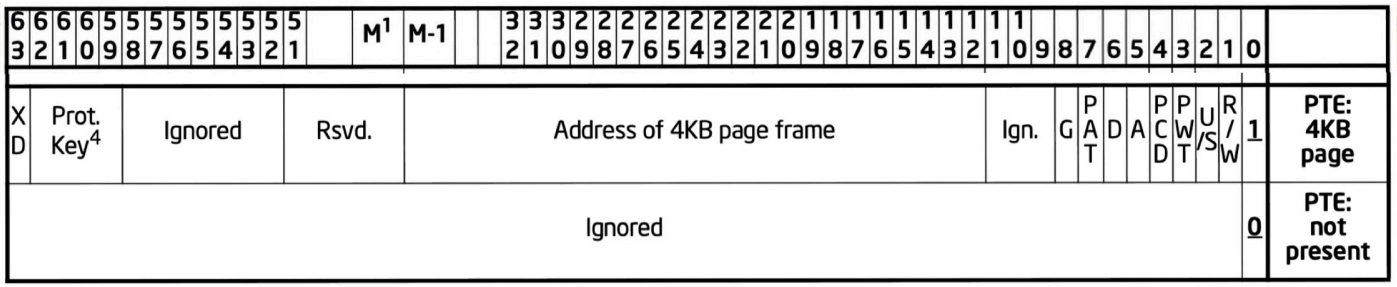


- PTE มีขนาดรวม 4KB
- เป็น Array ของ Element 512 Element
- แต่ละ Element มีขนาด 64 bits (หรือ 8 bytes)
- แต่ละ Element ใช้เพียงแค่ bit ที่ 12 ถึง 51 เพื่อเก็บ physical memory address ที่ชี้ไปยัง target physical mem frame
- Bit ที่เหลือเอาไว้เก็บค่า status



อ้างอิง AMD, AMD-V Nested Paging, 2008

x86_64 Page table Entry Format



Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1, determines the protection key of the page (see Section 4.6.2); ignored otherwise
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

อ้างอิง https://courses.cs.washington.edu/courses/cse451/24sp/lectures/notes/x86-paging.html

x86_64 Page table Entry Format
(อ้างอิง Perplexity AI)

Bits	Field/Usage
0	Present (P)
1	Read/Write (R/W)
2	User/Supervisor (U/S)
3	Page-level Write-Through
4	Page-level Cache Disable
5	Accessed (Reference)
6	Dirty
7	Page Size (for higher levels)
8	Global
9-11	Available to OS
12-51	Physical page frame address
52-62	Available to OS
63	No-execute (NX)

Valid Bit (bit 0)

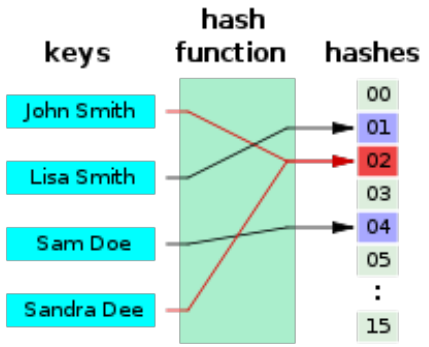
Reference Bit (bit 5)

Dirty Bit (bit 6)

