CS222
Operating Systems

Lecture 12 (version 1)
Virtual Memory

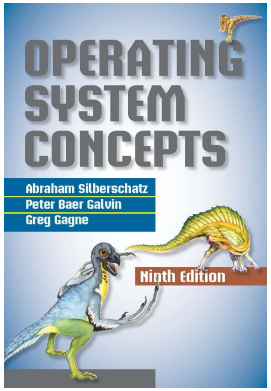
(Section 100001)

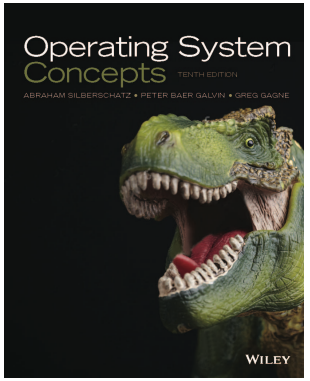
ผศ. ดร. กษิดิศ ชาญเชี่ยว

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 10 (10th edition)

  Virtual Memory

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

Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Other Considerations
- Operating-System Examples

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Design a virtual memory manager simulation in the C programming language.

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

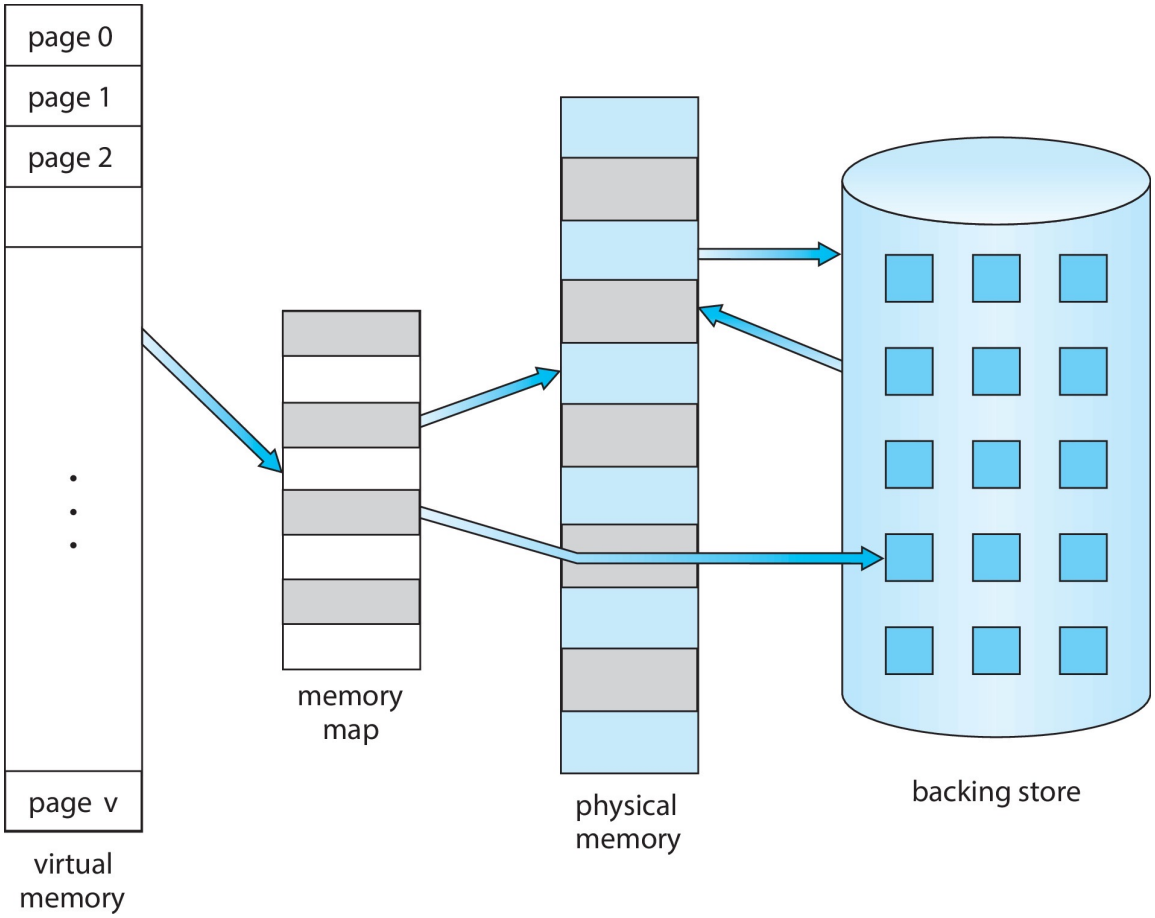
Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
    - เพราะ load หรือ swap เป็น pages ไม่ต้อง load หรือ swap ทั้งโปรแกรม

Virtual memory (Cont.)

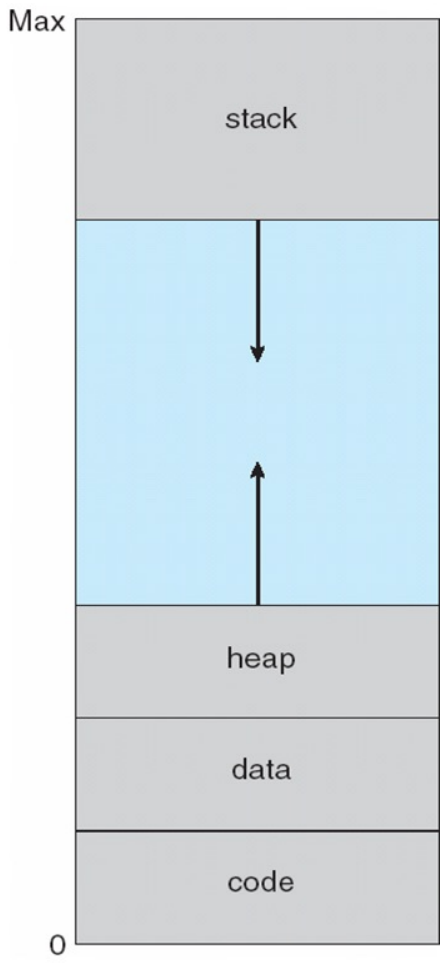
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

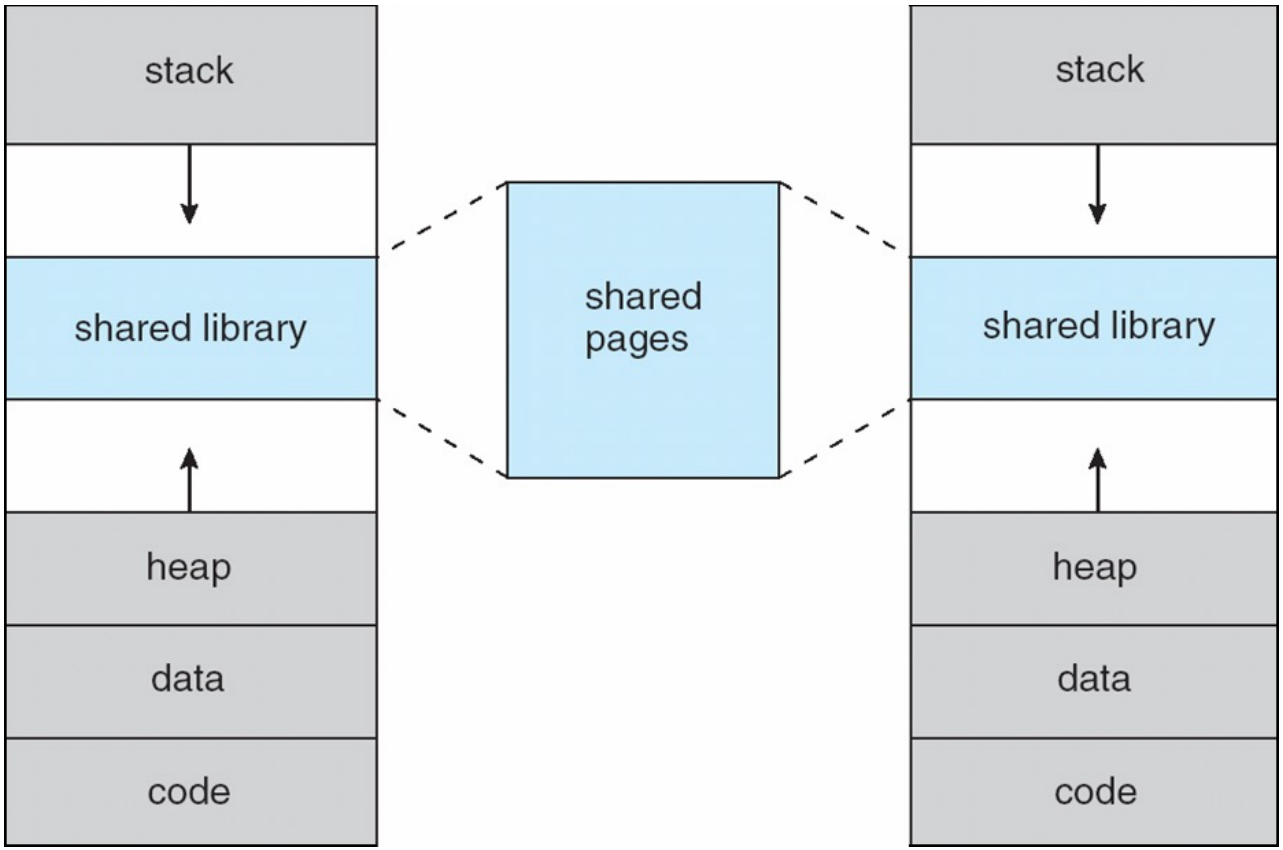


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory

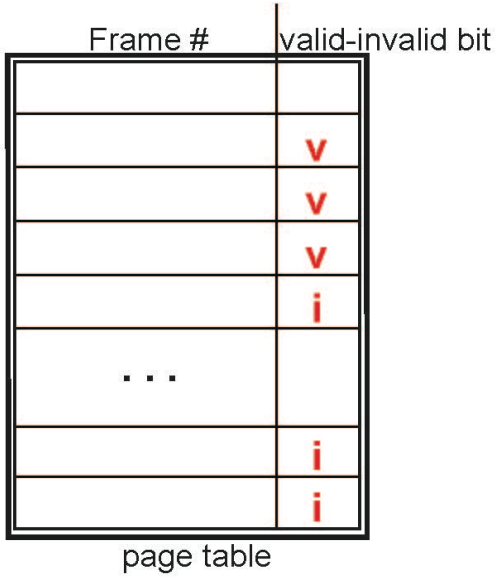


Demand Paging

- แต่ก่อนเรา bring entire process into memory at load time
- Demand Paging คือการ bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

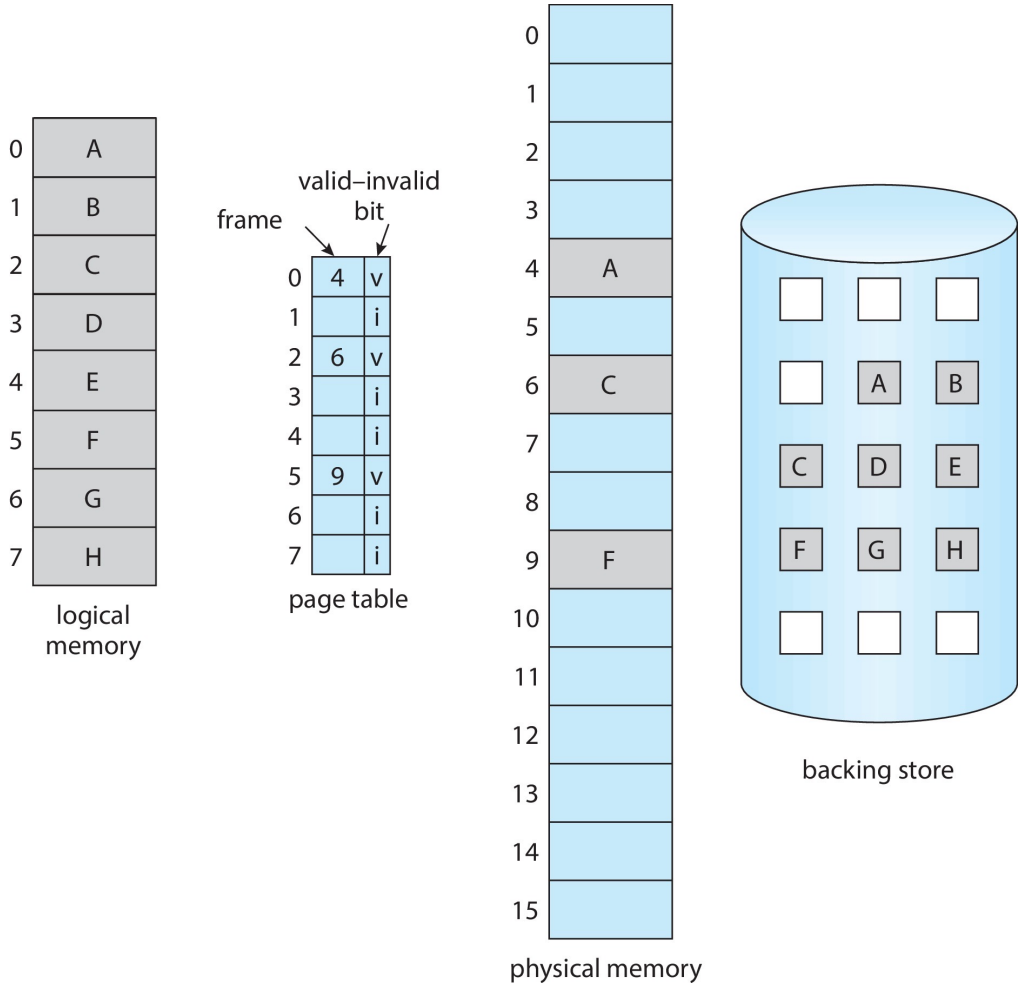
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

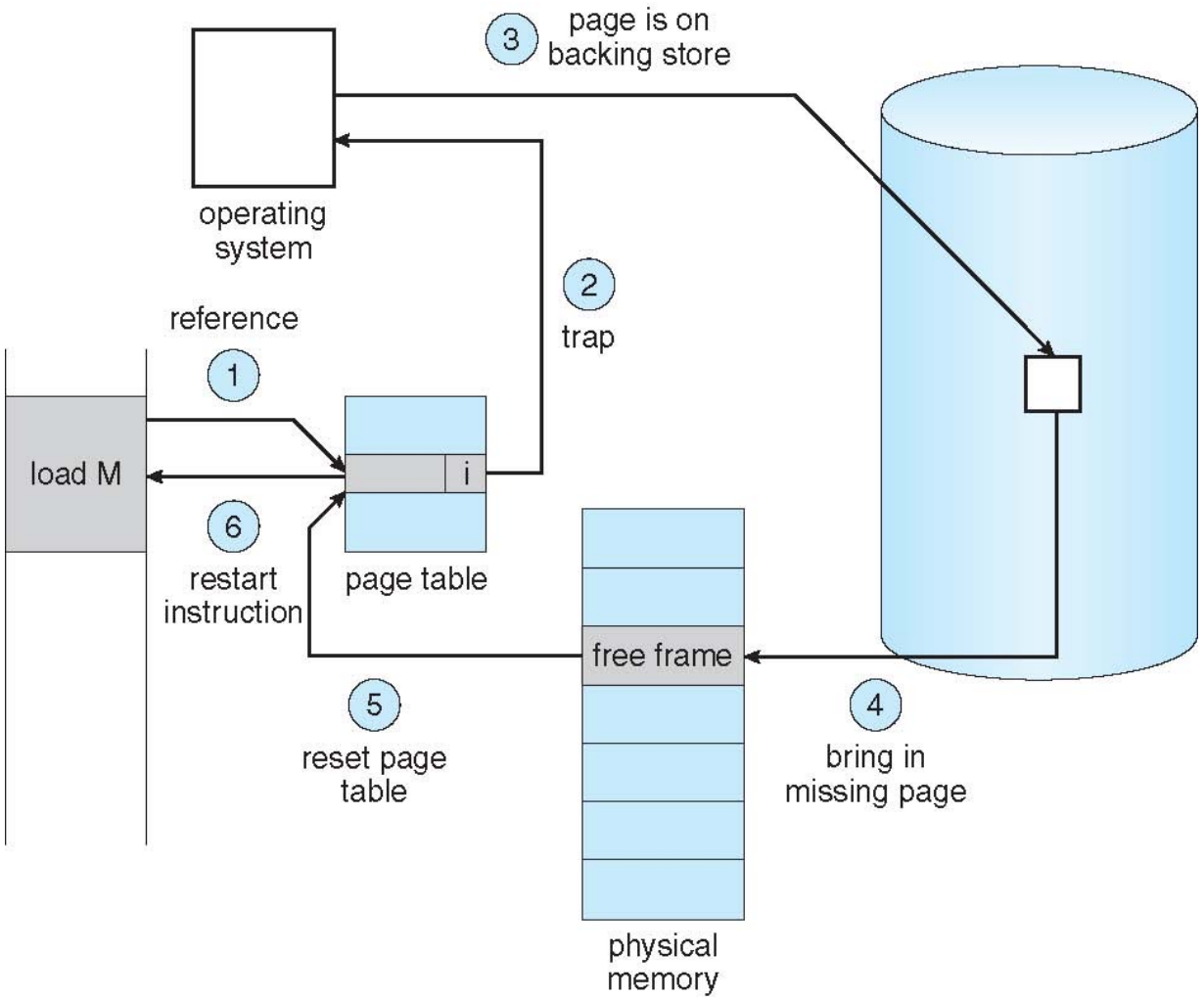
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

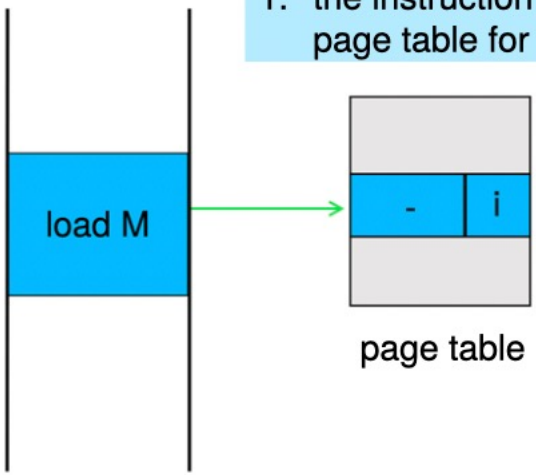
1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. ***Restart*** the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)

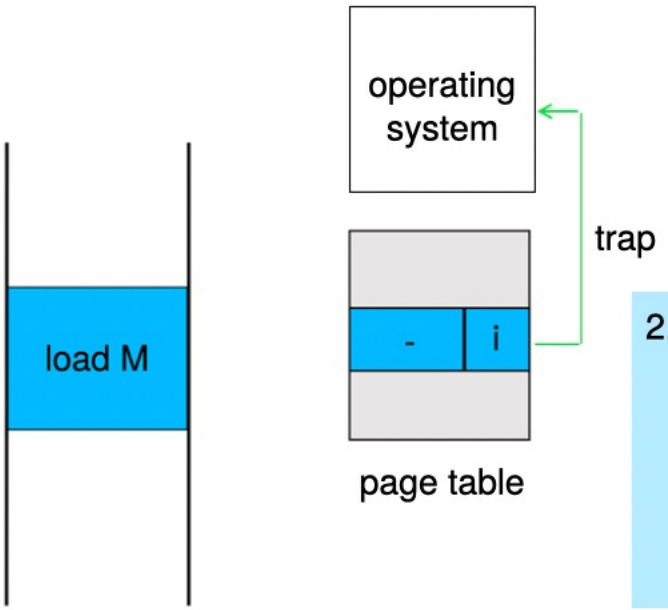


Steps in Handling a Page Fault

1. the instruction “load M” executes, accessing the page table for the address containing “M”

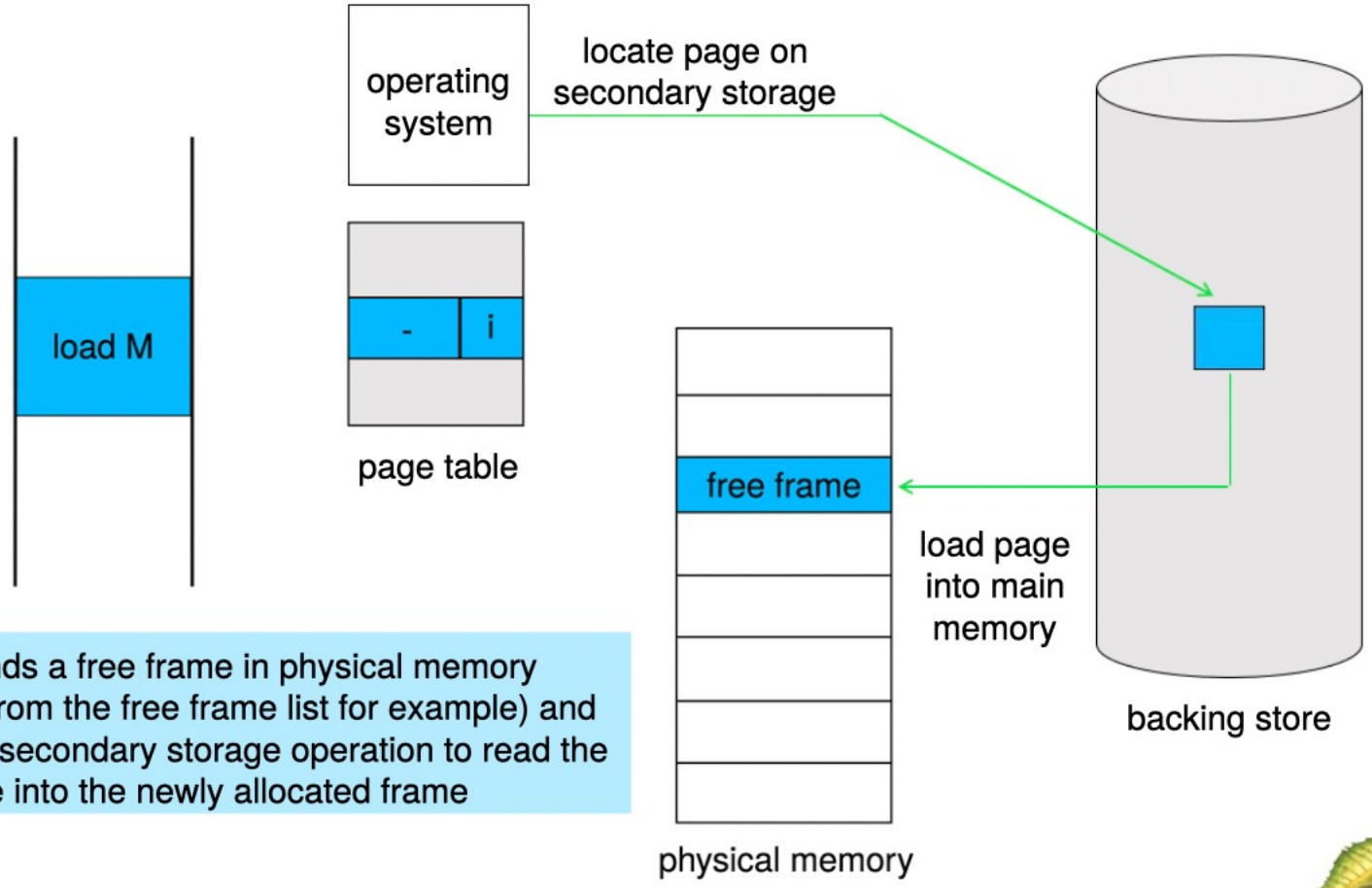


Steps in Handling a Page Fault



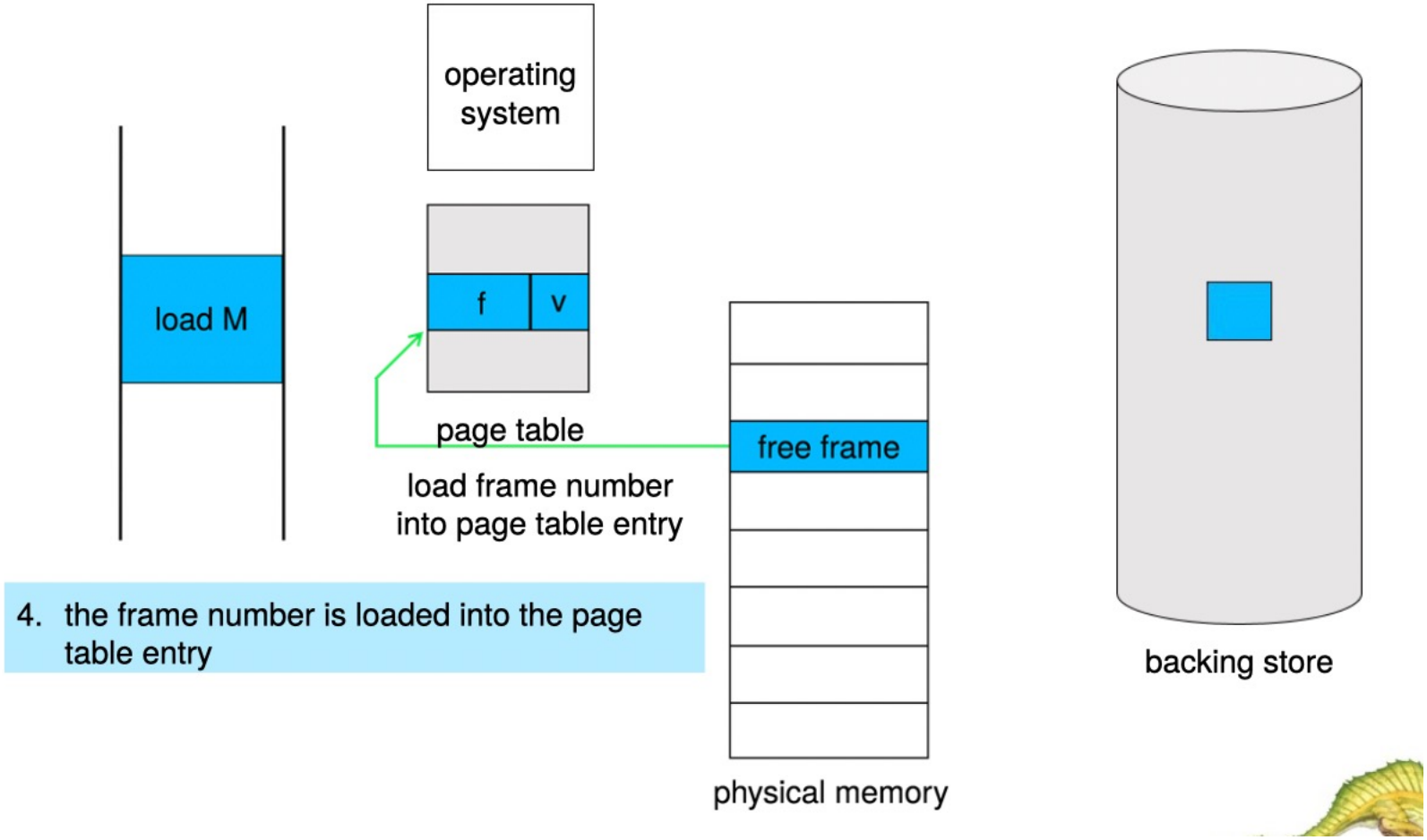
2. the page is marked as invalid, causing a page fault trap to the kernel
   - the operating system checks if the reference is legal or not
   - if it is illegal, the process is terminated
   - in this case it is valid but the page is not currently in a frame in physical memory

Steps in Handling a Page Fault



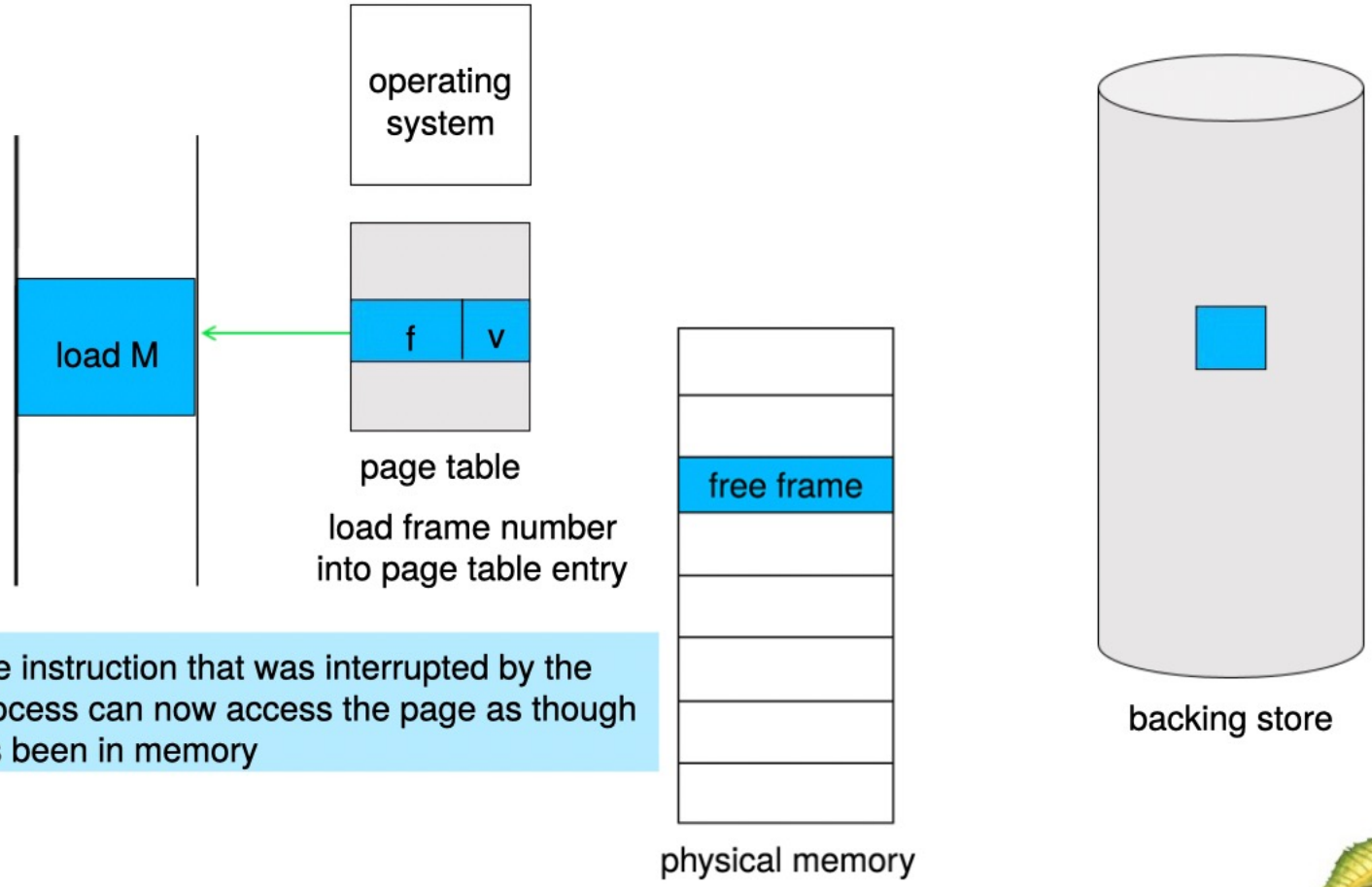
3. the kernel finds a free frame in physical memory (taking one from the free frame list for example) and schedules a secondary storage operation to read the desired page into the newly allocated frame

Steps in Handling a Page Fault



4. the frame number is loaded into the page table entry

Steps in Handling a Page Fault

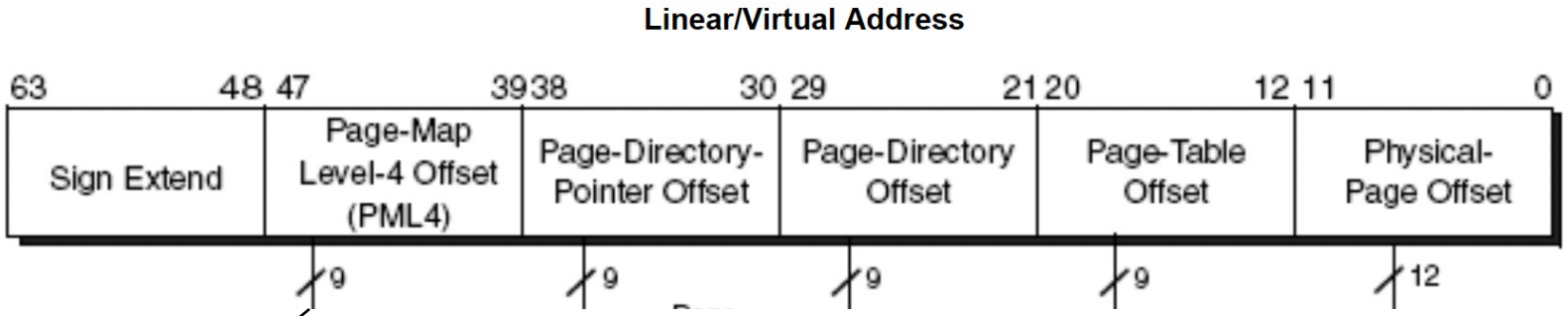


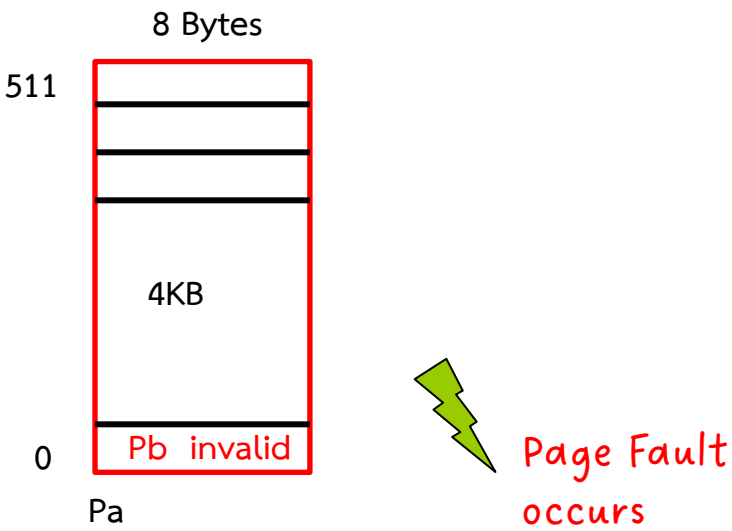
5. we restart the instruction that was interrupted by the trap – the process can now access the page as though it had always been in memory

จำลองการเกิด Page Fault บน x86_64 4-level Heirarhical Page Table

VA = 0x 10 = 0b 0001 0000

000_8 000_8 000_8 000_8 0020_8





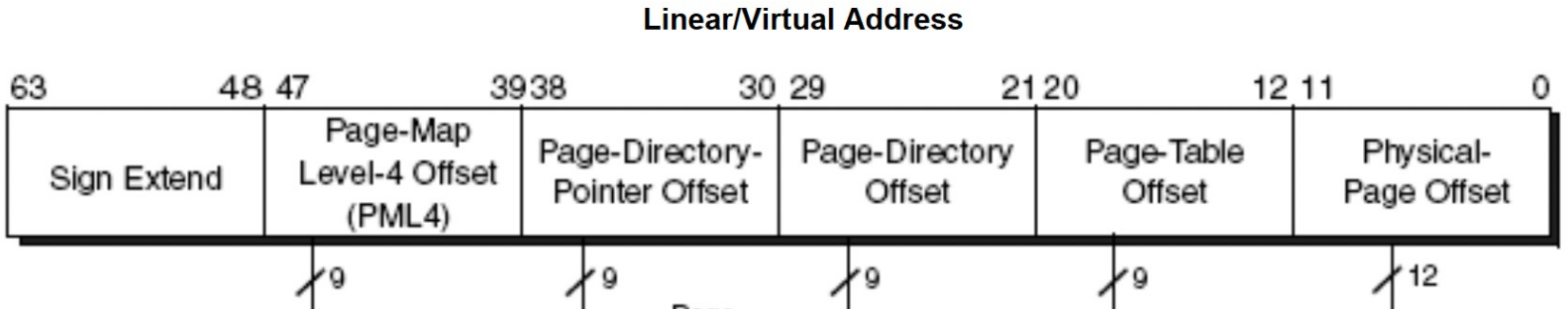
Page Fault occurs

CR3 = Pa

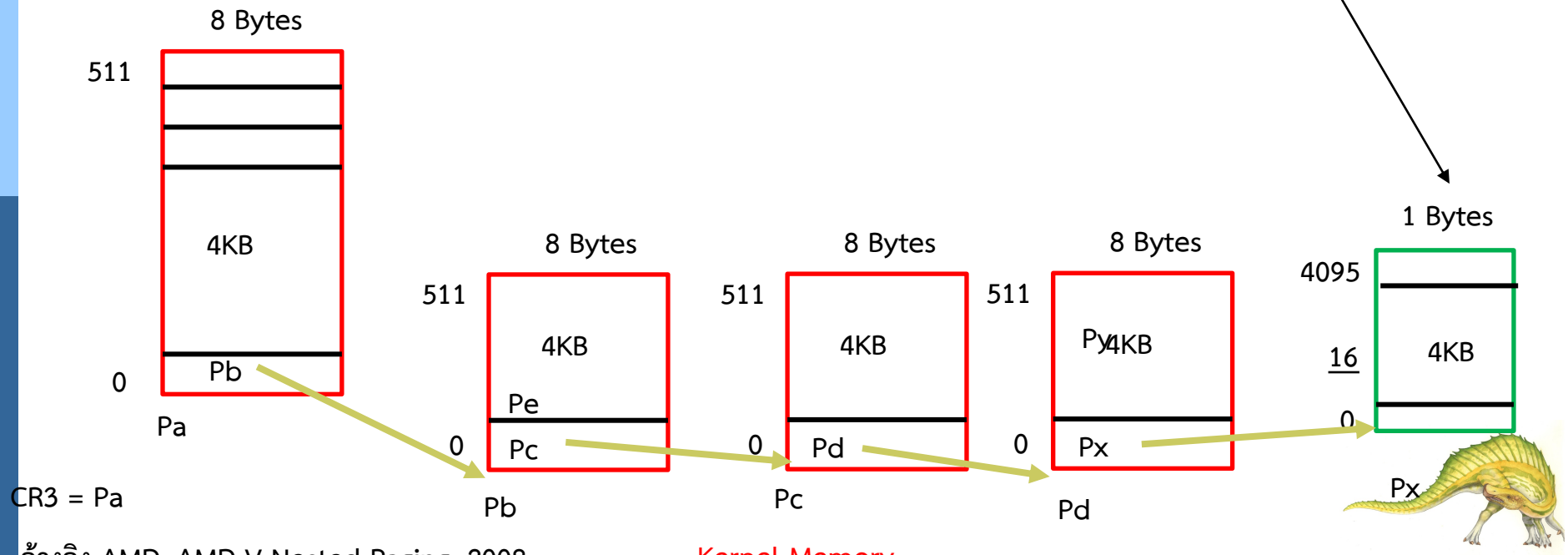
อ้างอิง AMD, AMD-V Nested Paging, 2008

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

000_8 000_8 000_8 000_8 0020_8



User Process Memory

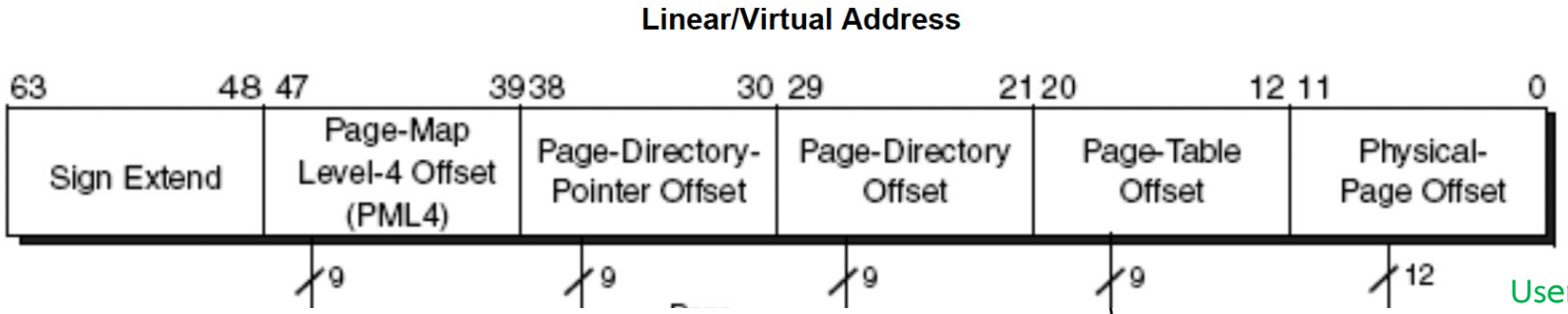


อ้างอิง AMD, AMD-V Nested Paging, 2008

Kernel Memory

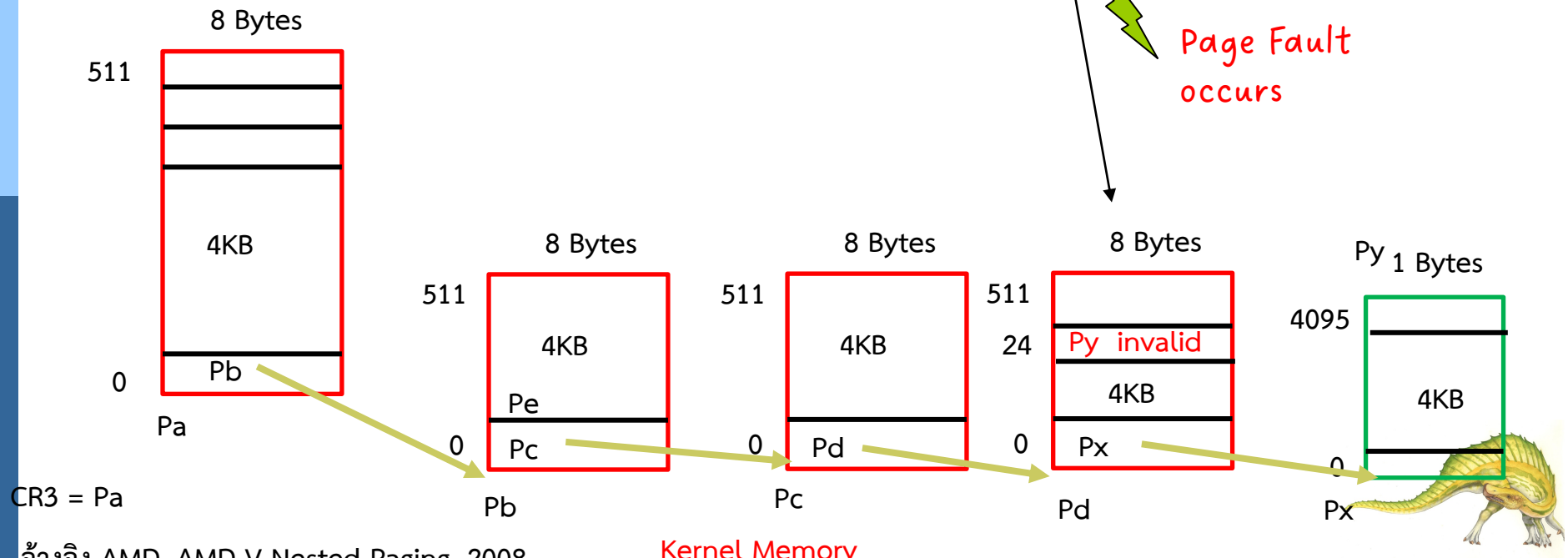
VA = 0x 3000 = 0b 0011 0000 0000 0000 = 0o 30000