Physical Address

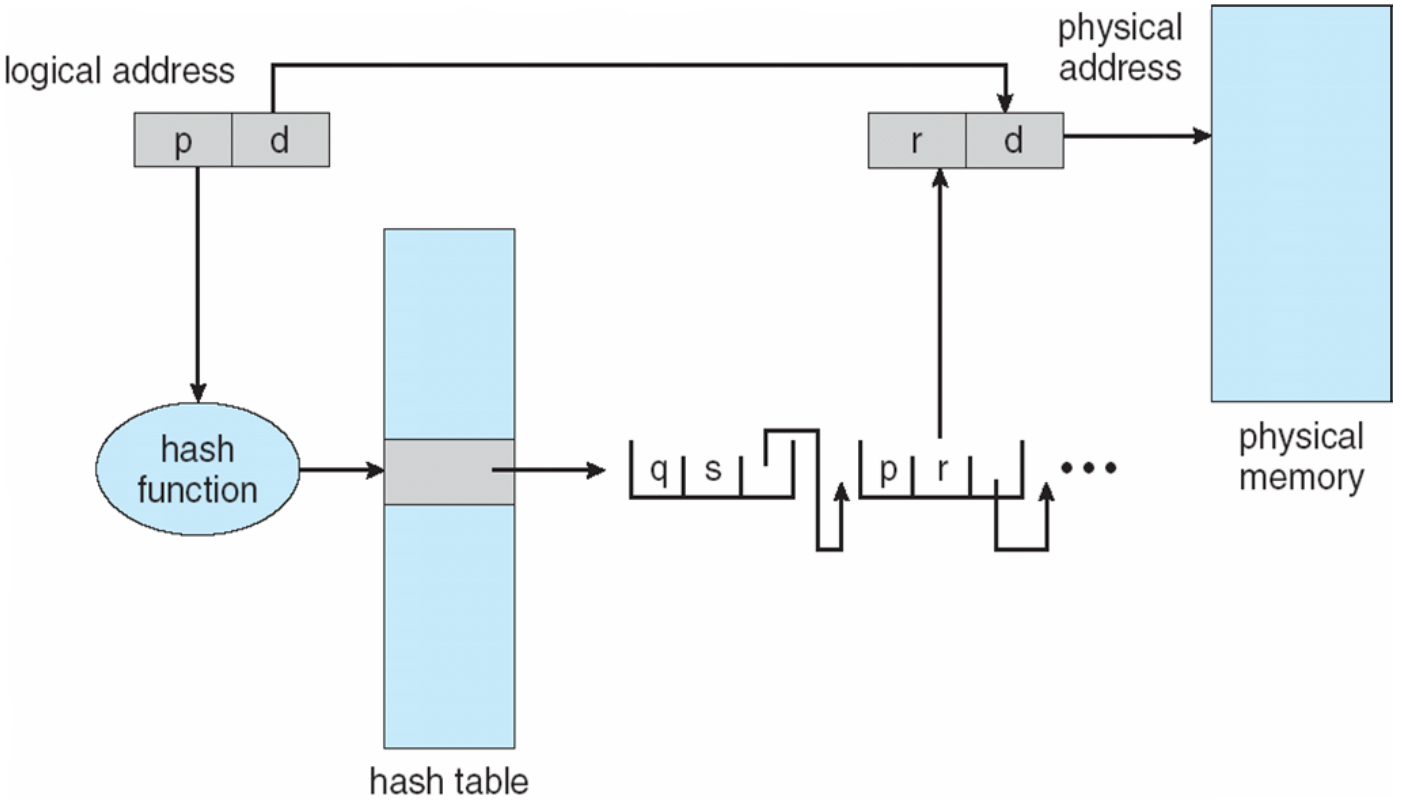
เอา bit ที่ 12 ถึง 51 มาใช้ และเติมค่า 0 ใน bit 0 ถึง 11

Hashed Page Tables



- hash function คือ function ที่ map ค่าข้อมูล ให้เป็นค่าที่มีขนาดจำกัด
  - ยกตัวอย่าง เช่น f(x) = x mod 10
  - ถ้าได้ค่า f(x) เดียวกัน จากค่า x ต่างกัน เรียกว่าเกิด collision
  - https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/HashFuncExamp.html
  - https://en.wikipedia.org/wiki/Hash_function
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Hashed Page Table