000_8 000_8 000_8 003_8 0000_8



User Process Memory

Page Fault occurs

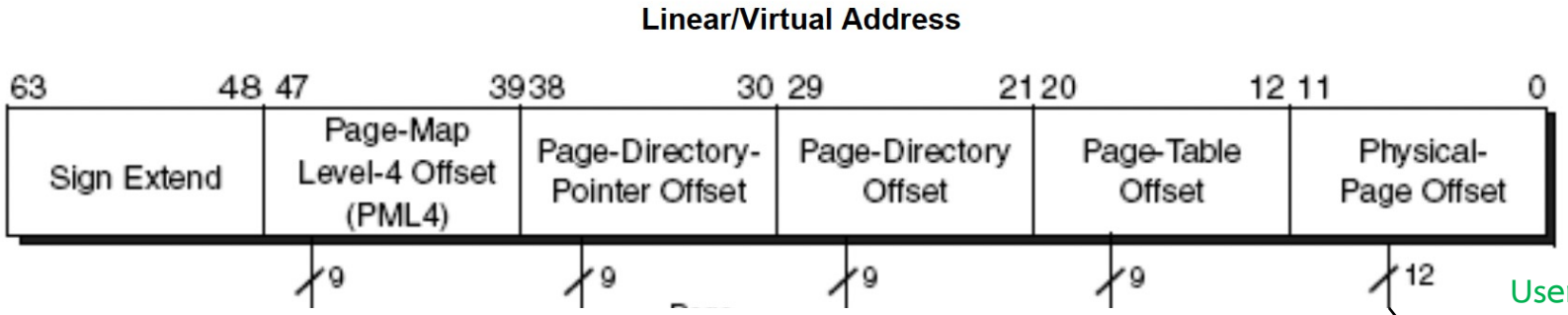


อ้างอิง AMD, AMD-V Nested Paging, 2008

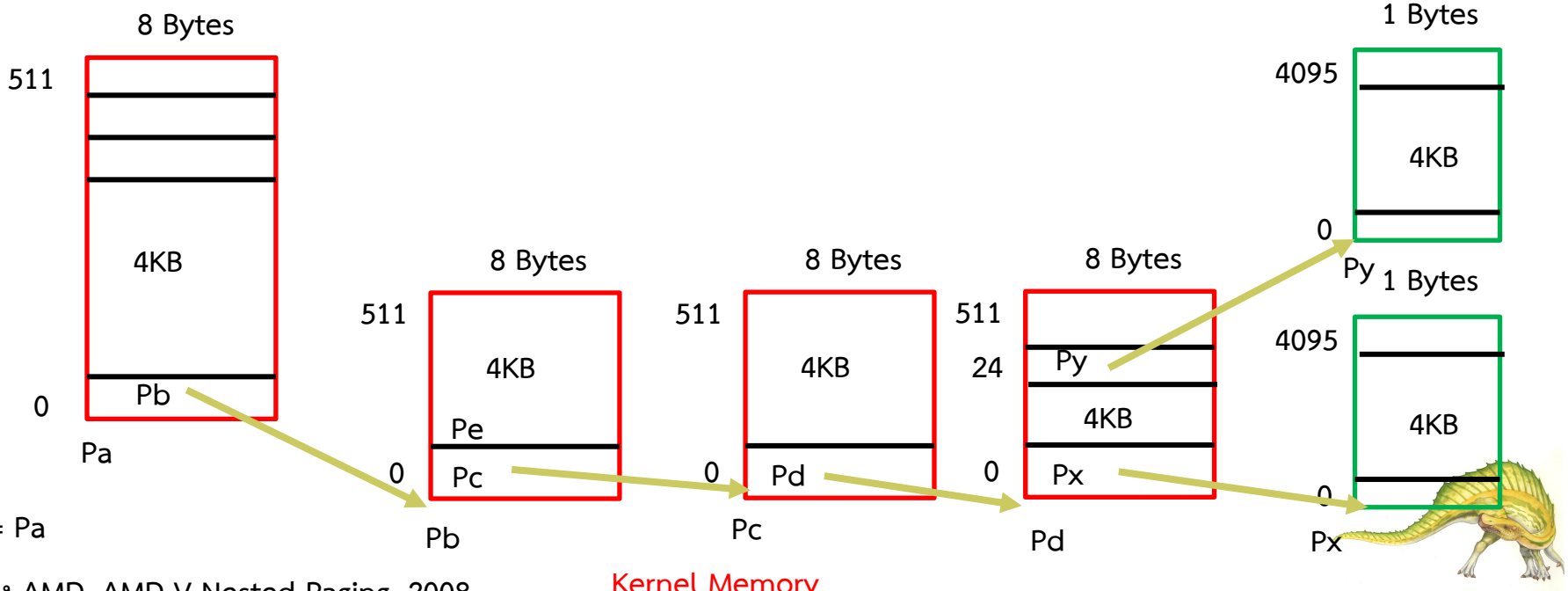
Kernel Memory

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

000_8 000_8 000_8 003_8 0000_8



User Process Memory



Kernel Memory

อ้างอิง AMD, AMD-V Nested Paging, 2008

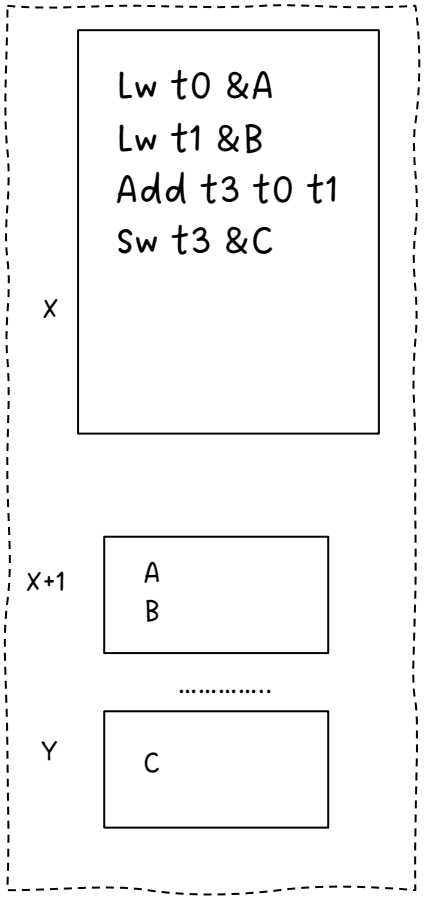
Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
  - **locality of reference** คือการที่โปรเซสมีแนวโน้มที่จะใช้งานหน่วยความจำที่อยู่ใกล้กัน และมีแนวโน้มที่จะใช้คำสั่งที่ใช้บ่อยซ้ำๆกันในโปรแกรม

Aspects of Demand Paging

- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart
    - Instruction restart หมายถึงการรันคำสั่งที่ทำให้เกิด Page Fault Trap หลังจากการ Page Fault handler ทำงานเสร็จแล้ว
    - เมื่อ Page Fault Handler ทำงานเสร็จ และ OS เลือก Process ที่เกิด Page Fault เข้าใช้งานซีพียูอีกครั้งหนึ่ง
    - เมื่อซีพียูประมวลผล Process มันจะต้อง Restart Instruction ที่ทำให้เกิด Page Fault ไม่ไช่รันคำสั่งถัดไป เหมือนการเกิด Trap/Interrupt แบบอื่น
    - จำเป็นต้องมี Hardware Support มาช่วยกรณีนี้

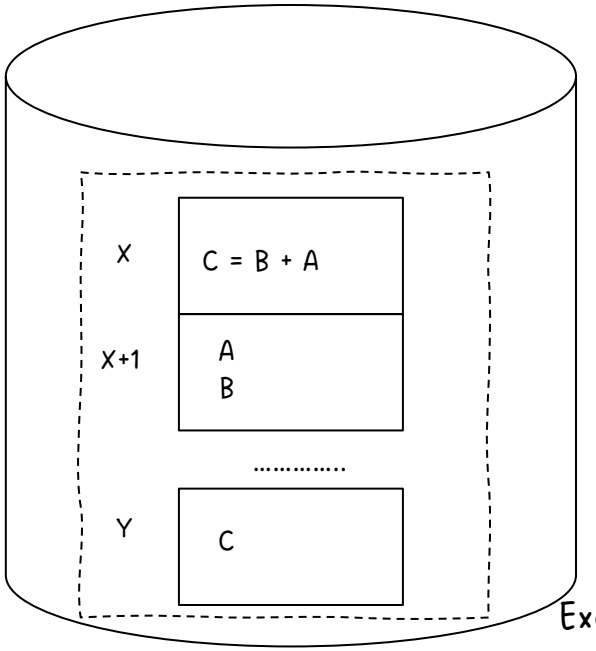
Pure Demand Paging



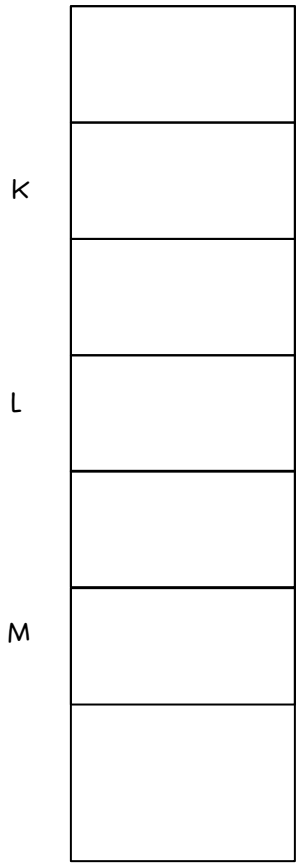
Virtual address space

Page Table

p	f.	v/i
x		I
x+1		I
y		i

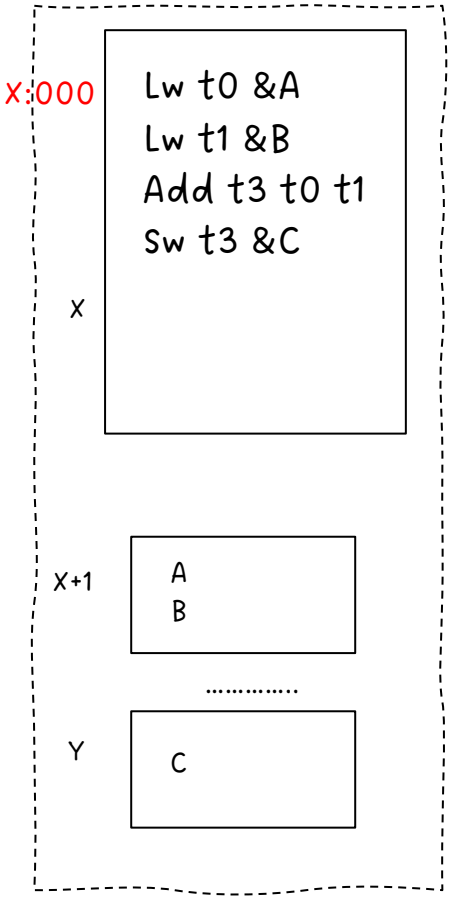


Executable file



Physical address space

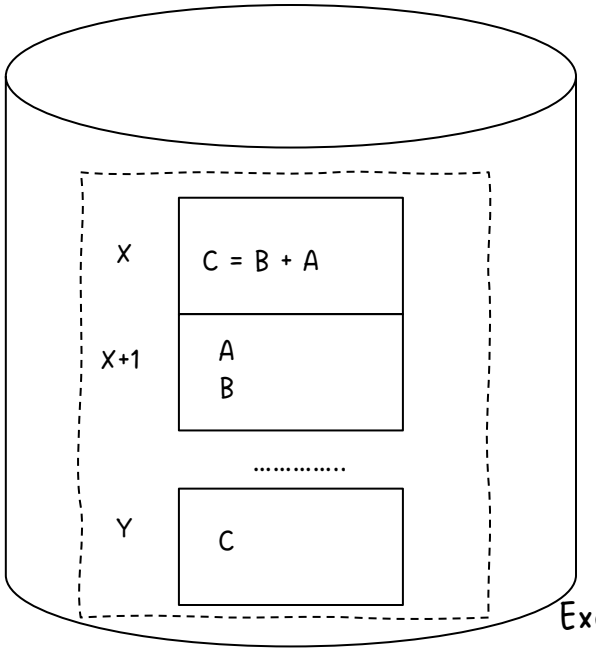
(1) เริ่มประมวลผล x:000 แต่พบ I ใน PT เกิด page fault



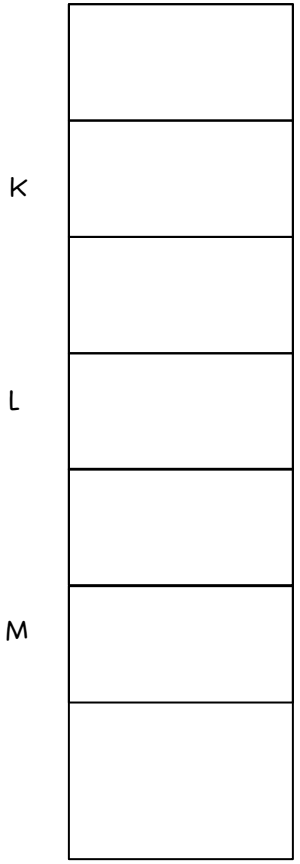
Virtual address space

Page Table

p	f.	v/i
x		I
x+1		I
y		i

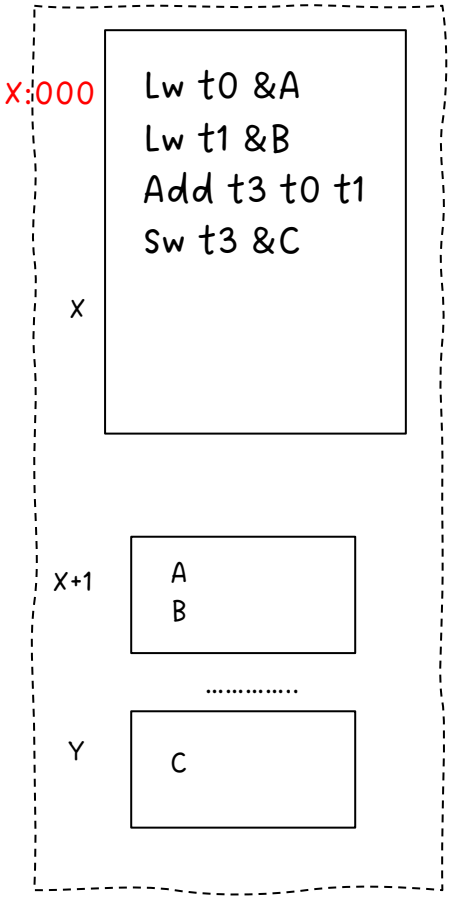


Executable file



Physical address space

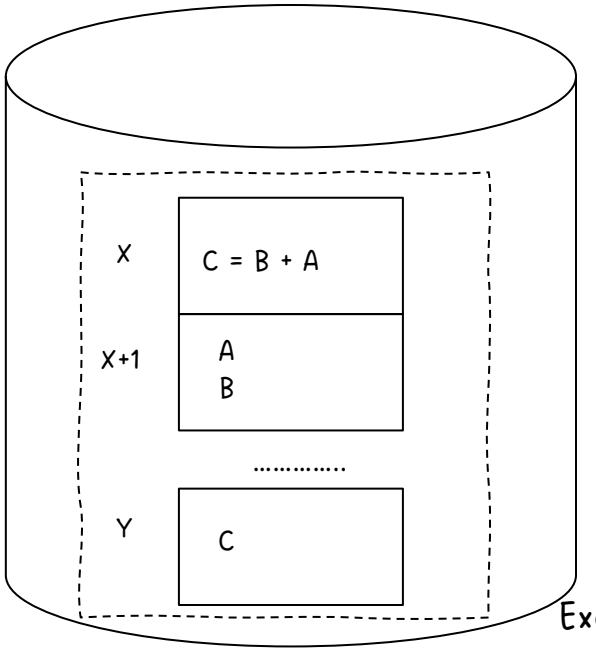
(1) เริ่มประมวลผล x:000 แต่พบ I ใน PT เกิด page fault
(2) อ่าน X จาก storage มาไว้ที่ frame K update PT



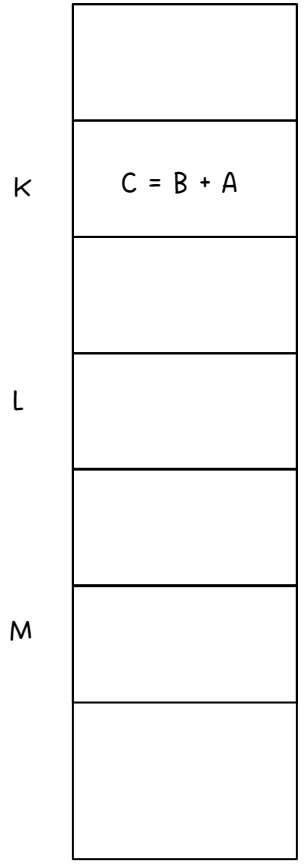
Virtual address space

Page Table

p	f.	v/i
x	k	v
x+1		I
y		i



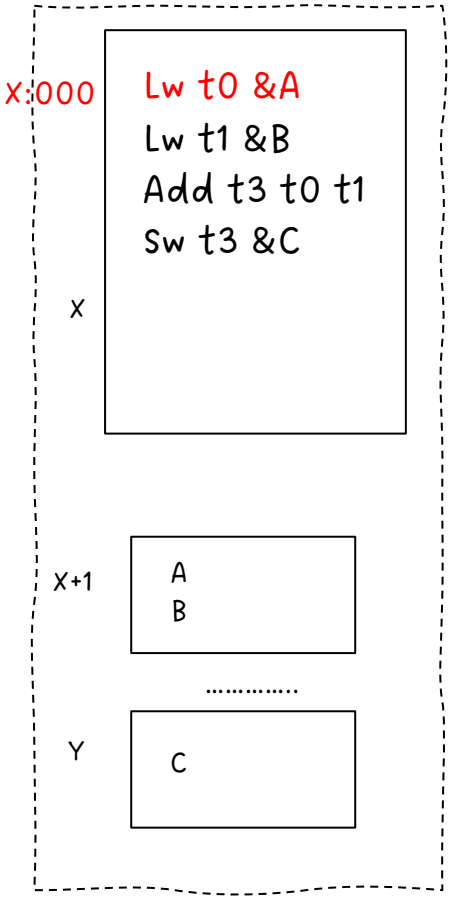
Executable file



Physical address space

(1) Restat คำสั่งที่ x:000

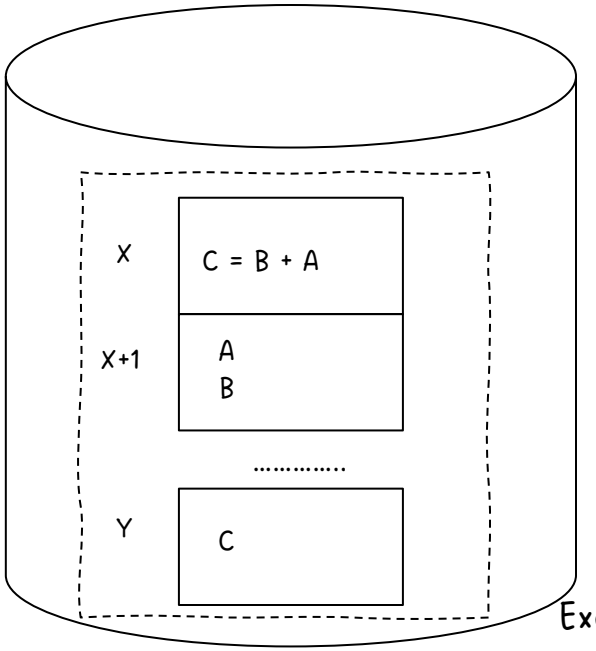
(2) เกิด page fault เพราะ x+1 ไม่อยู่ใน phy mem



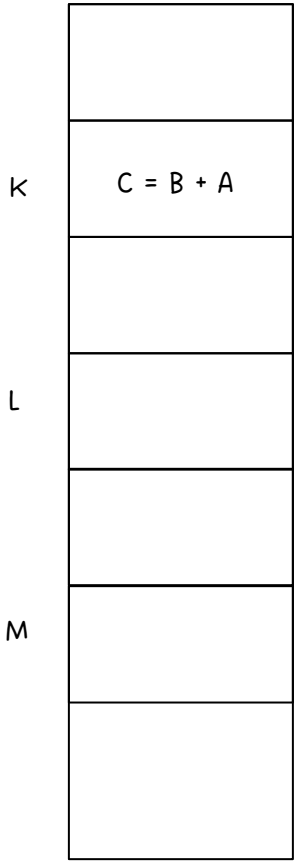
Virtual address space

Page Table

p	f.	v/i
x	k	v
x+1		I
y		i



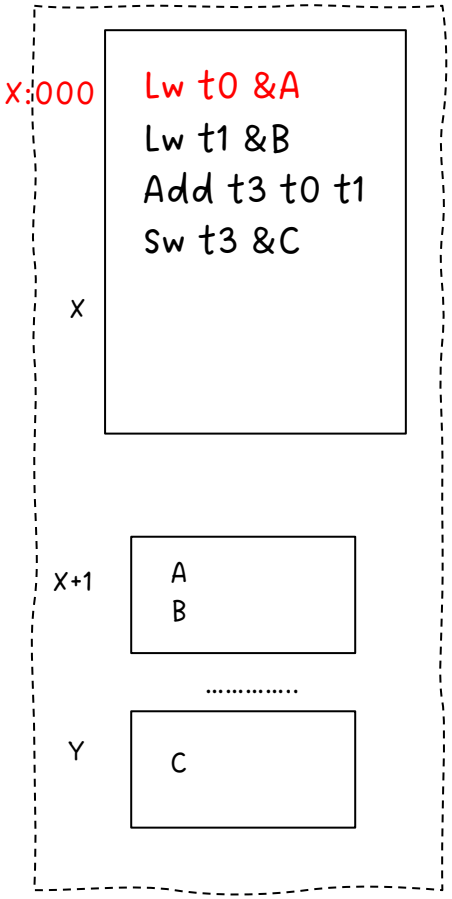
Executable file



Physical address space

(1) Restat คำสั่งที่ x:000

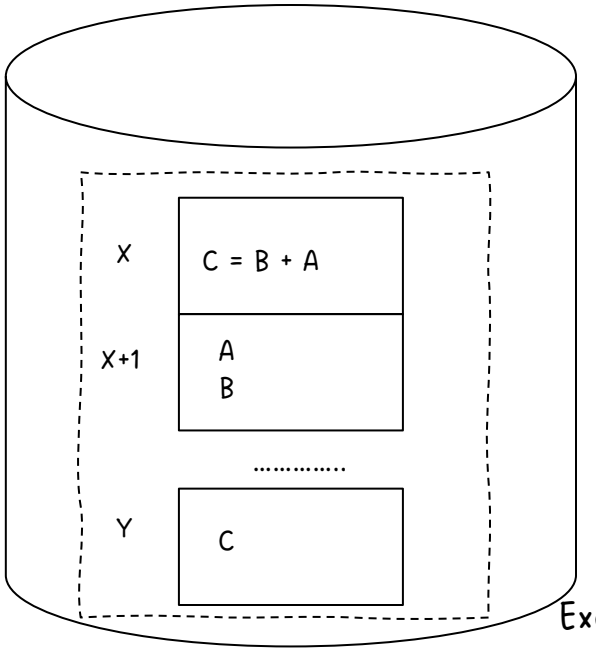
(2) เกิด page fault เพราะ x+1 ไม่อยู่ใน phy mem ทำให้ OS เอา x+1 เข้ามาที่ L



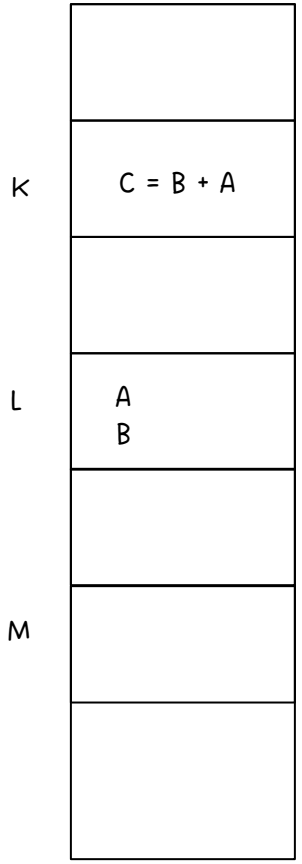
Virtual address space

Page Table

p	f.	v/i
x	k	v
x+1	L	v
y		i



Executable file



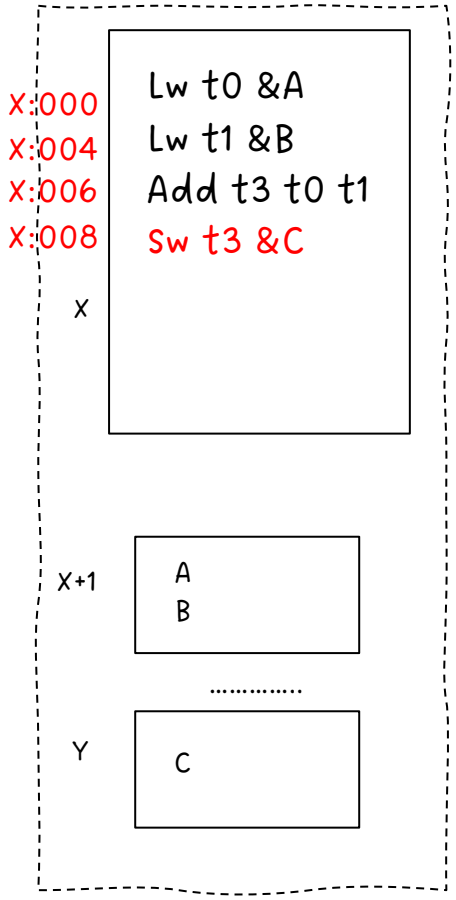
Physical address space

(1) Restat คำสั่งที่ x:000

(2) Run คำสั่ง x:004

(3) Run คำสั่ง x:006

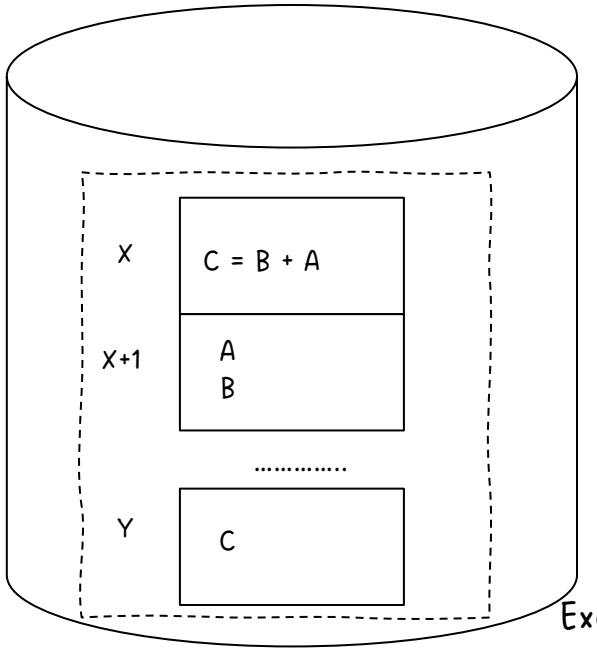
(4) Run คำสั่ง x:008 (เกิด page fault เพราะ ?)