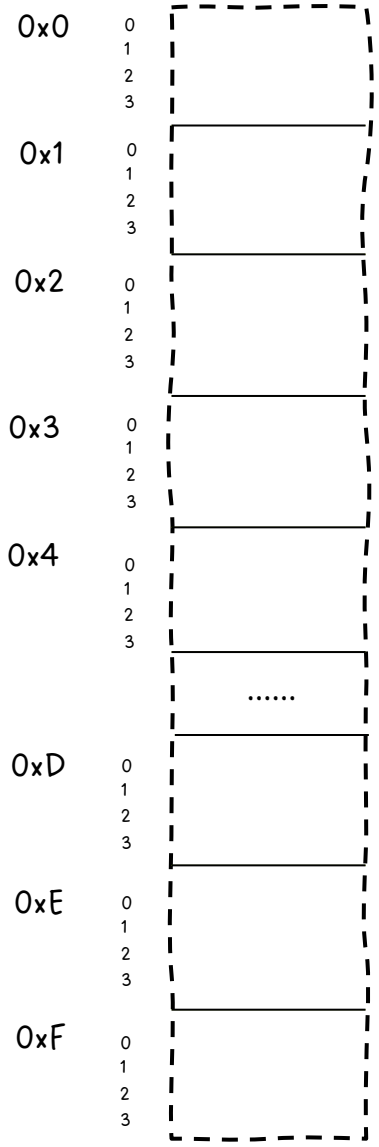
ในตัวอย่างนี้ hash(p) = hash(q) = x

จะต้องมี list ของ page table entries สำหรับแก้ปัญหา collision

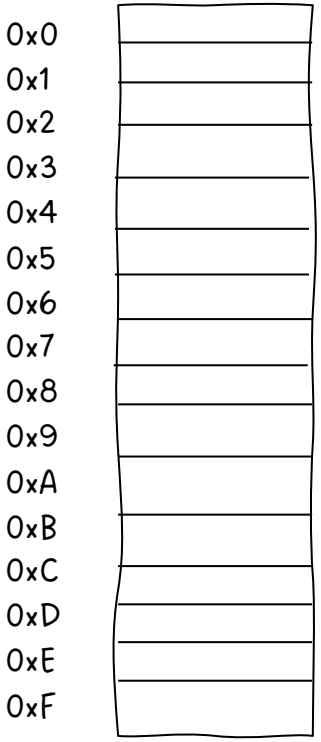
Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)

Page id (p) : offset (d)

4 bits : 2 bits

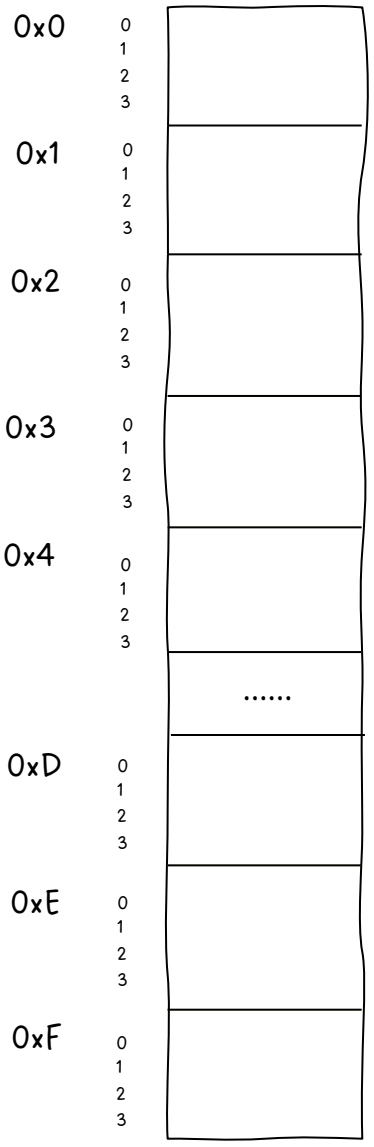


Page id (p)



Frame id (f) : offset (d)

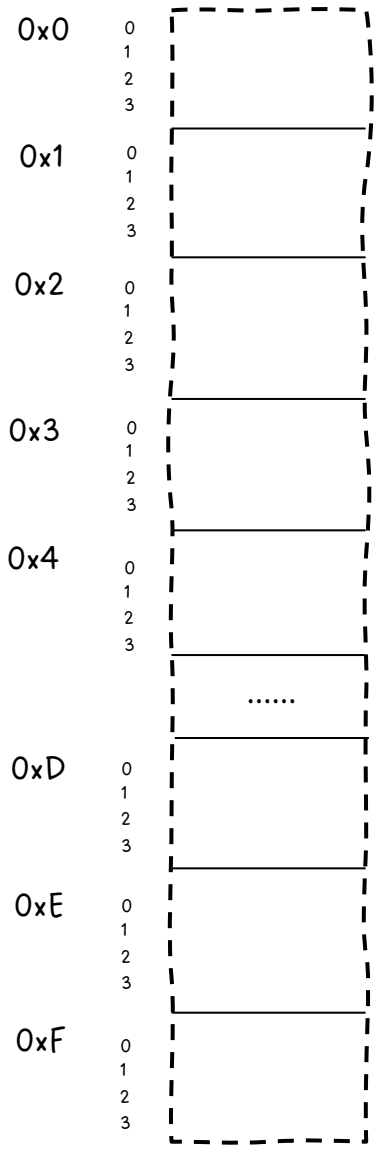
4 bits : 2 bits



Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 64 bytes (16 pages)

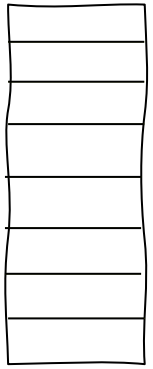
Page id (p) : offset (d)