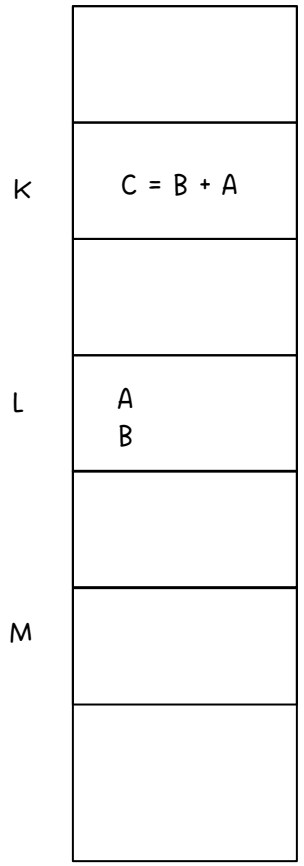
Virtual address space

Page Table

p	f.	v/i
x	k	v
x+1	L	v
y		i



Executable file



Physical address space

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Stages in Demand Paging – Worse Case

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    a) Wait in a queue for this device until the read request is serviced
    b) Wait for the device seek and/or latency time
    c) Begin the transfer of the page to a free frame

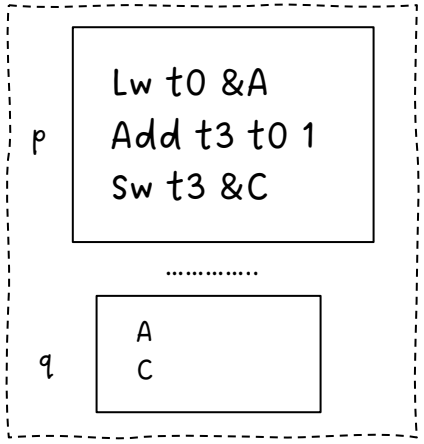
Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

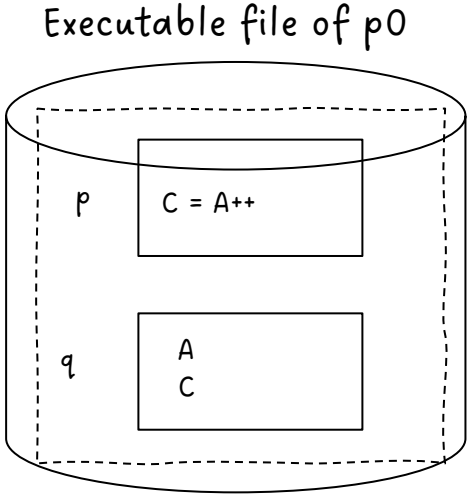
เงื่อนไขตัวอย่างการประมวลผล (1)

สมมุติว่า:

1. ในระบบมีโปรเซส p0 และ p1
2. CPU ประมวลผล p0 อยู่ดังนั้น p0 มี State คือ running และ p1 อยู่ใน Ready State
3. p0 มี page "p" อยู่ใน physical mem

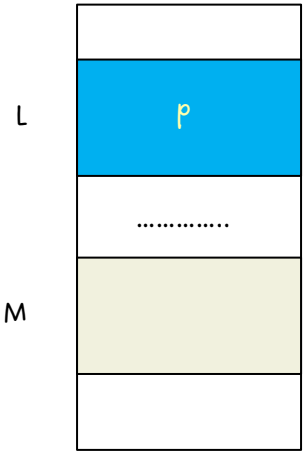


Virtual address space



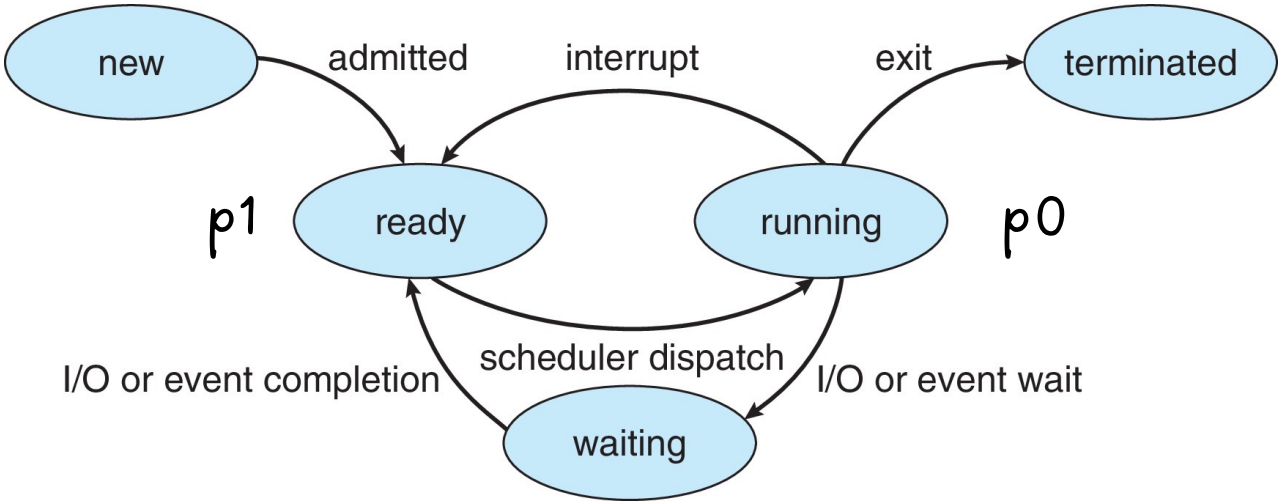
Page Table of p0

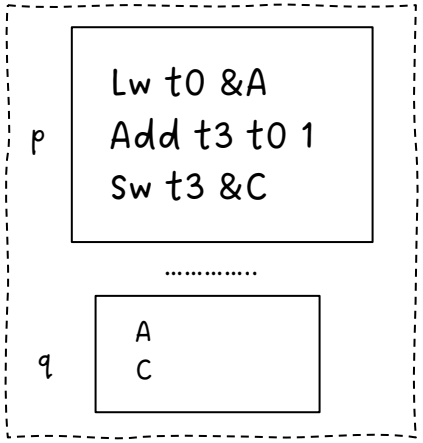
p	f.	v/i
p	L	v
q		i



Physical address space

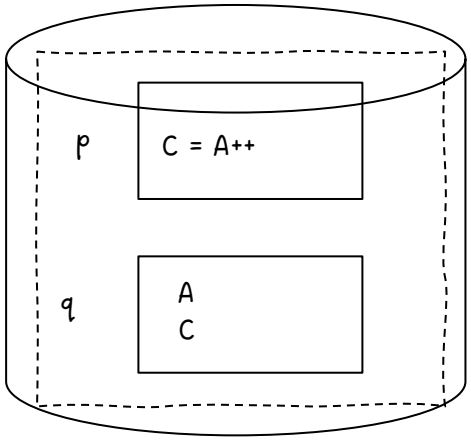
เงื่อนไขตัวอย่างการประมวลผล (2)





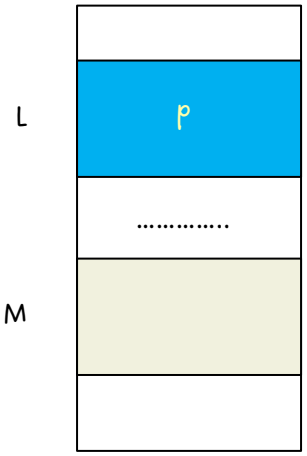
Virtual address space

Executable file of p0



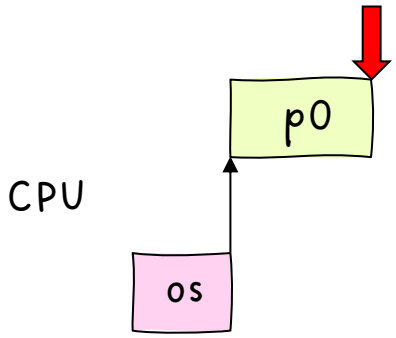
Page Table of p0

p	f.	v/i
p	L	v
q		i

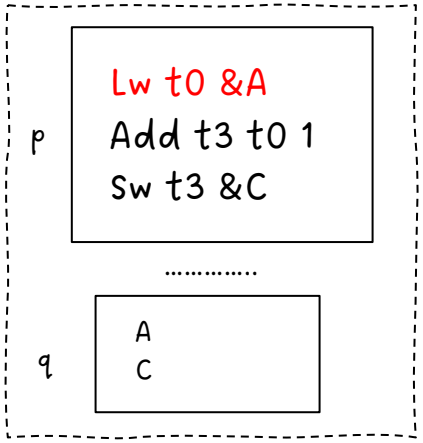


Physical address space

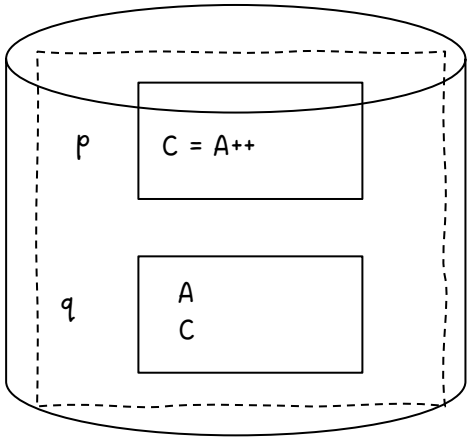
เงื่อนไขตัวอย่างการประมวลผล (3)



I/O

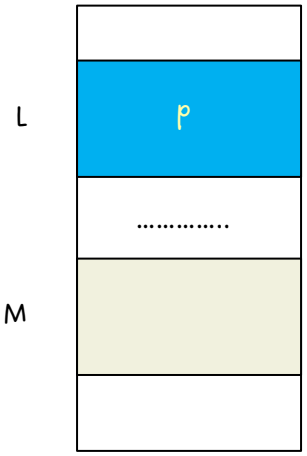


Virtual address space



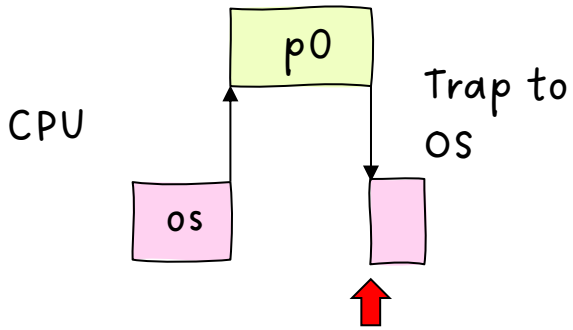
Page Table of p0

p	f.	v/i
p	L	v
q		i

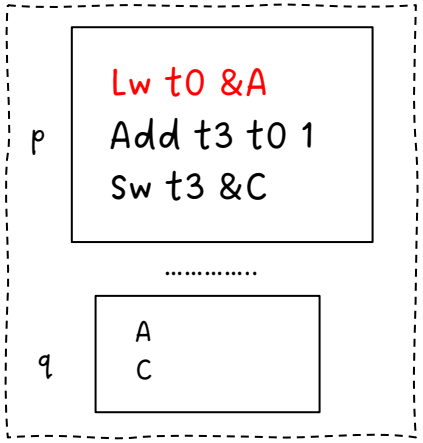


Physical address space

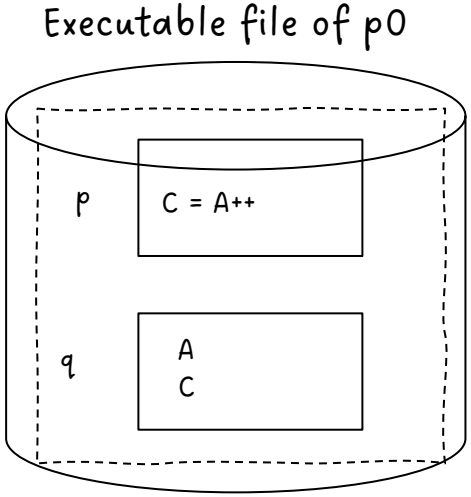
1.Trap to the operating system



I/O

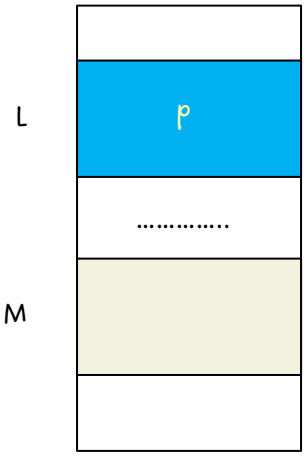


Virtual address space



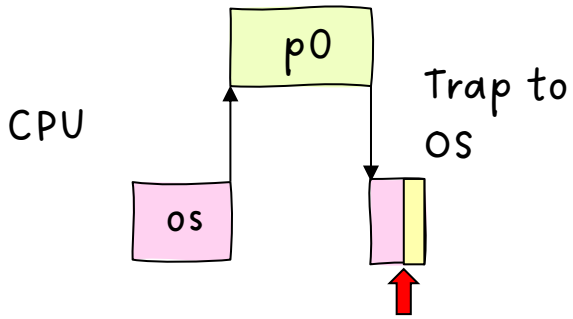
Page Table of p0

p	f.	v/i
p	L	v
q		i

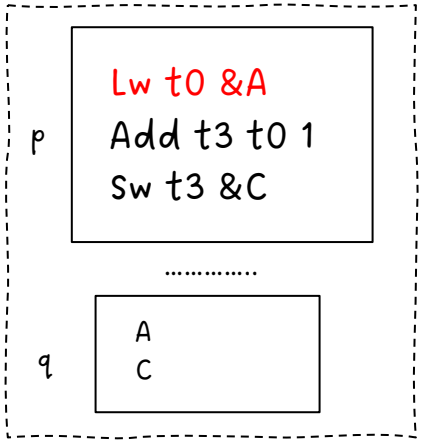


Physical address space

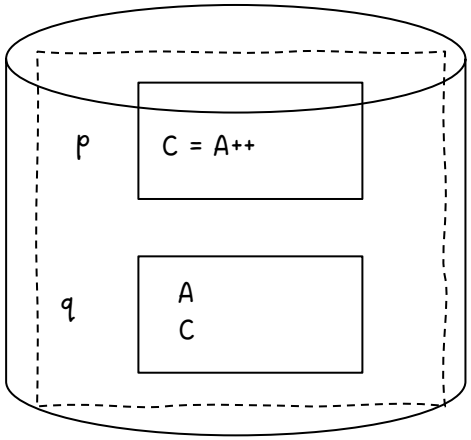
2. Save the user registers and process state



I/O

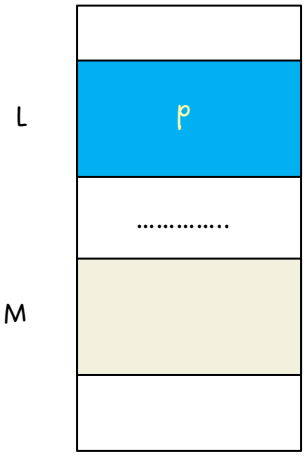


Virtual address space



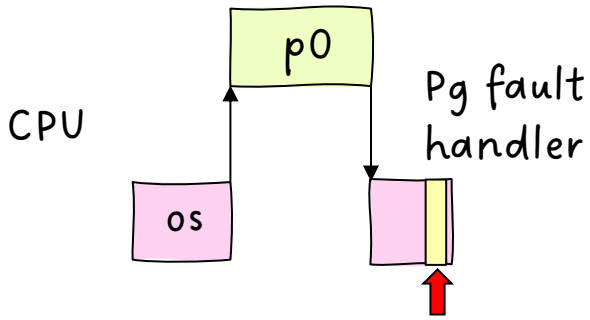
Page Table of p0

p	f.	v/i
p	L	v
q		i

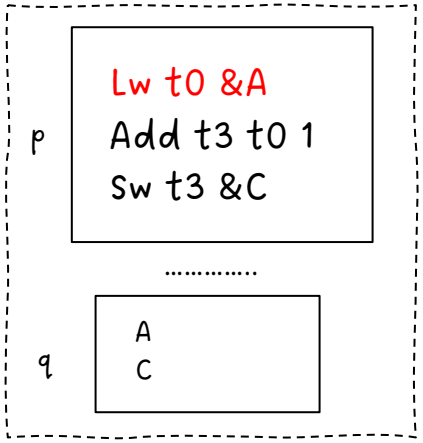


Physical address space

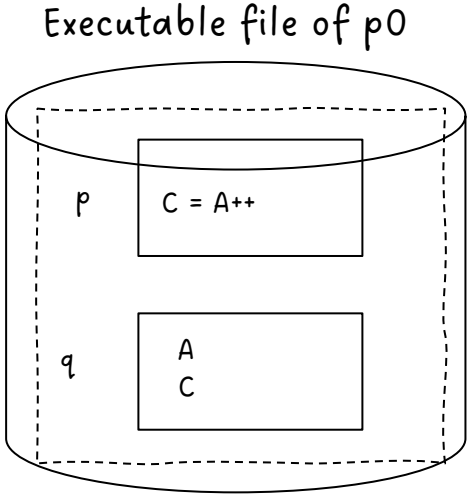
3. Determine that the interrupt was a page fault



I/O

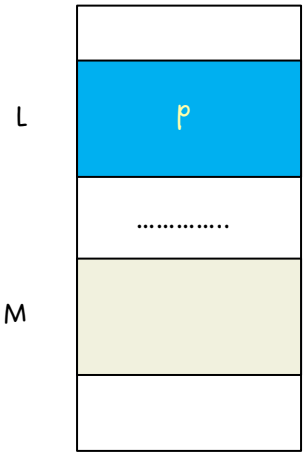


Virtual address space



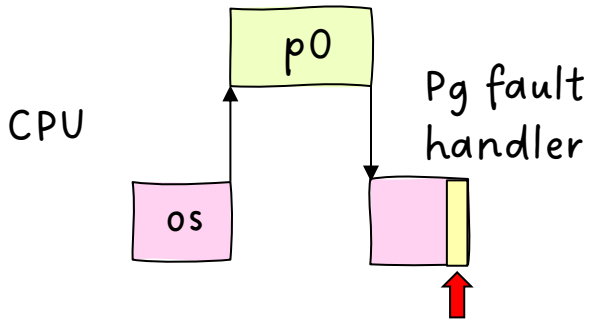
Page Table of p0

p	f.	v/i
p	L	v
q		i

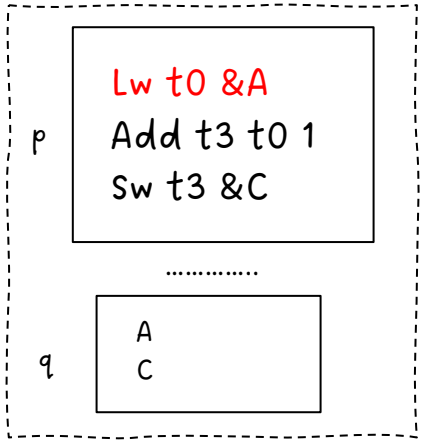


Physical address space

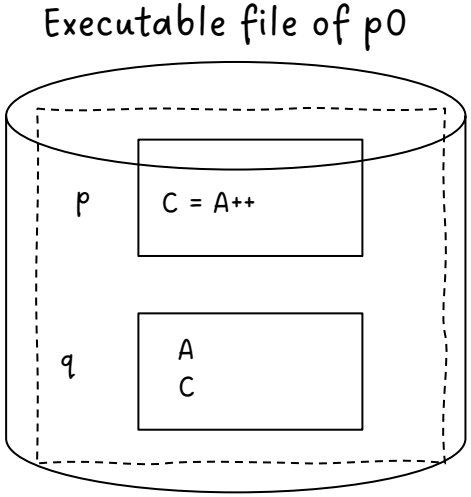
4. Check that the page reference was legal and determine the location of the page on the disk