4 bits : 2 bits



0x0 = 0 000
0x1 = 0 001
0x2. = 0 010
0x3 = 0 011
0x4 = 0. 100
0x5 = 0 101
0x6 = 0 110
0x7 = 0 111
0x8 = 1 000
0x9 = 1 001
0xA = 1 010
0xB = 1 011
0xC = 1 100
0xD = 1 101
0xE = 1 110
0xF = 1 111

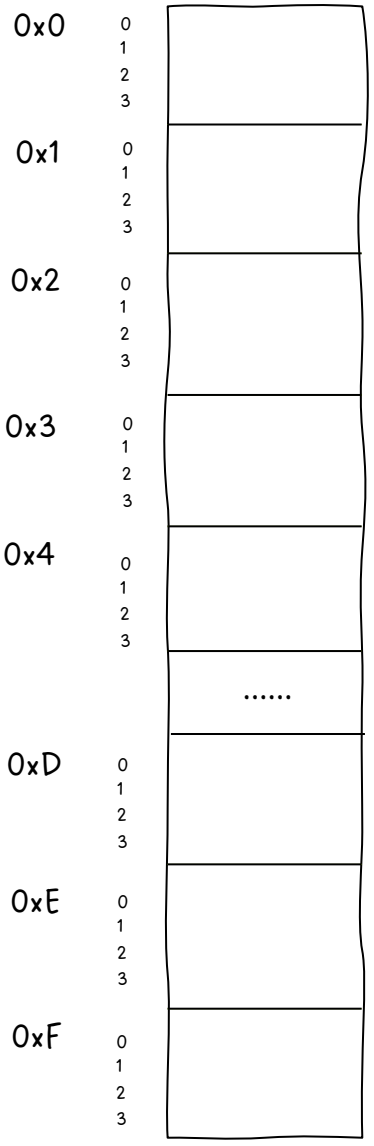
X = hash(p) = p % 2^3
= p % 8
= p % 0b1000



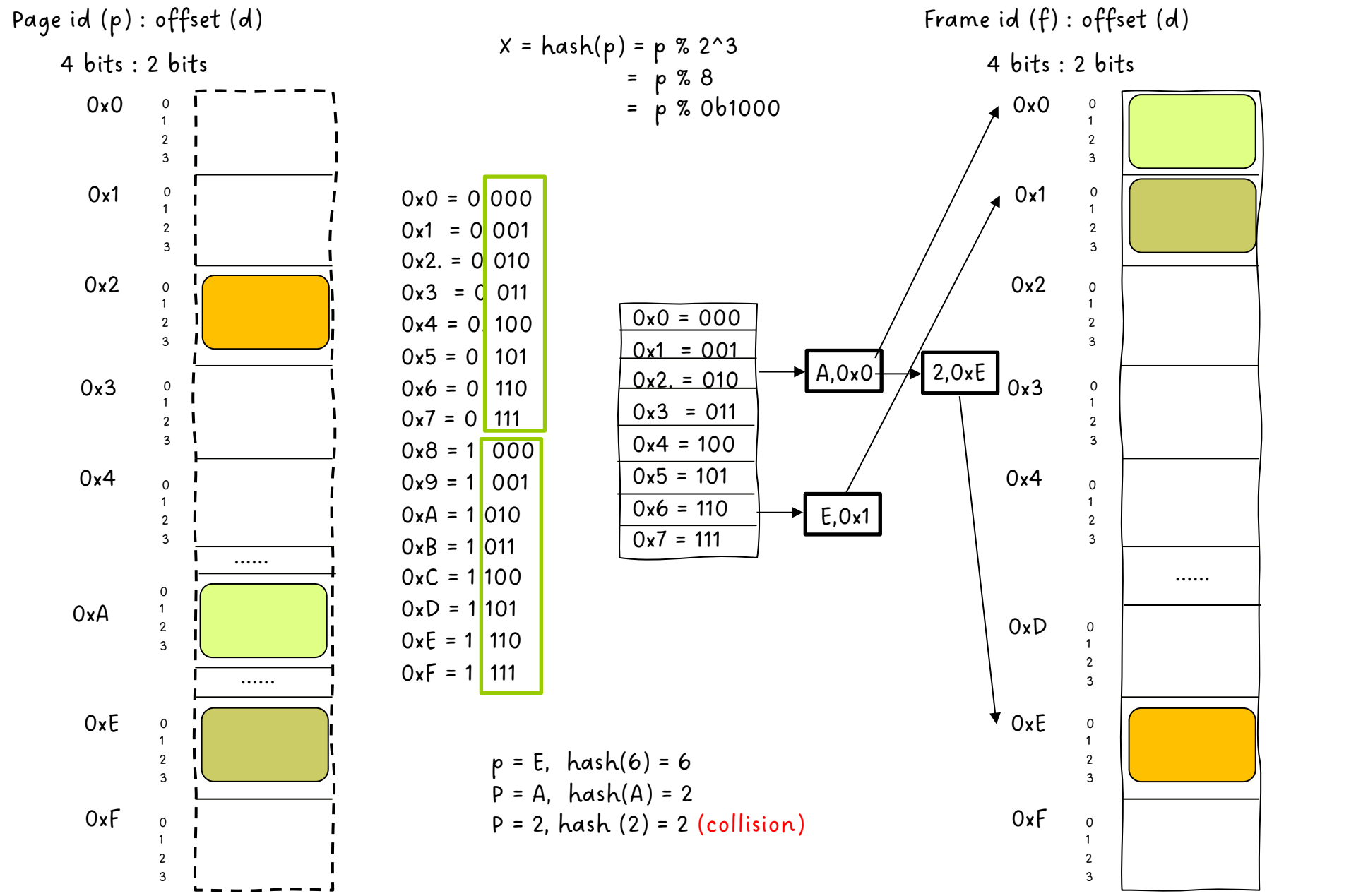
0x0 = 000
0x1 = 001
0x2. = 010
0x3 = 011
0x4 = 100
0x5 = 101
0x6 = 110
0x7 = 111