I/O

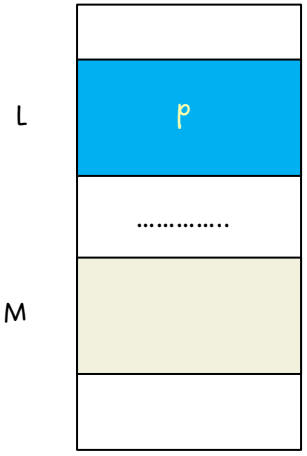


Virtual address space



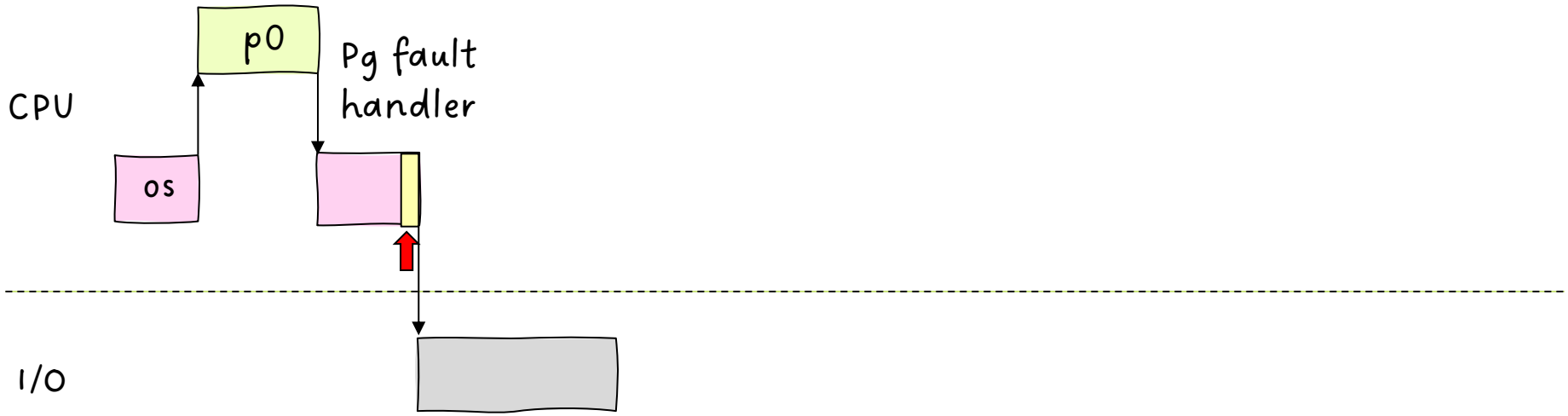
Page Table of p0

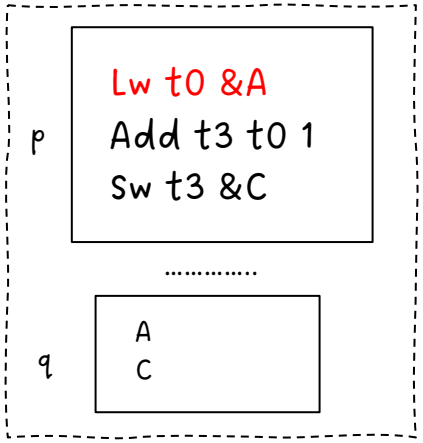
p	f.	v/i
p	L	v
q		i



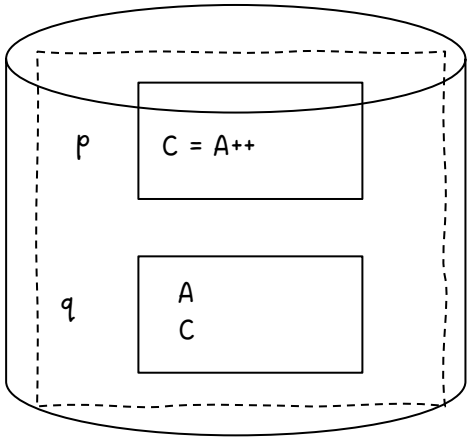
Physical address space

5. Issue a read from the disk to a free frame



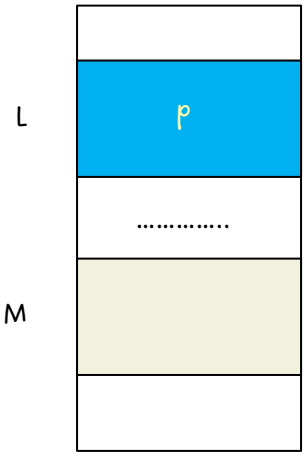


Virtual address space



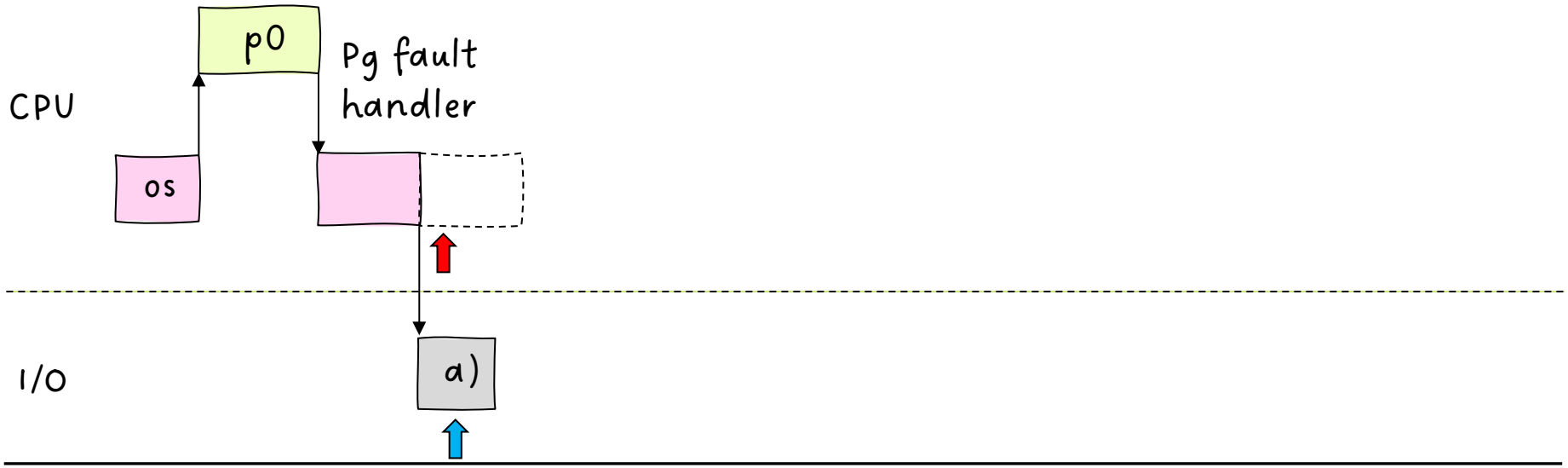
Page Table of p0

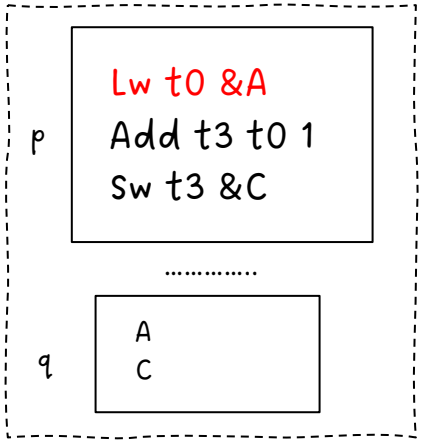
p	f.	v/i
p	L	v
q		i



Physical address space

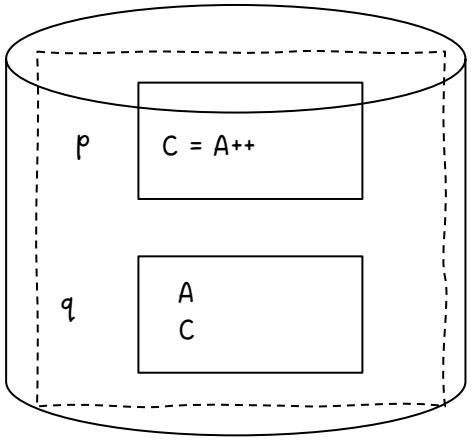
a) Wait in a queue for this device until the read request is serviced





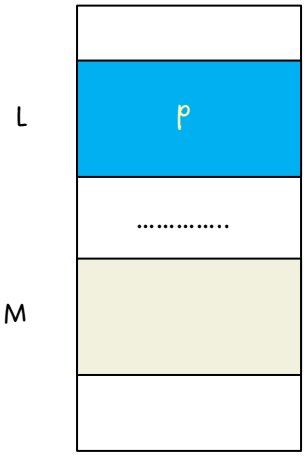
Virtual address space

Executable file of p0



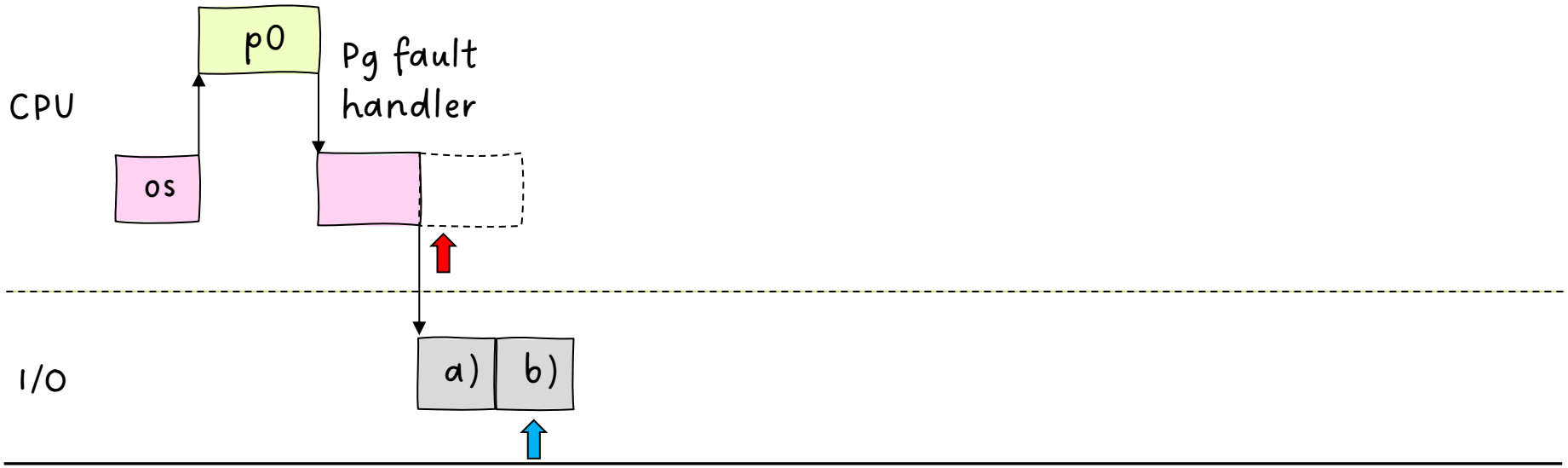
Page Table of p0

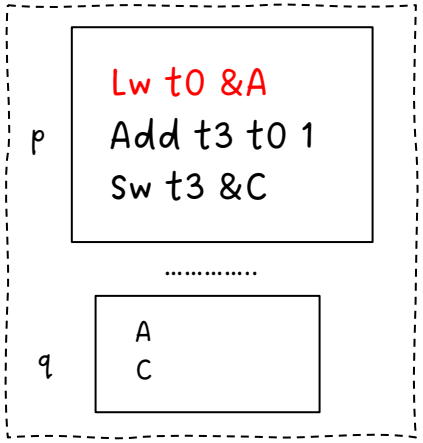
p	f.	v/i
p	L	v
q		i



Physical address space

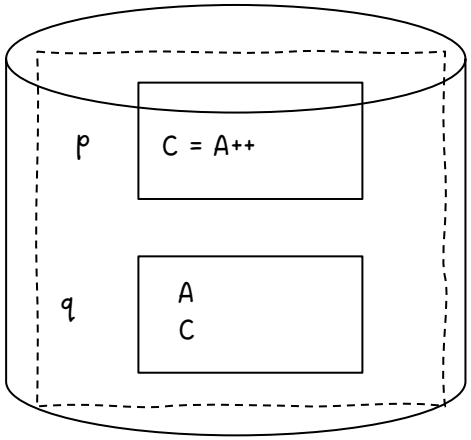
b) Wait for the device seek and/or latency time





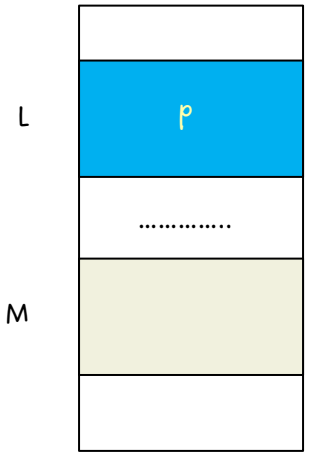
Virtual address space

Executable file of p0



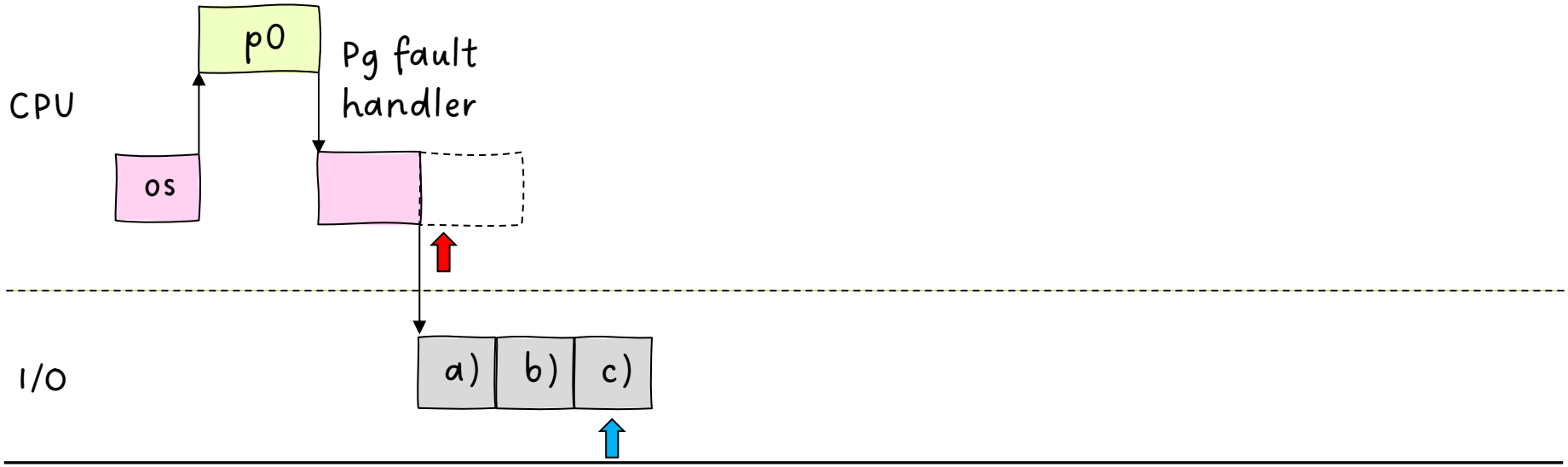
Page Table of p0

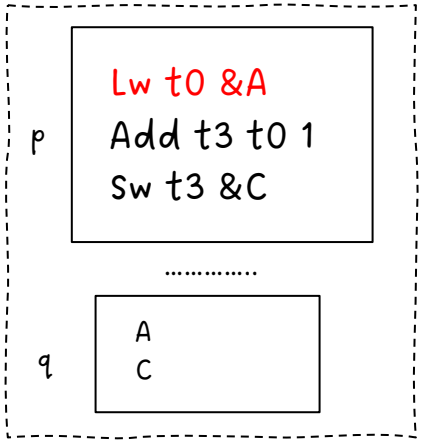
p	f.	v/i
p	L	v
q		i



Physical address space

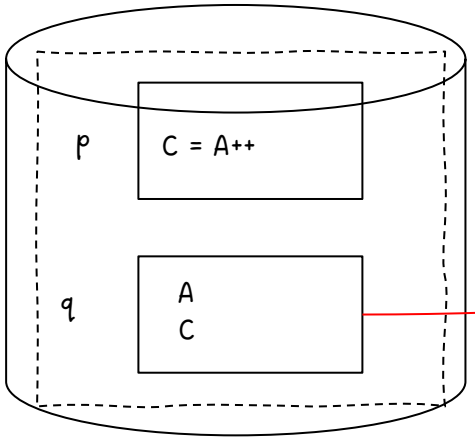
c) Begin the transfer of the page to a free frame (DMA)





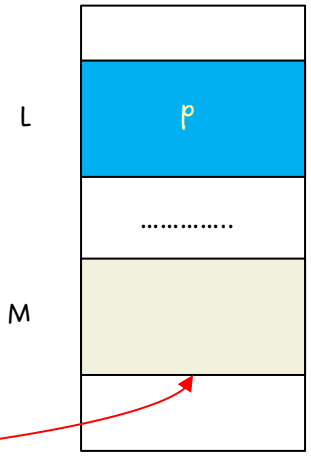
Virtual address space

Executable file of p0



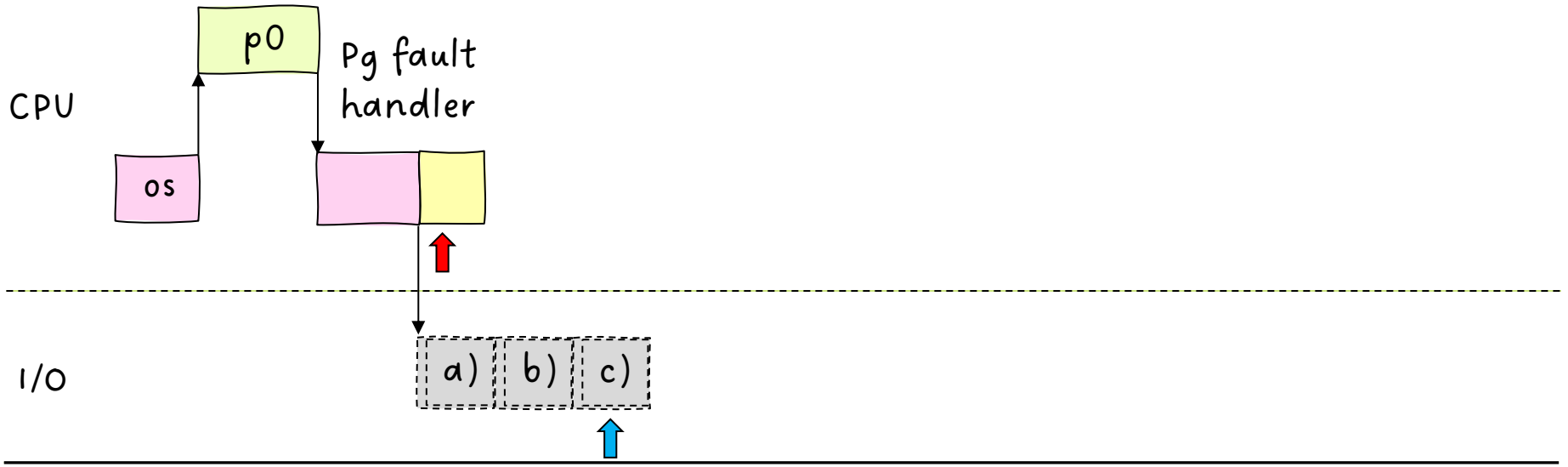
Page Table of p0

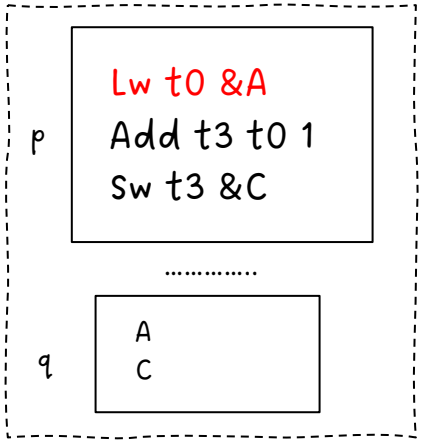
p	f.	v/i
p	L	v
q		i



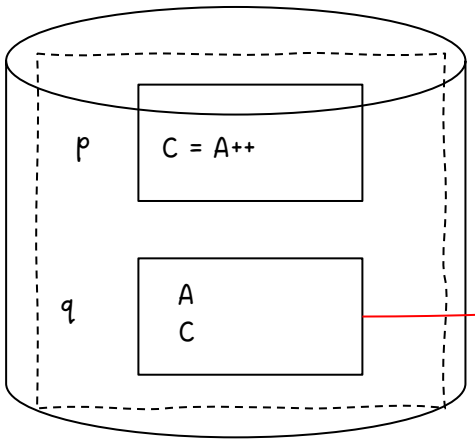
Physical address space

6. While waiting, allocate the CPU to some other user



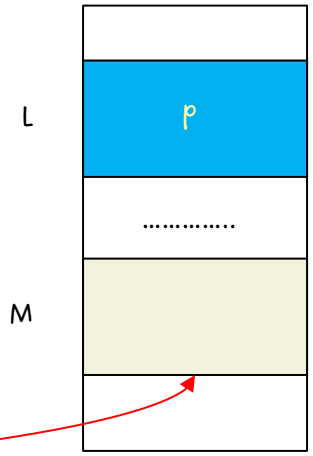


Virtual address space



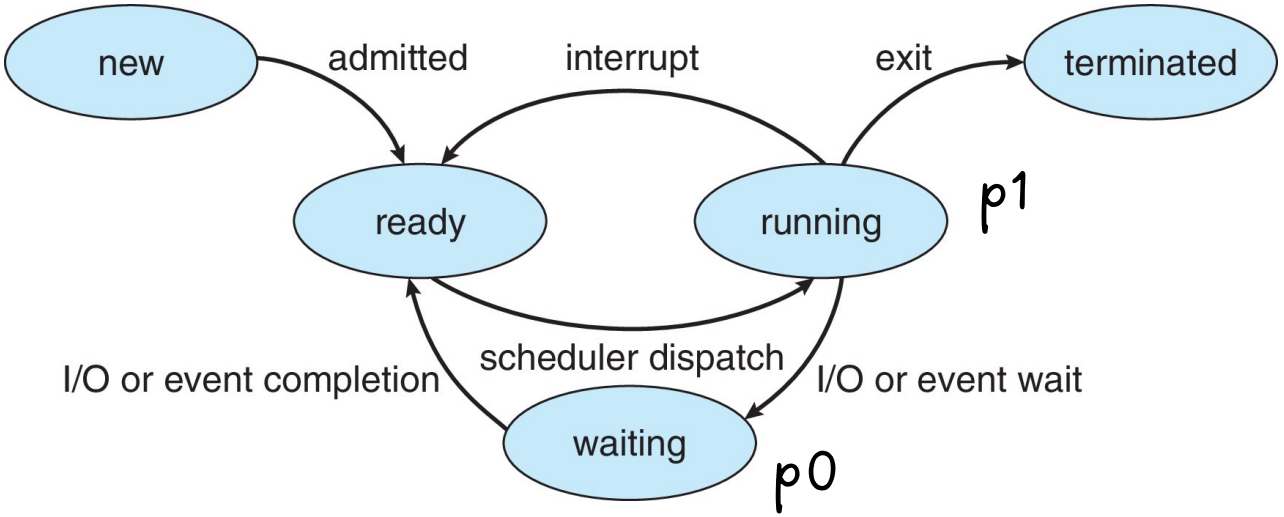
Page Table of p0

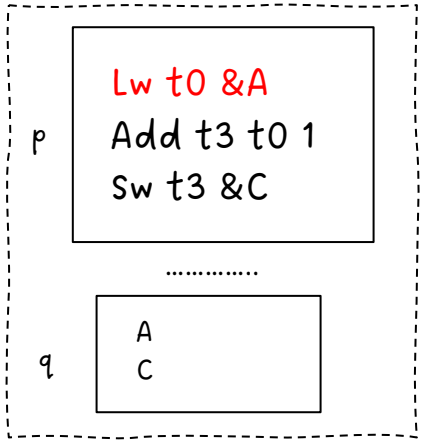
p	f.	v/i
p	L	v
q		i



Physical address space

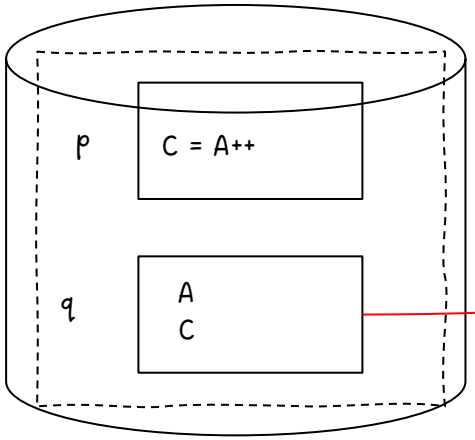
6. While waiting, allocate the CPU to some other user