Frame id (f) : offset (d)

4 bits : 2 bits



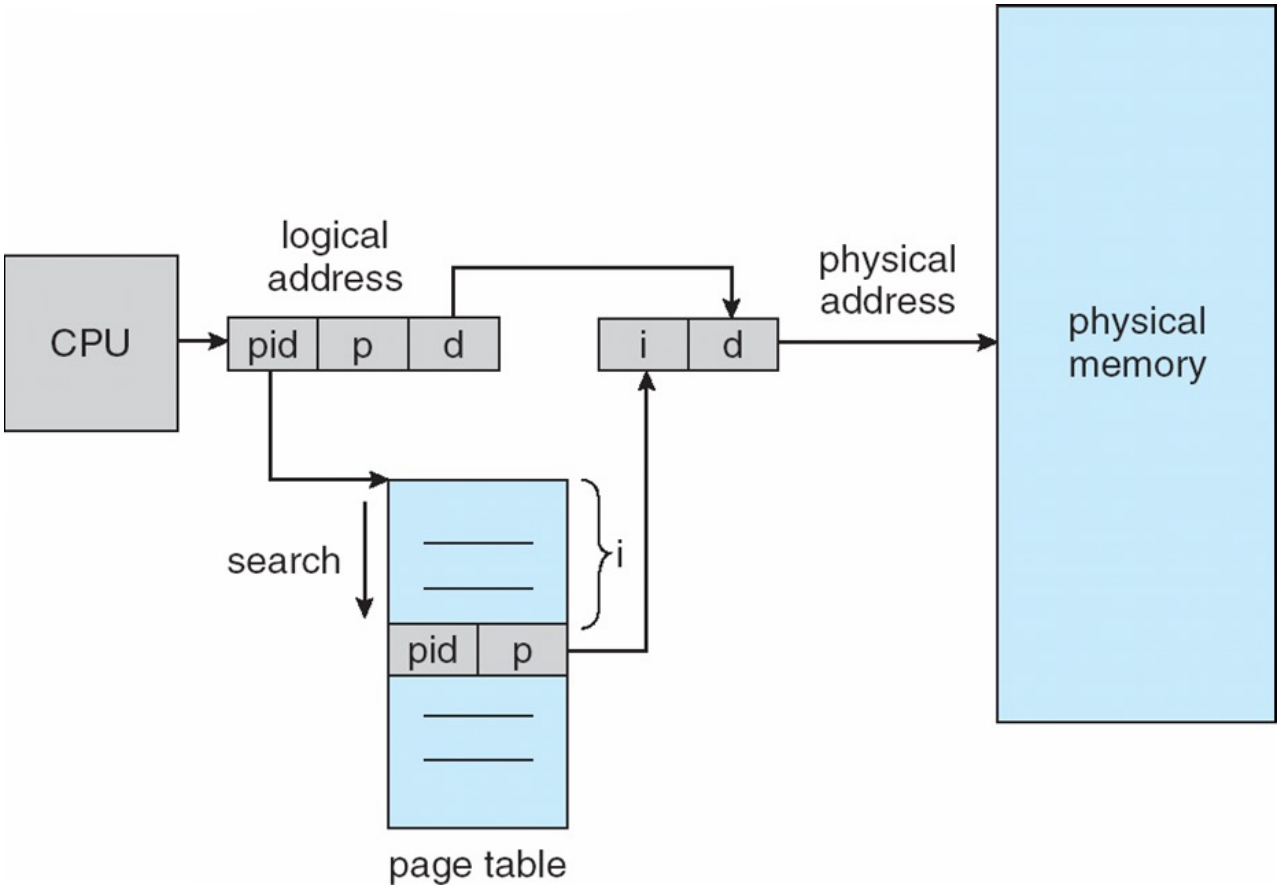
Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 32 bytes (16 pages)