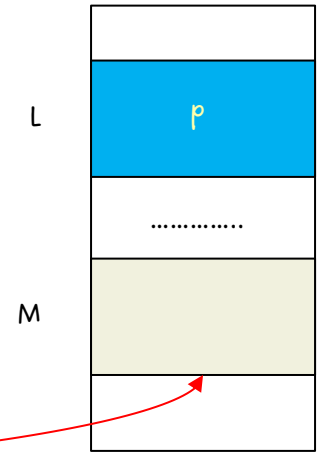
Virtual address space

Executable file of p0



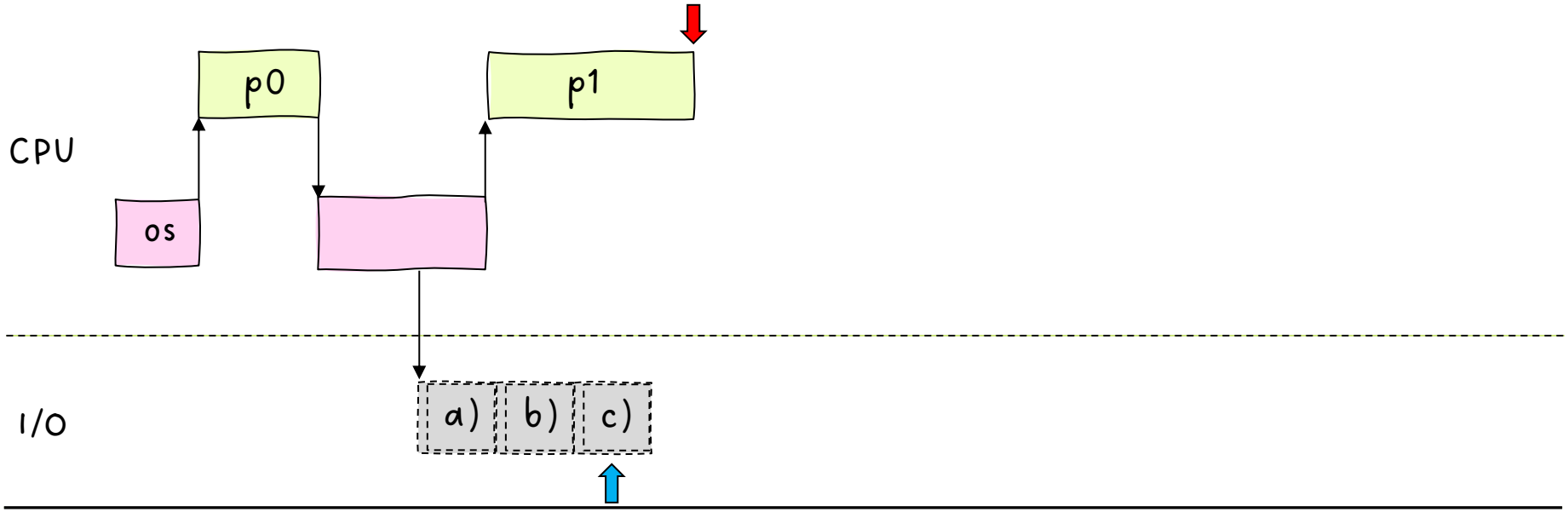
Page Table of p0

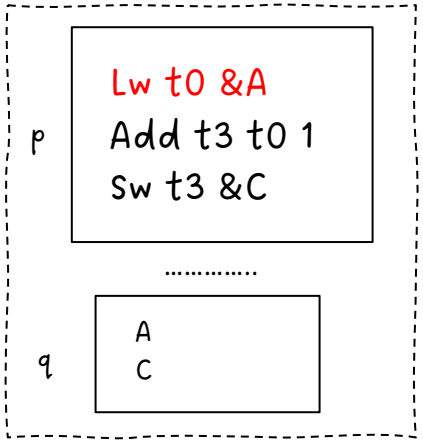
p	f.	v/i
p	L	v
q		i



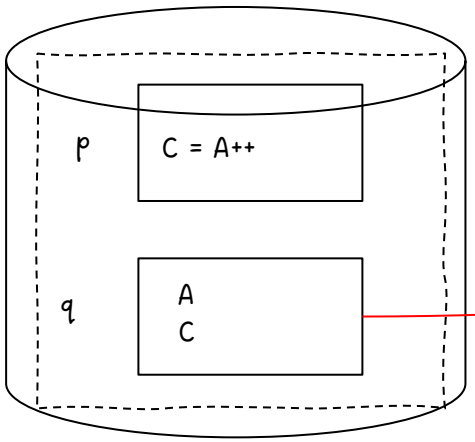
Physical address space

6) While waiting, allocate the CPU to some other user



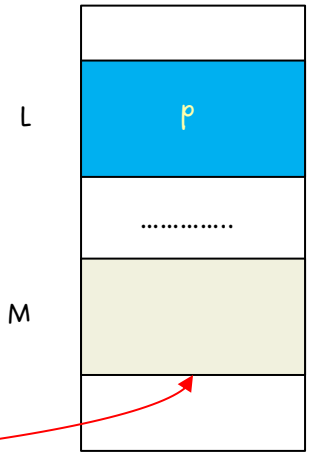


Virtual address space



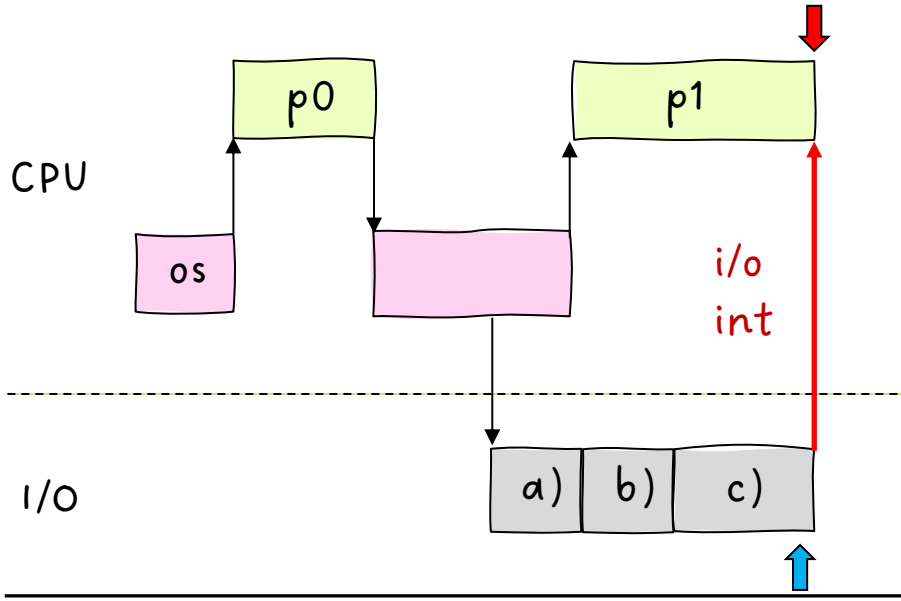
Page Table of p0

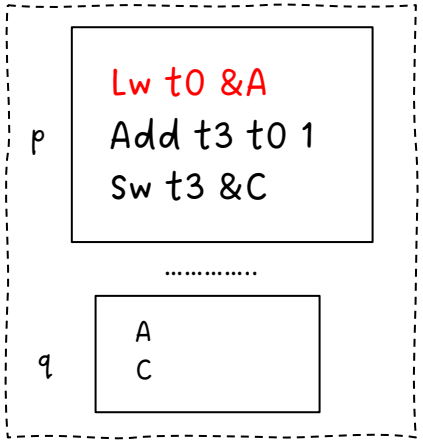
p	f.	v/i
p	L	v
q		i



Physical address space

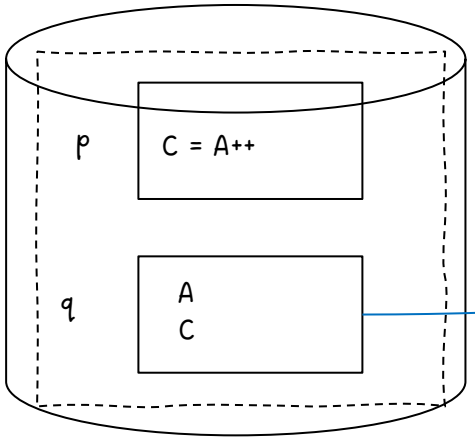
7. Receive an interrupt from the disk I/O subsystem (I/O completed)





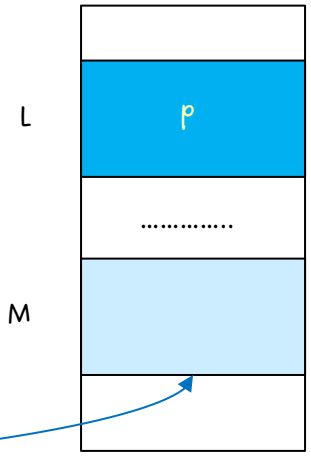
Virtual address space

Executable file of p0



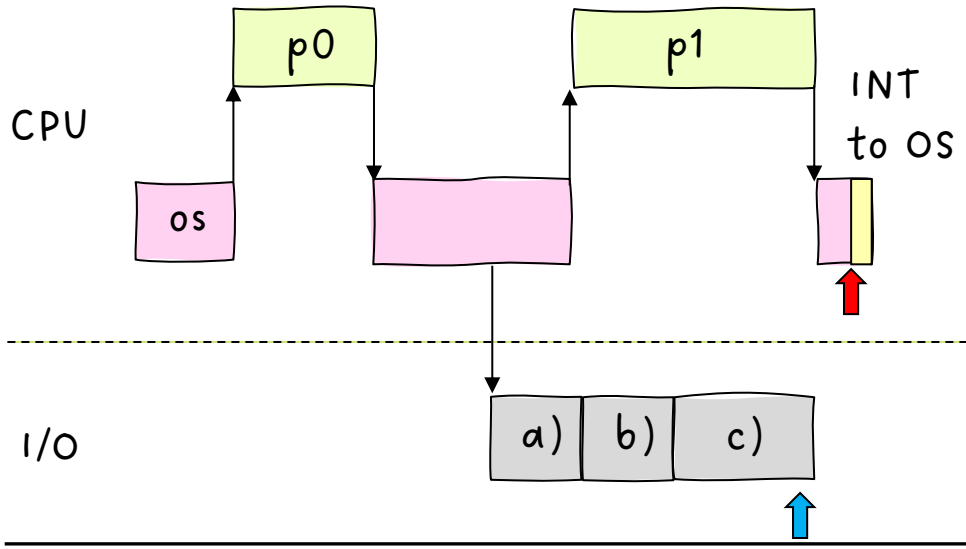
Page Table of p0

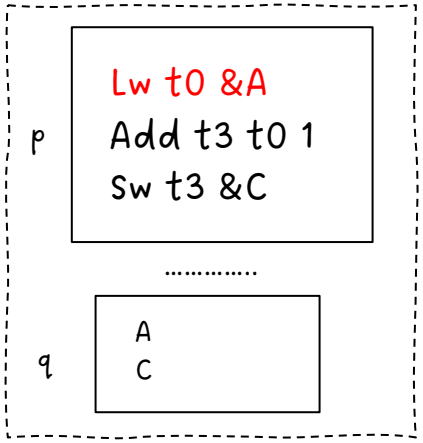
p	f.	v/i
p	L	v
q		i



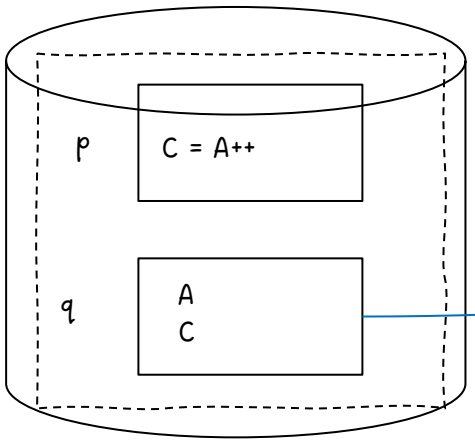
Physical address space

8. Save the registers and process state for the other user



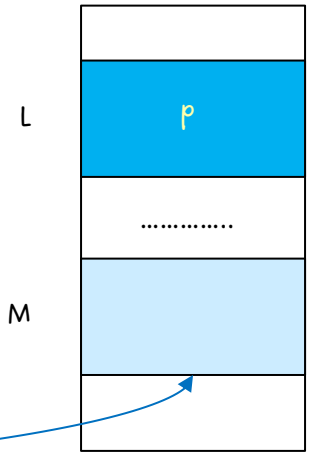


Virtual address space



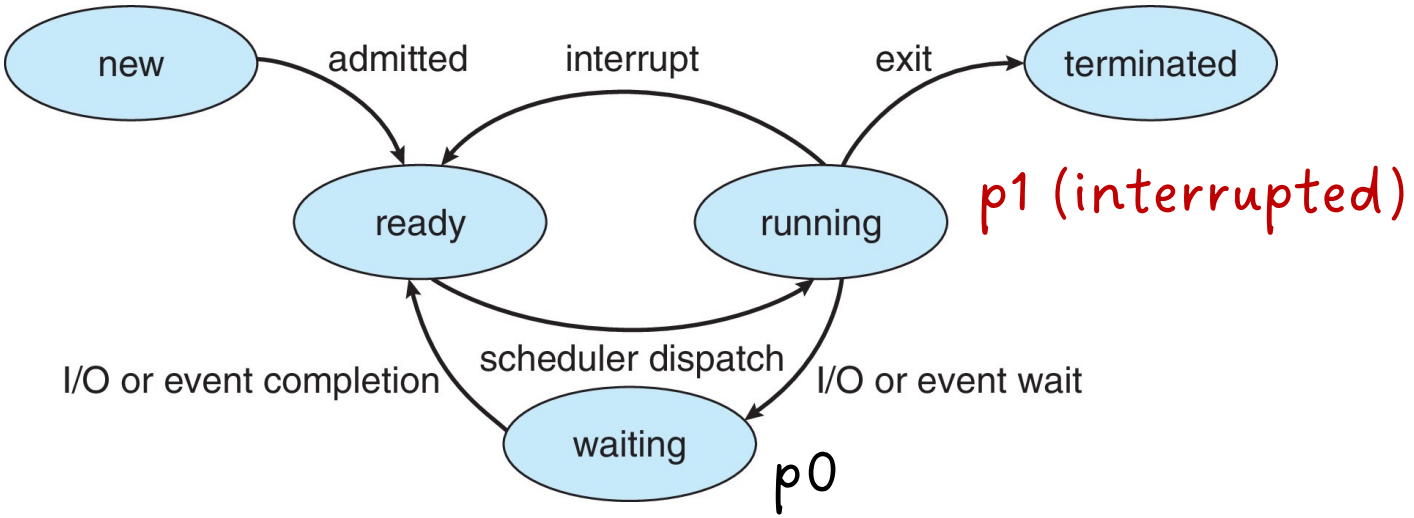
Page Table of p0

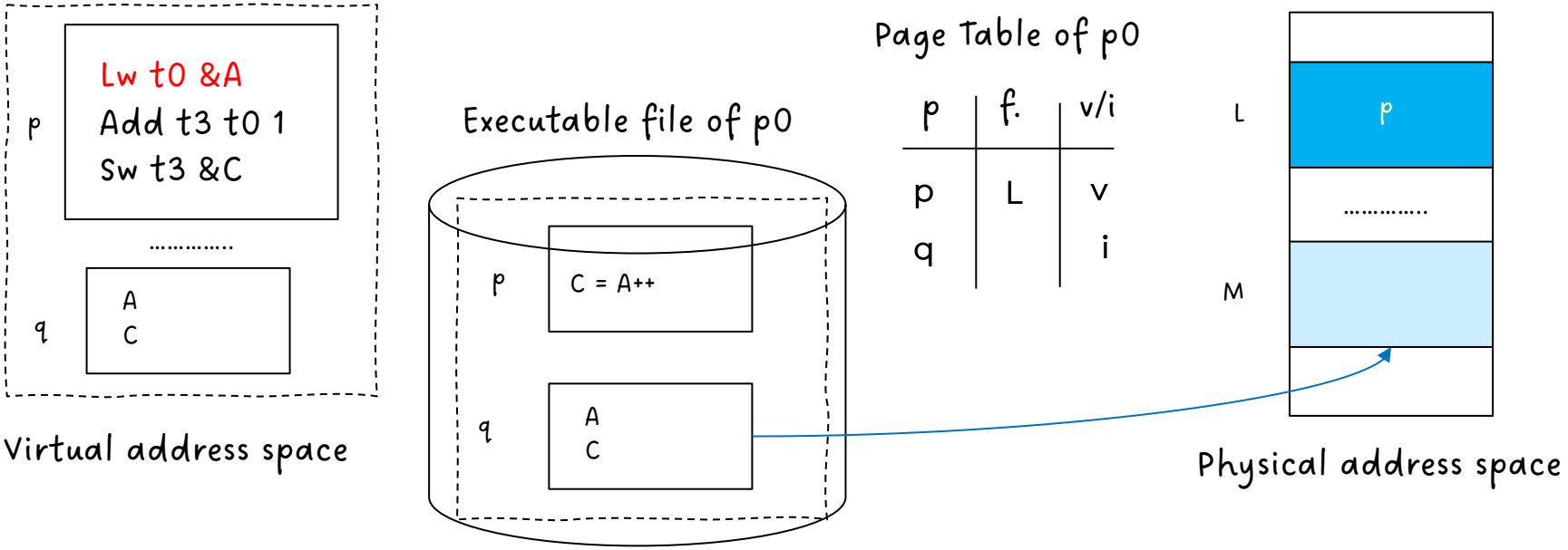
p	f.	v/i
p	L	v
q		i



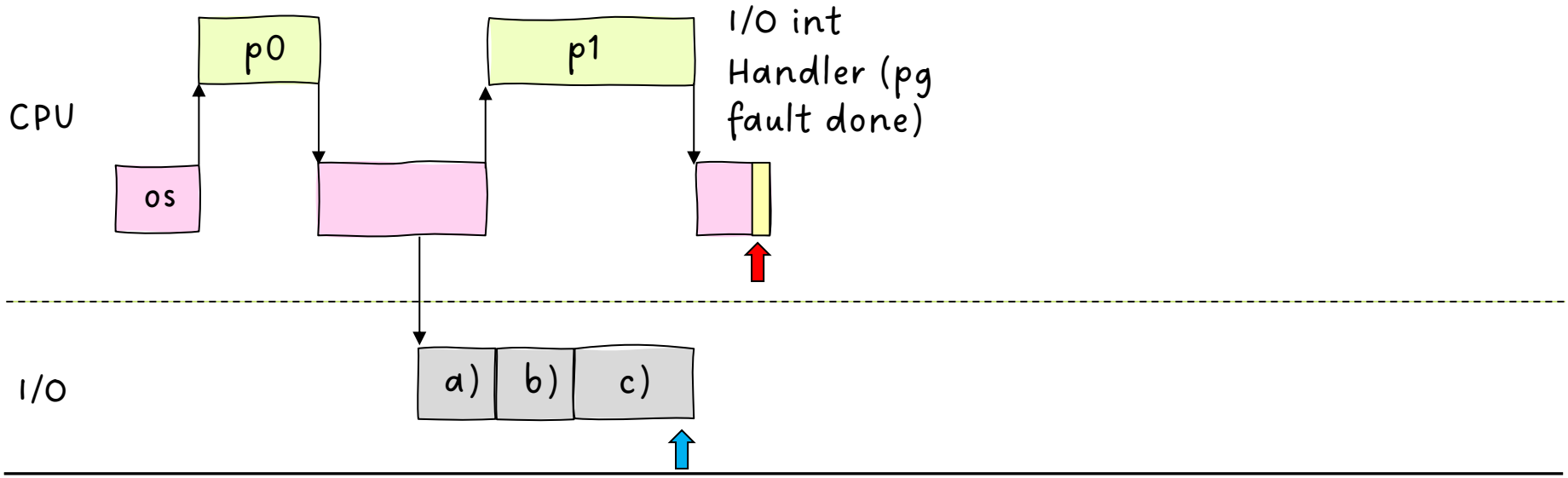
Physical address space

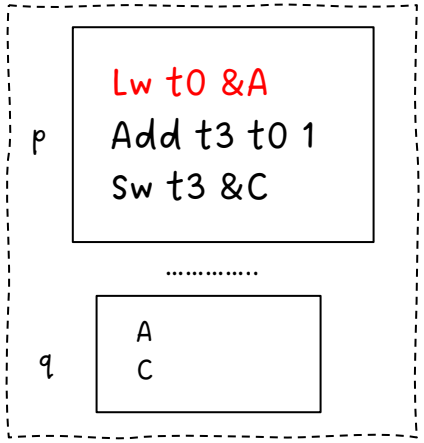
8. Save the registers and process state for the other user





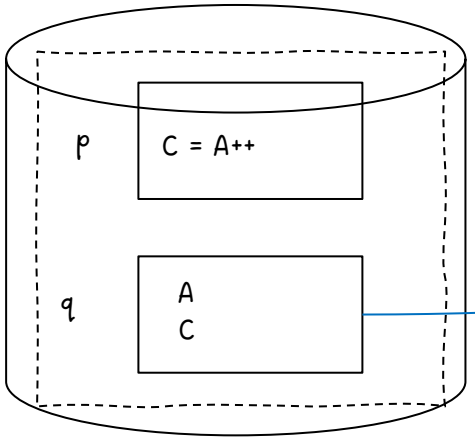
9. Determine that the interrupt was from the disk





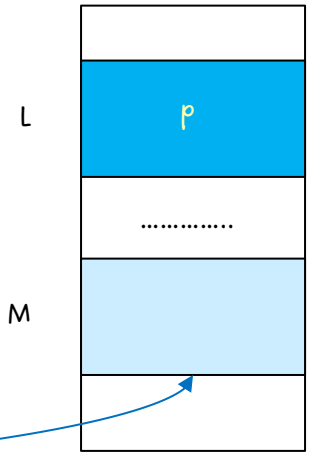
Virtual address space

Executable file of p0



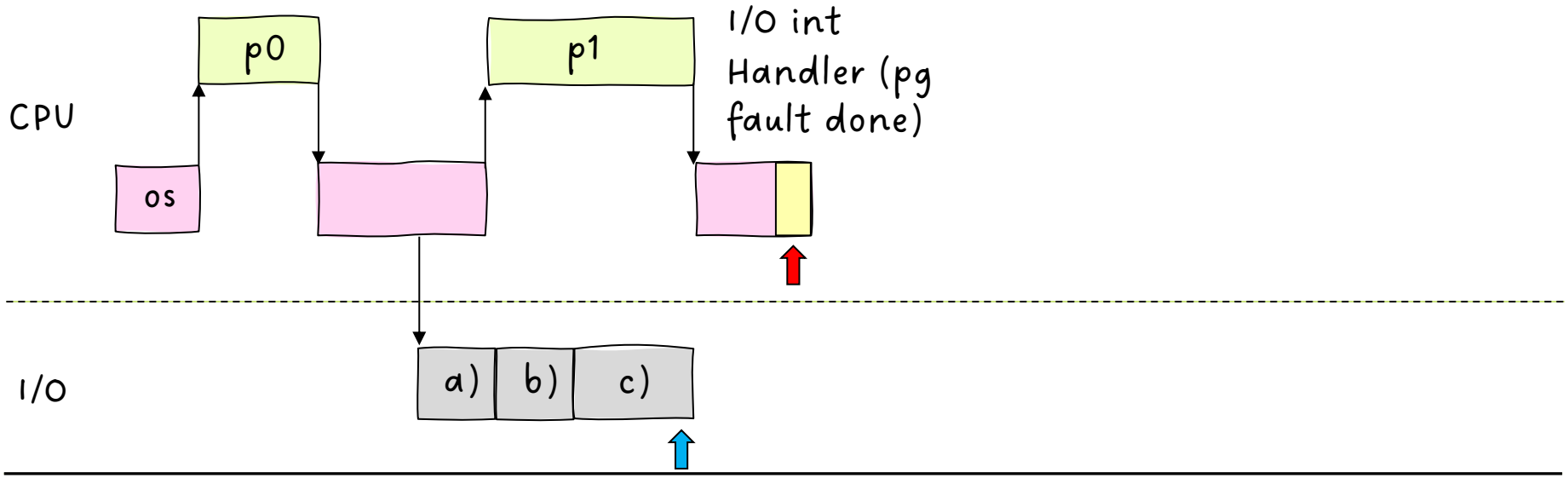
Page Table of p0

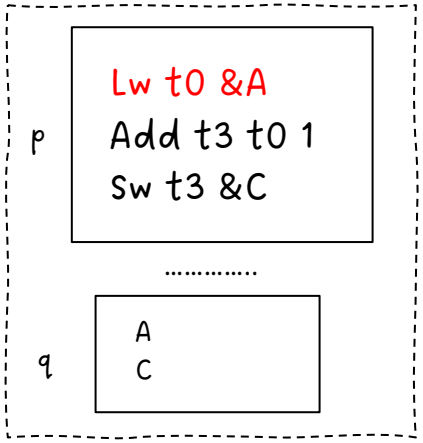
p	f.	v/i
p	L	v
q		i



Physical address space

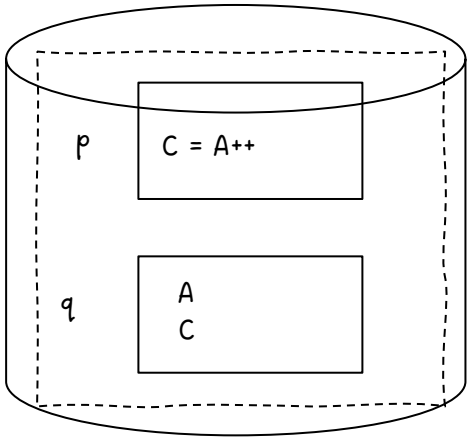
10. Correct the page table and other tables to show page is now in memory





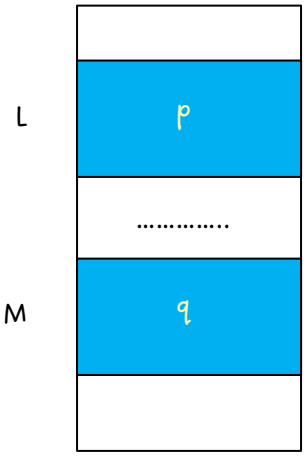
Virtual address space

Executable file of p0



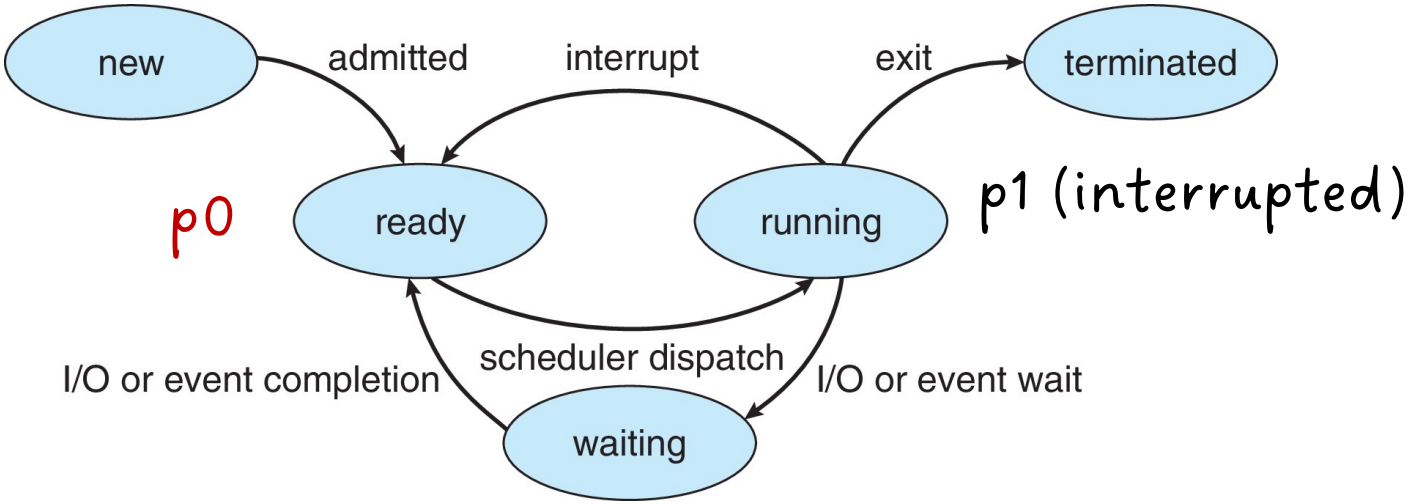
Page Table of p0

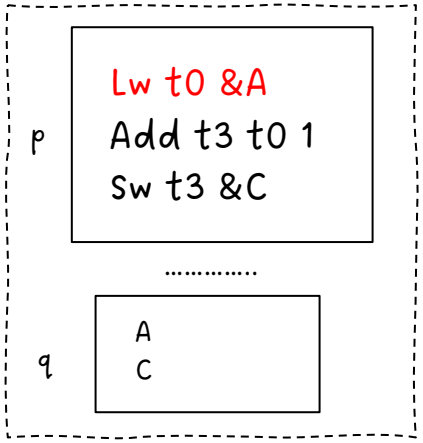
p	f.	v/i
p	L	v
q	M	v



Physical address space

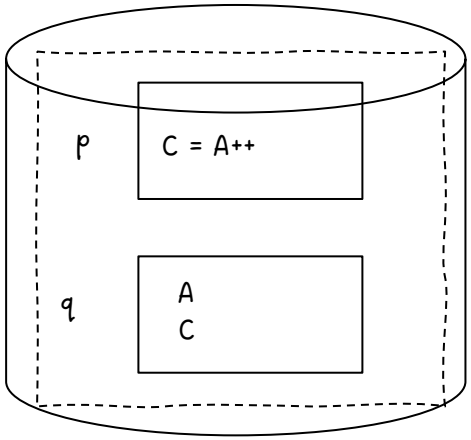
10. Correct the page table and other tables to show page is now in memory





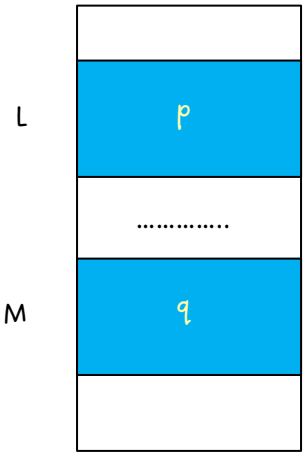
Virtual address space

Executable file of p0



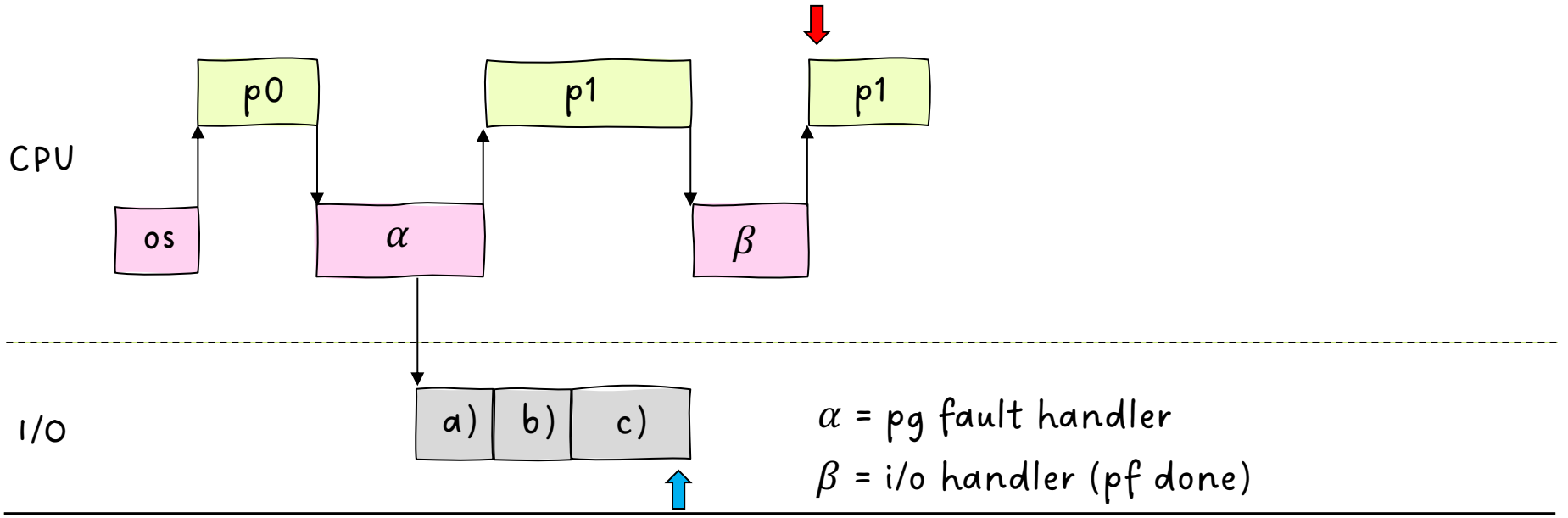
Page Table of p0

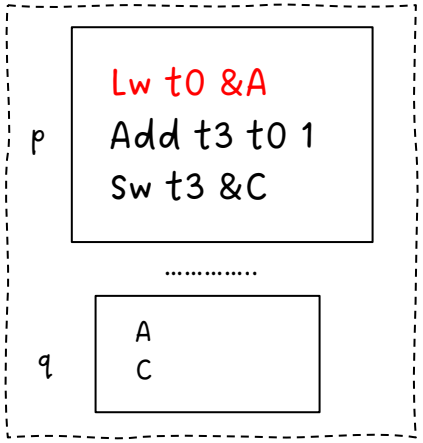
p	f.	v/i
p	L	v
q	M	v



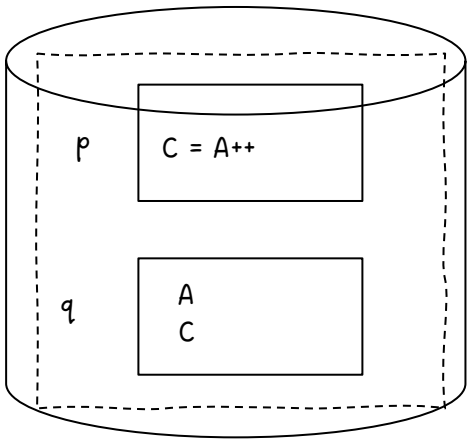
Physical address space

11. Wait for the CPU to be allocated to this process again



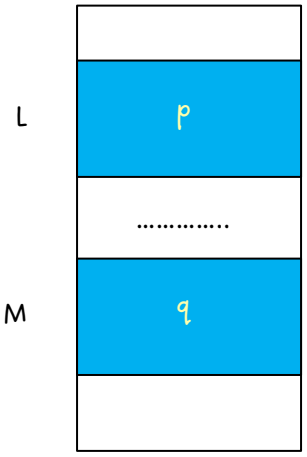


Virtual address space



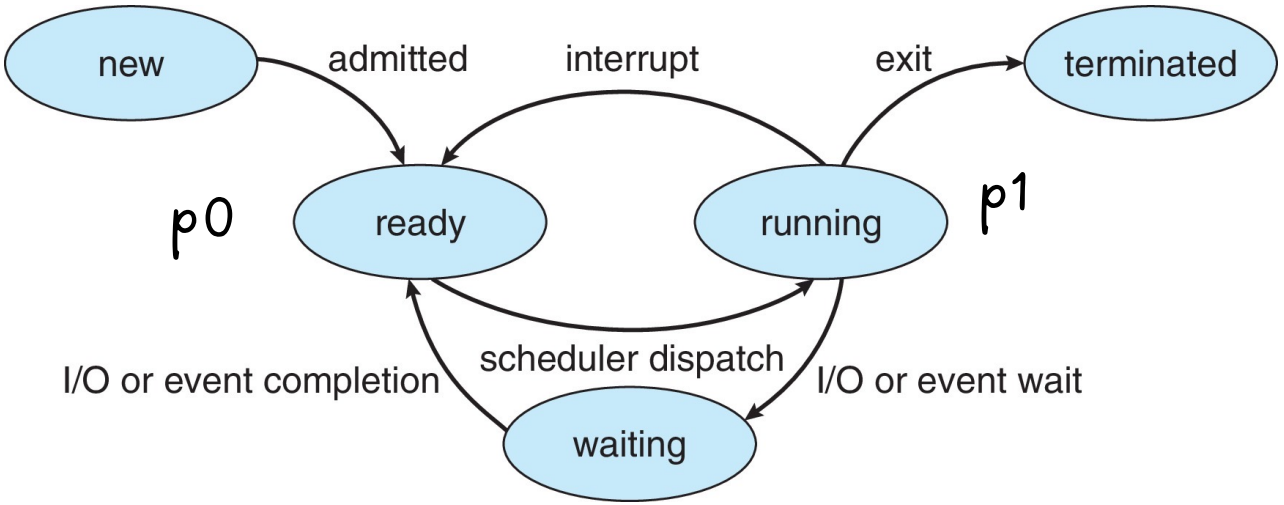
Page Table of p0

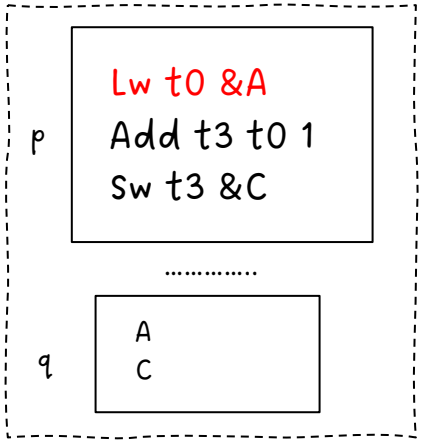
p	f.	v/i
p	L	v
q	M	v



Physical address space

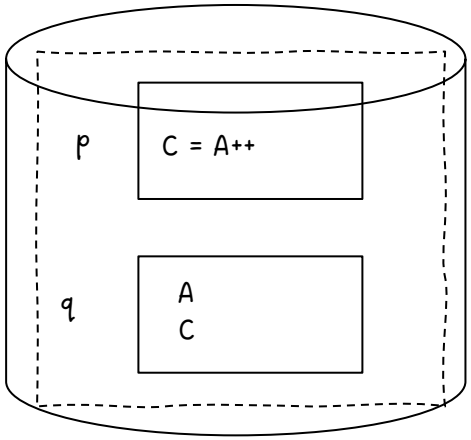
11. Wait for the CPU to be allocated to this process again





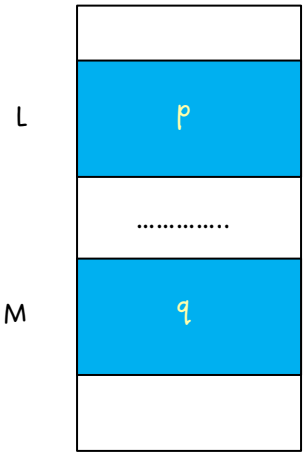
Virtual address space

Executable file of p0



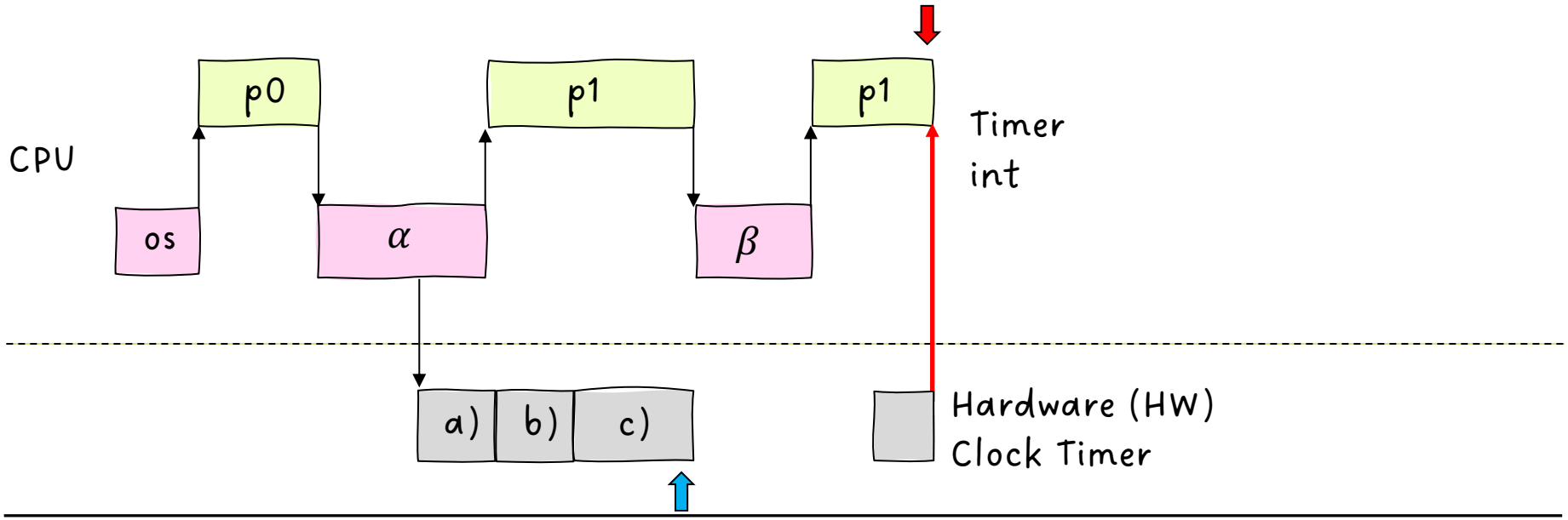
Page Table of p0

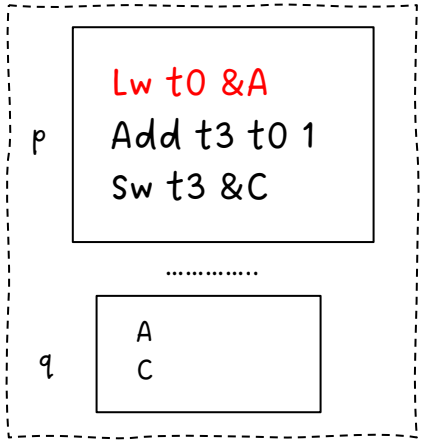
p	f.	v/i
p	L	v
q	M	v



Physical address space

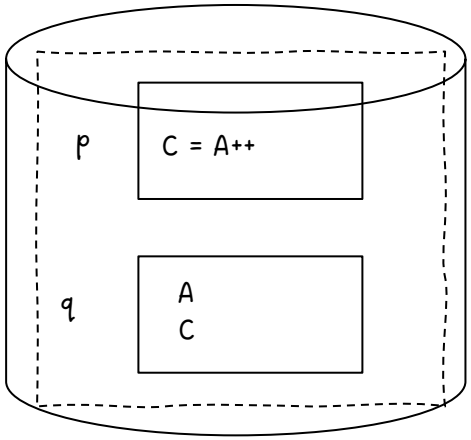
11. Wait for the CPU to be allocated to this process again





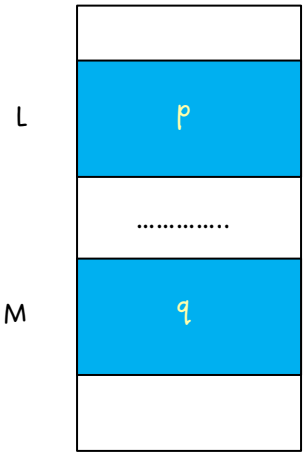
Virtual address space

Executable file of p0



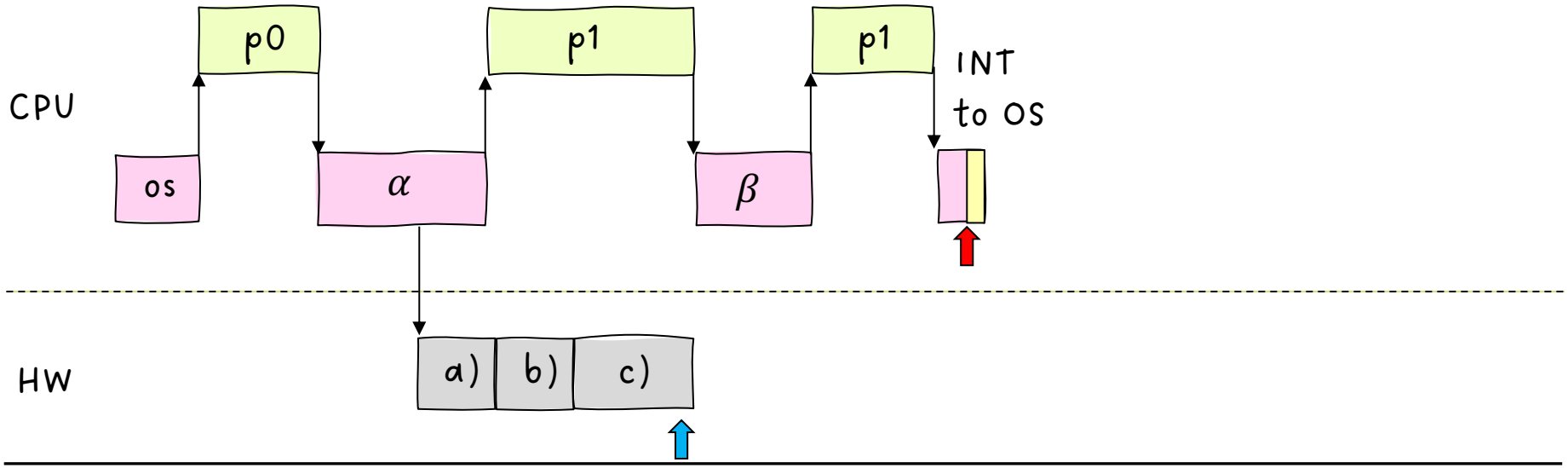
Page Table of p0

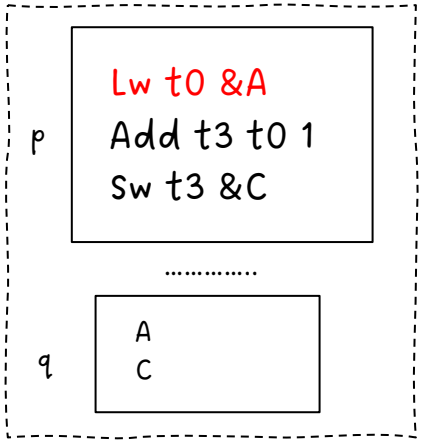
p	f.	v/i
p	L	v
q	M	v



Physical address space

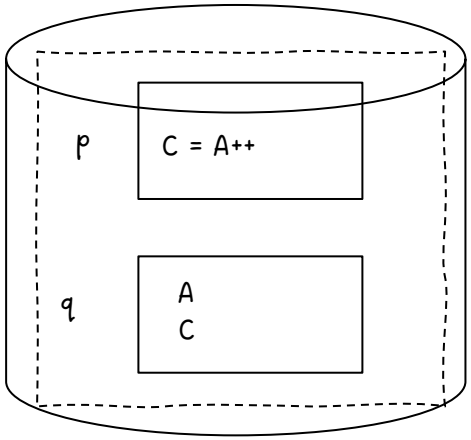
11. Wait for the CPU to be allocated to this process again