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, *with information about the process that owns that page*
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address
  - ในช่อง frame หนึ่งช่อง เก็บค่า page id ได้ค่าเดียว

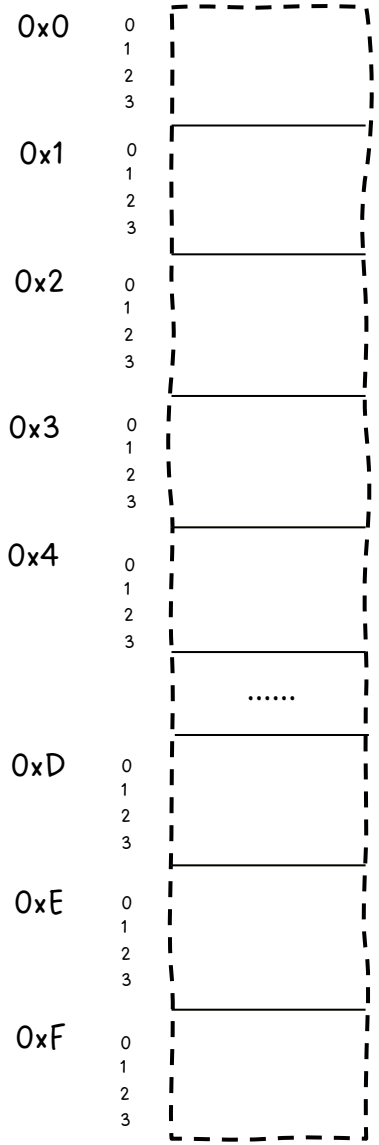
Inverted Page Table Architecture



Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 32 bytes (16 pages)

Page id (p) : offset (d)

4 bits : 2 bits

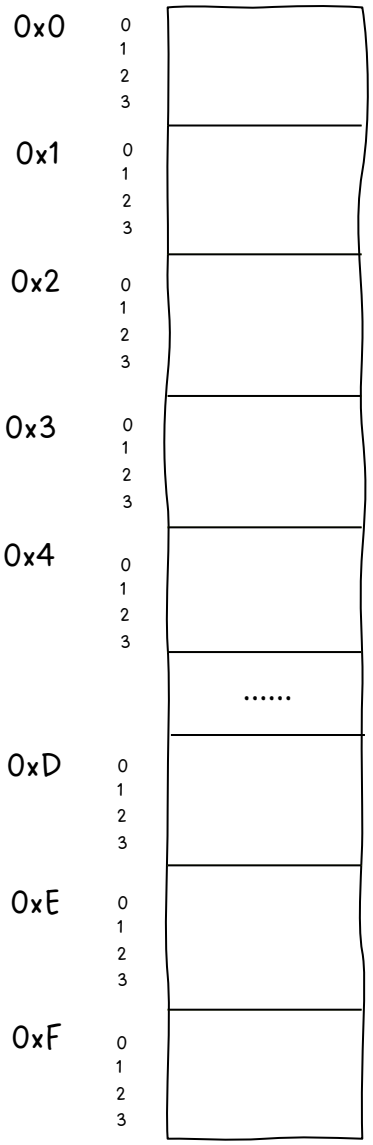


Inverted Page table

Frame id	PID	Page id
0x0	P1	0x11
0x1	P2	0x3C
0x2	P2	0x06
0x3	----	----
0x4	----	----
0x5	P1	0x10
0x6	----	----
0x7	P2	0x2E
0x8	P1	0x0F
0x9	----	----
0xA	P1	0x1B
0xB	P2	0x34
0xC	----	----
0xD	P1	0x2A
0xE	P1	0x14
0xF	----	----

Frame id (f) : offset (d)

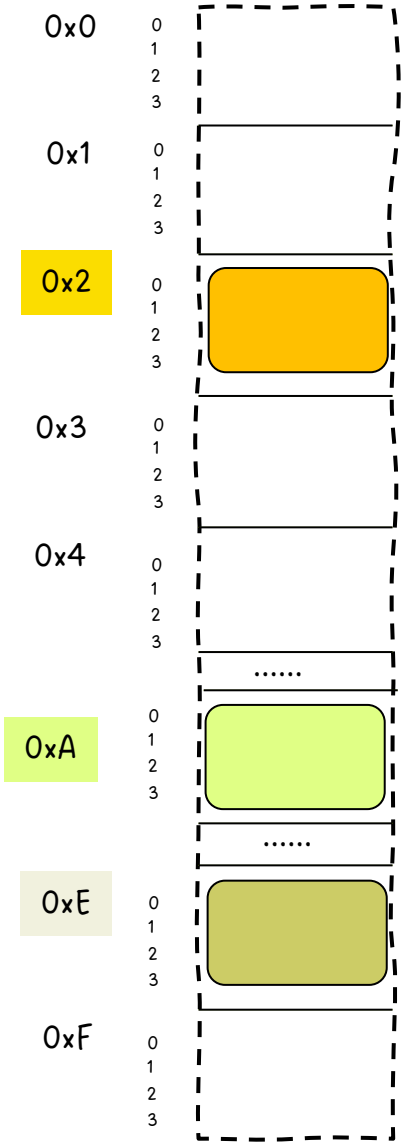
4 bits : 2 bits



Logical address: n = 2 and m = 6. Using a page size of 2^2 =4 bytes and a physical memory of 32 bytes (16 pages)

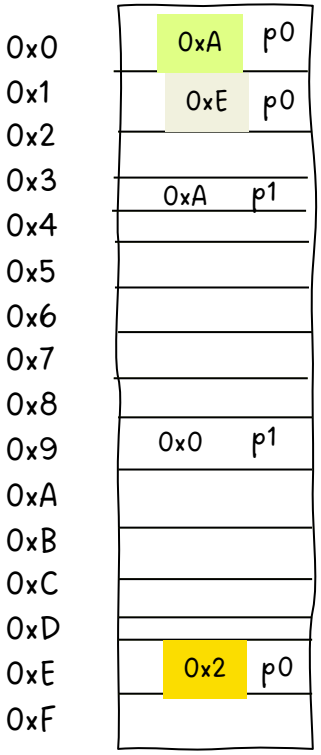
Page id (p) : offset (d)

4 bits : 2 bits



Inverted Page Table (IVT)

Frame id : Page id



When p0 runs:

Pg = 0xE, Search entire IVT for ("0xE", p0) = 0x1

Pg = 0xA, Search entire IVT for ("0xA", p0) = 0x0

Pg = 0x2, Search entire IVT for ("0x2", p0) = 0xE

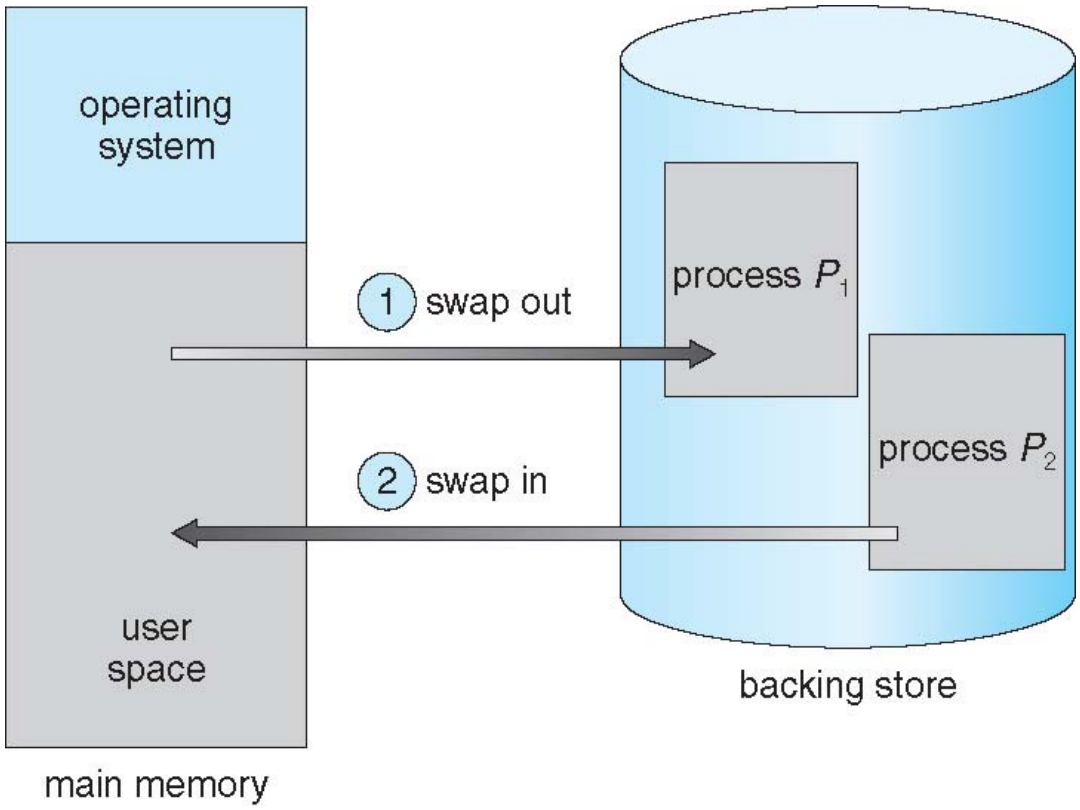
Frame id (f) : offset (d)

4 bits : 2 bits

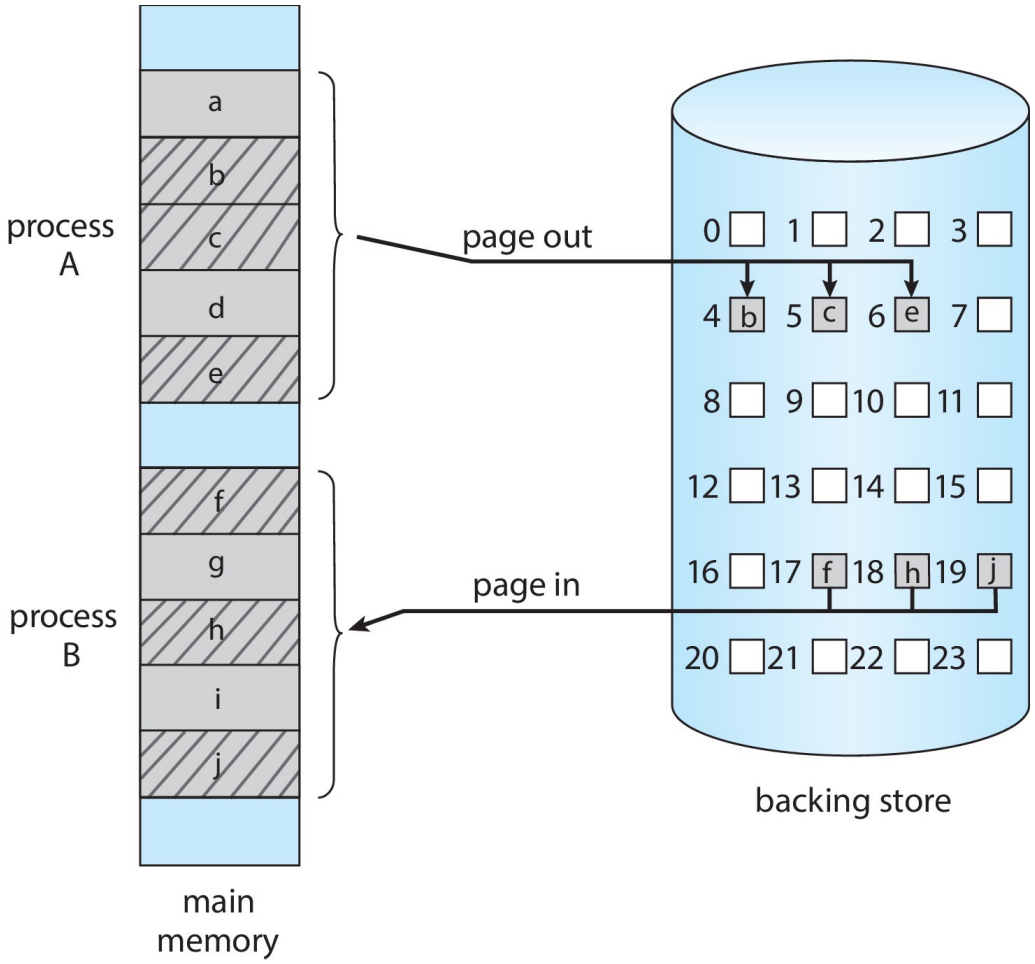
Swapping

- Standard Swapping
- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Swapping with Paging



Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

สรุป

- Memory Management
- Page Table
- TLB
- Page Table Implementations