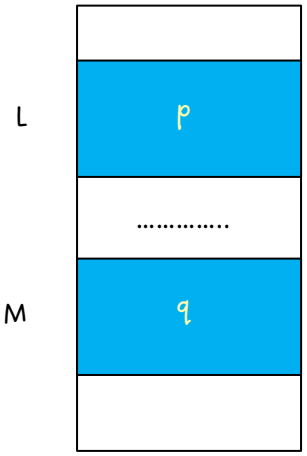
Virtual address space

Executable file of p0



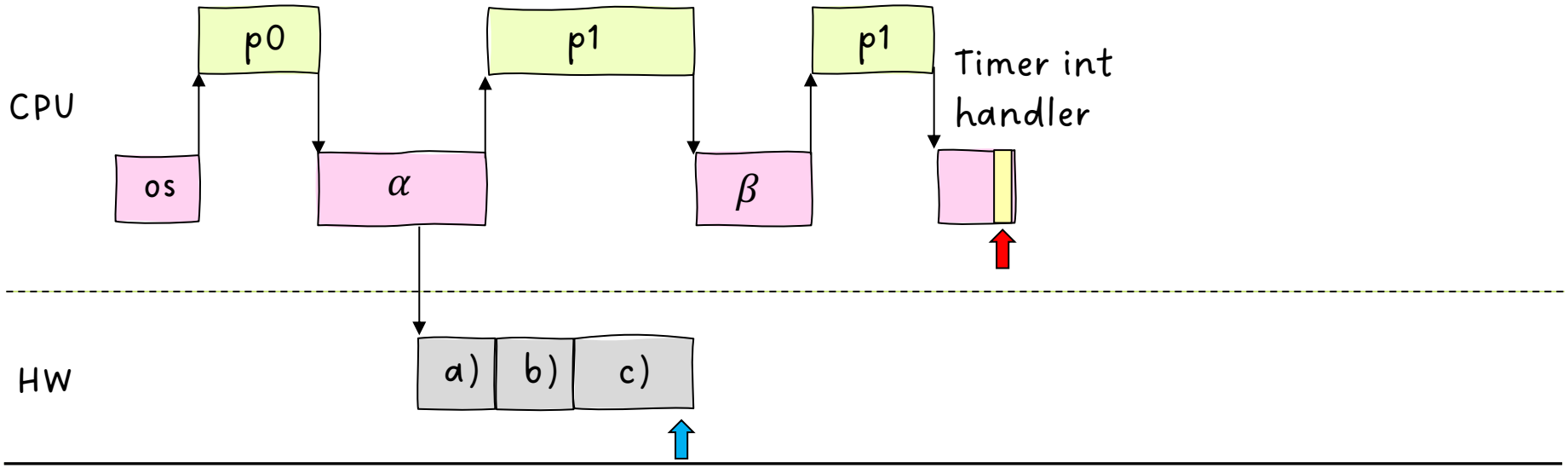
Page Table of p0

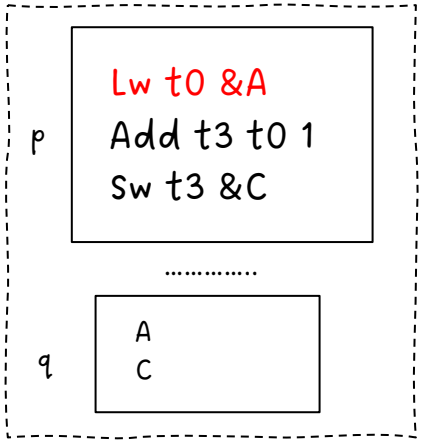
p	f.	v/i
p	L	v
q	M	v



Physical address space

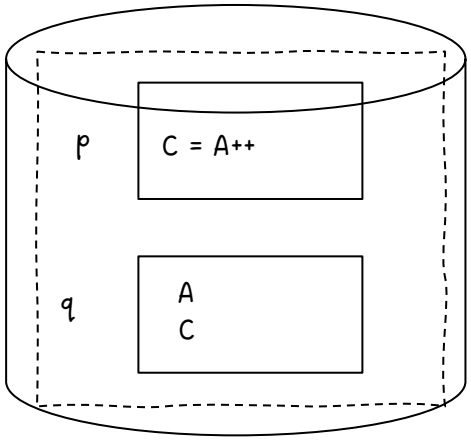
11. Wait for the CPU to be allocated to this process again





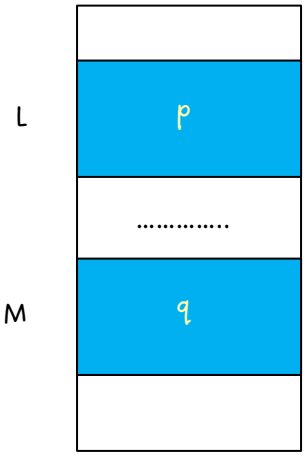
Virtual address space

Executable file of p0



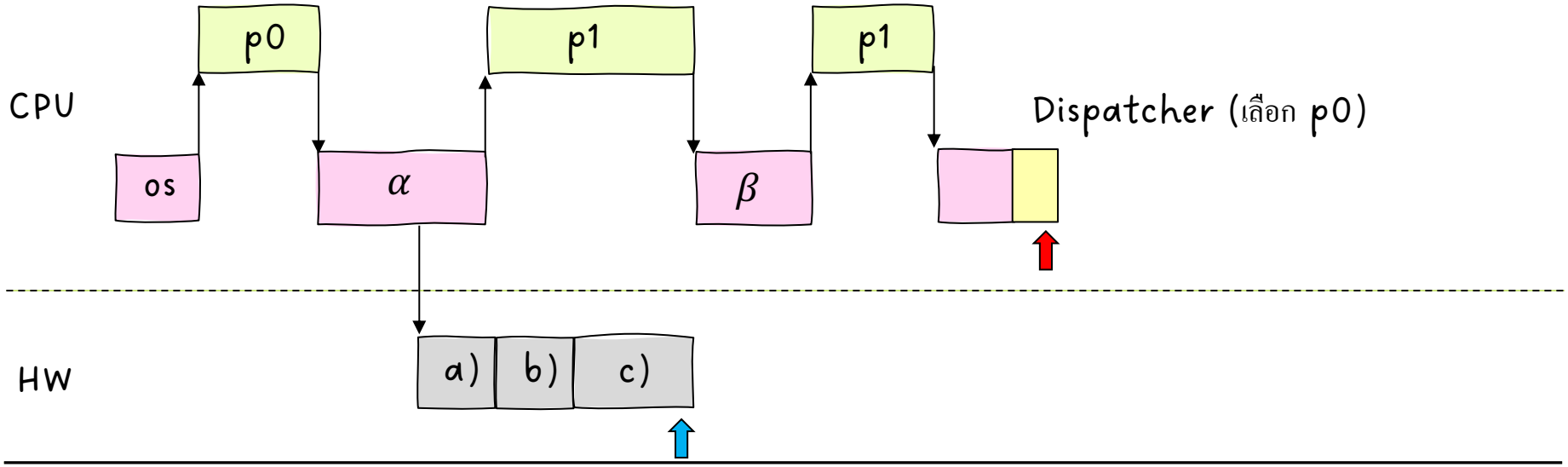
Page Table of p0

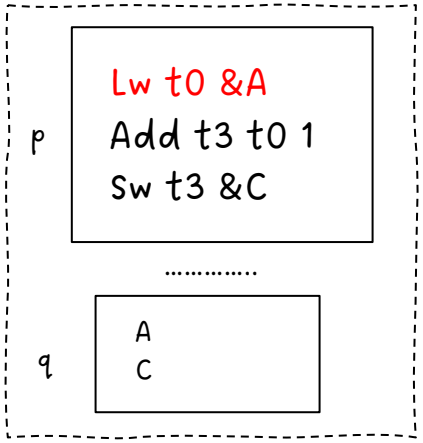
p	f.	v/i
p	L	v
q	M	v



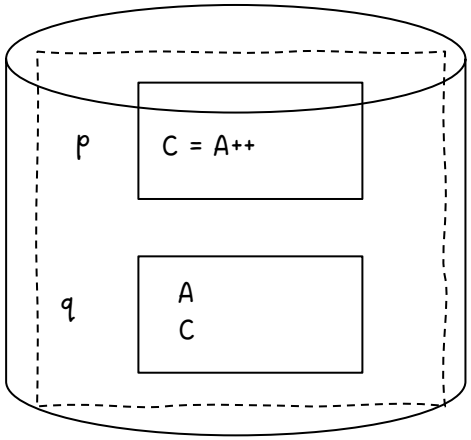
Physical address space

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction



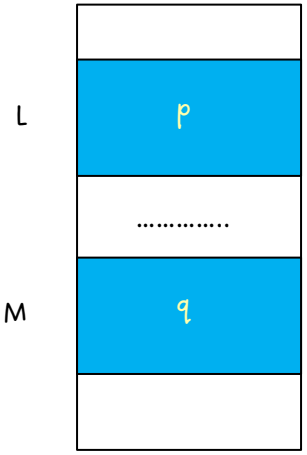


Virtual address space



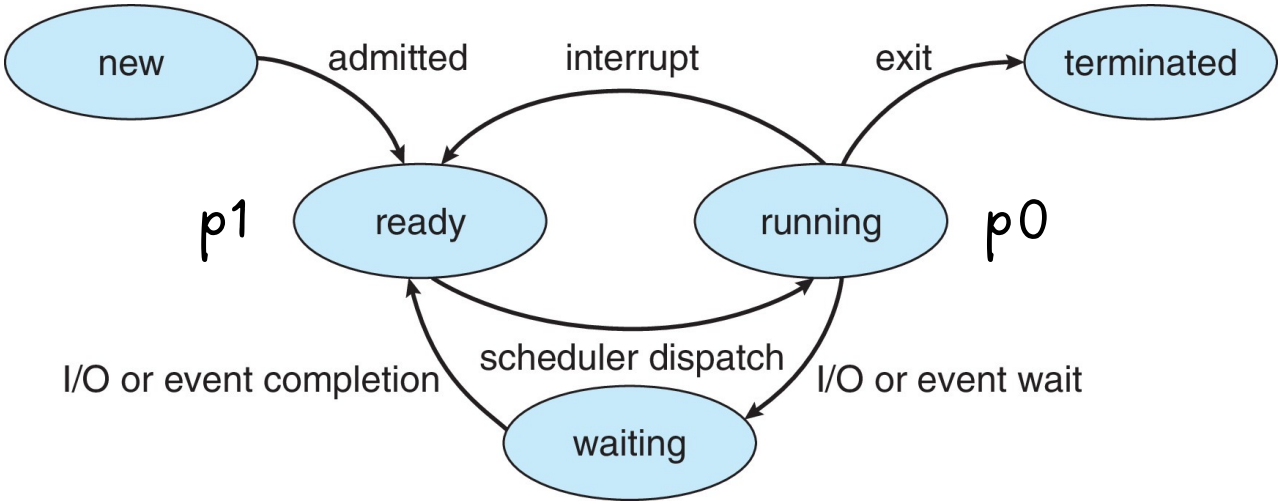
Page Table of p0

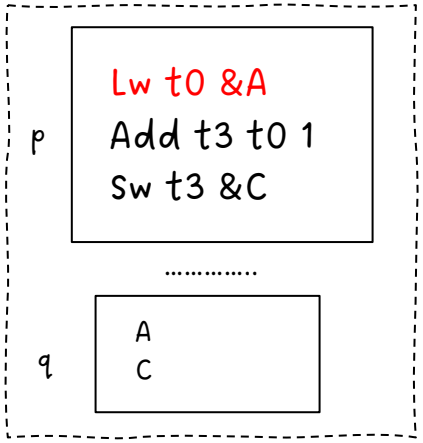
p	f.	v/i
p	L	v
q	M	v



Physical address space

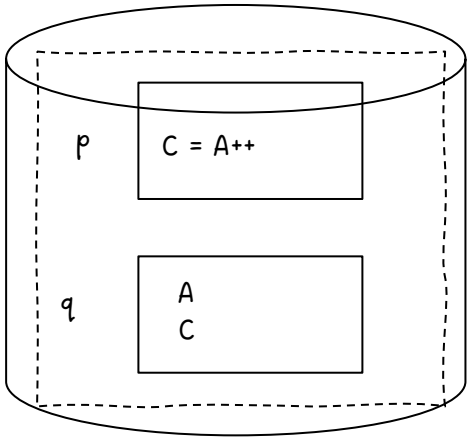
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction





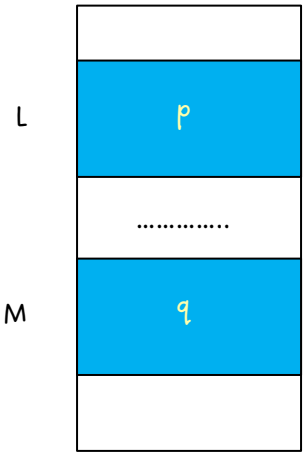
Virtual address space

Executable file of p0



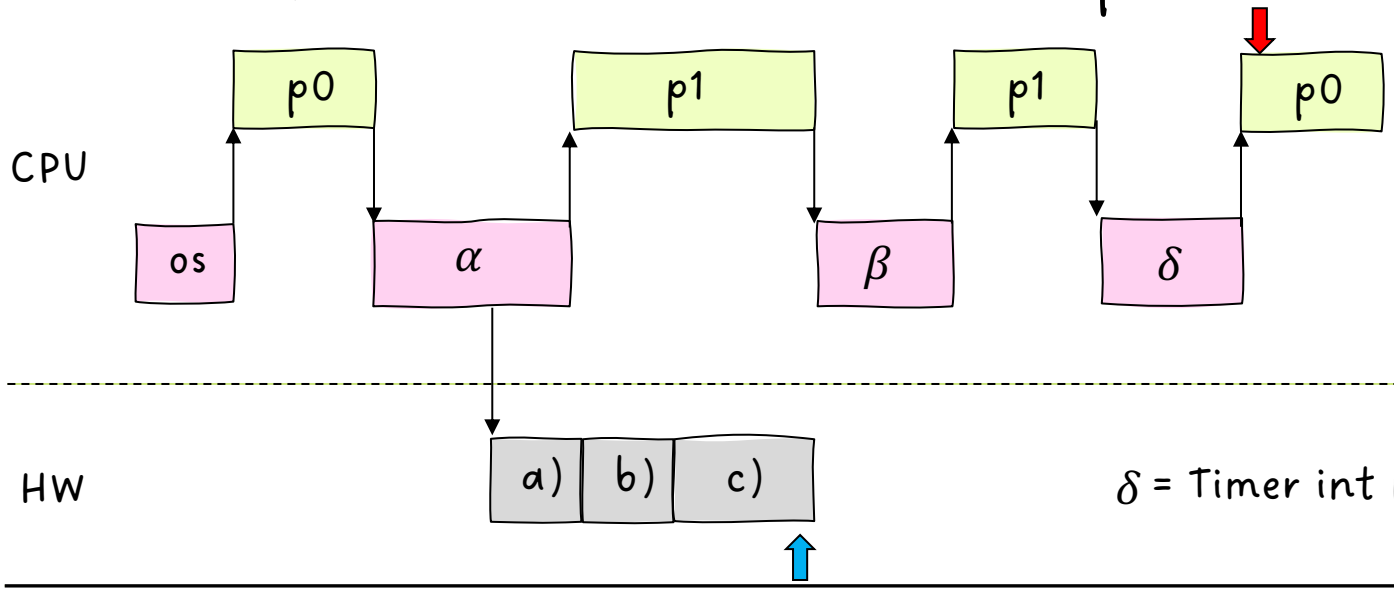
Page Table of p0

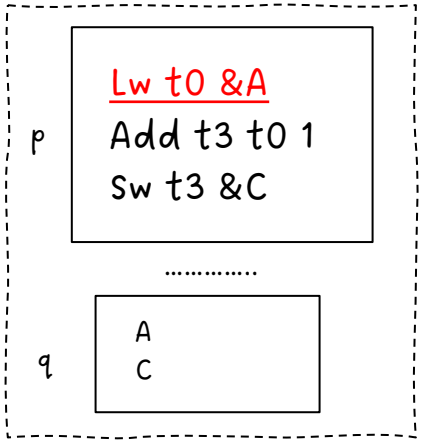
p	f.	v/i
p	L	v
q	M	v



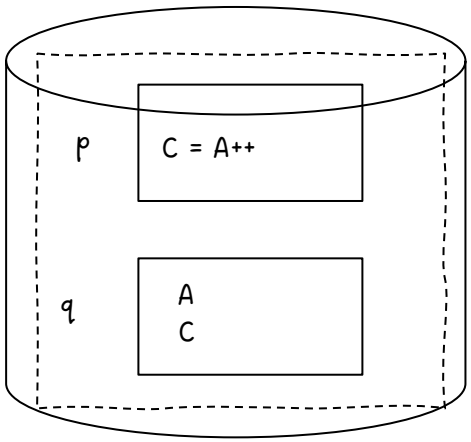
Physical address space

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction



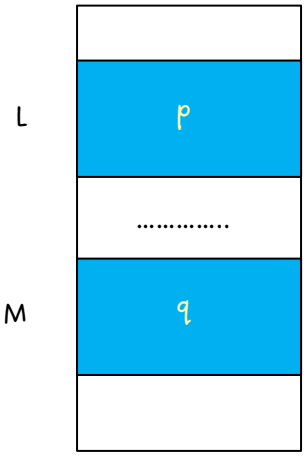


Virtual address space



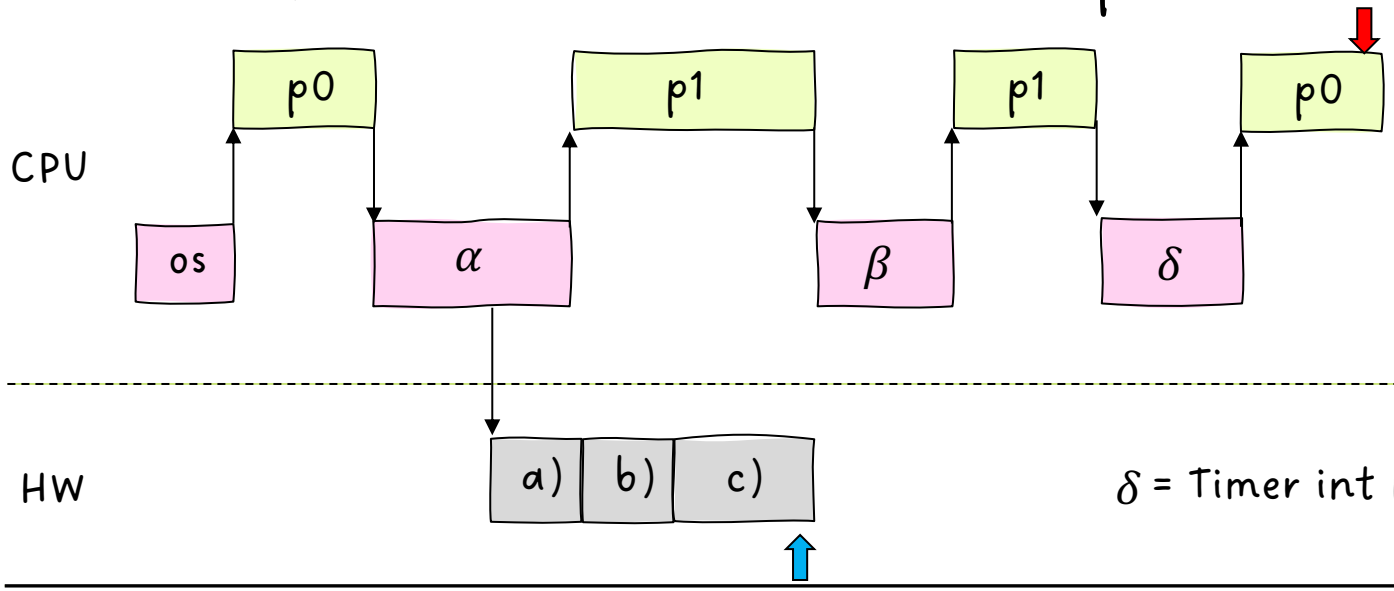
Page Table of p0

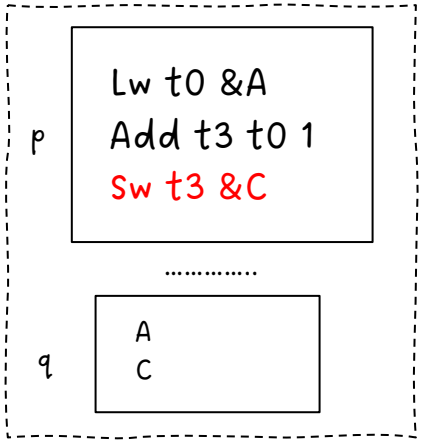
p	f.	v/i
p	L	v
q	M	v



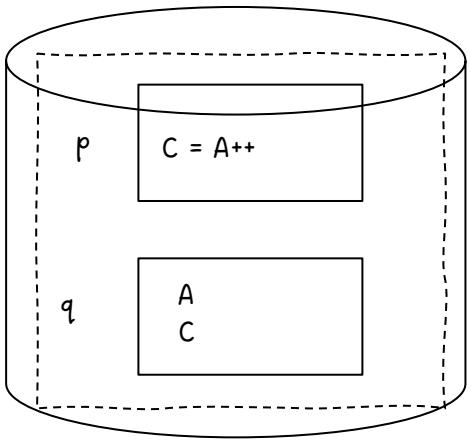
Physical address space

12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction



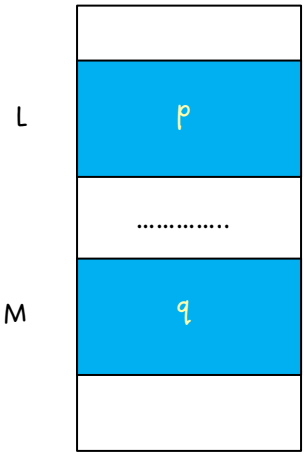


Virtual address space



Page Table of p0

p	f.	v/i
p	L	v
q	M	v



Physical address space

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p (page fault overhead
  + swap page out
  + swap page in)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

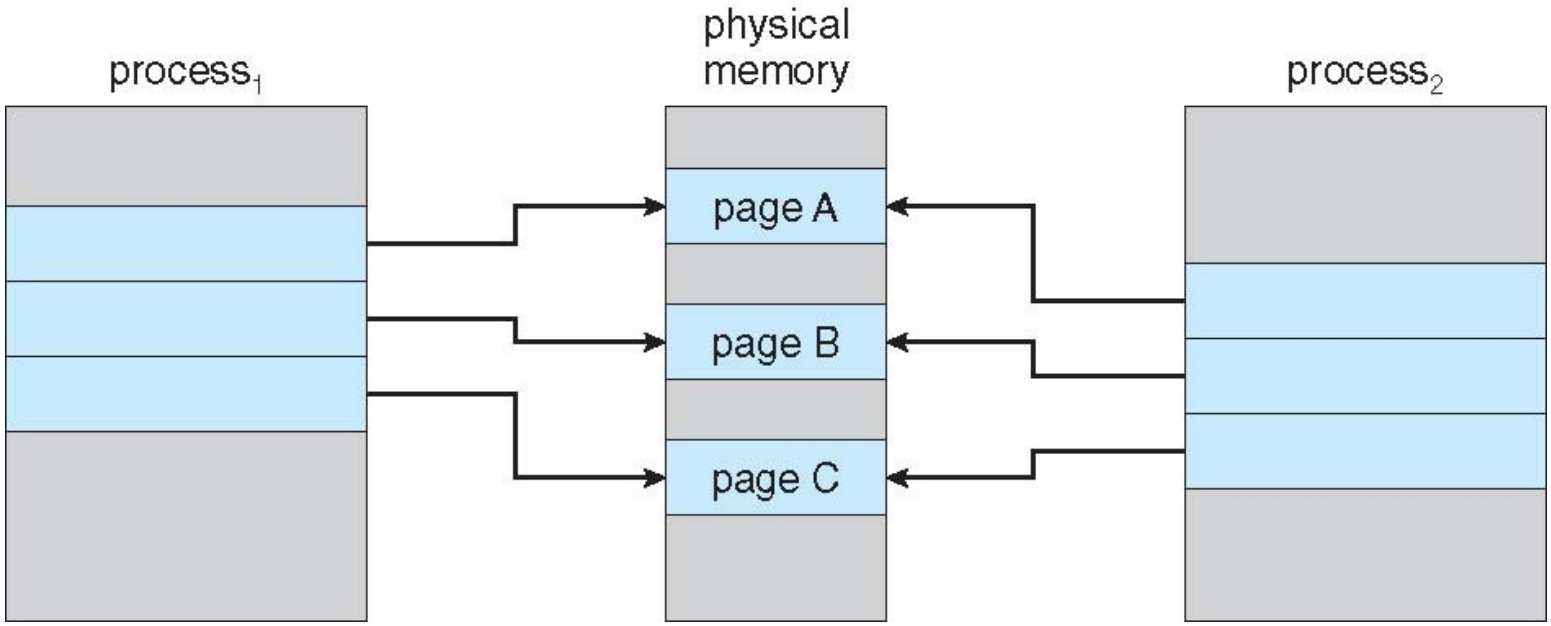
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

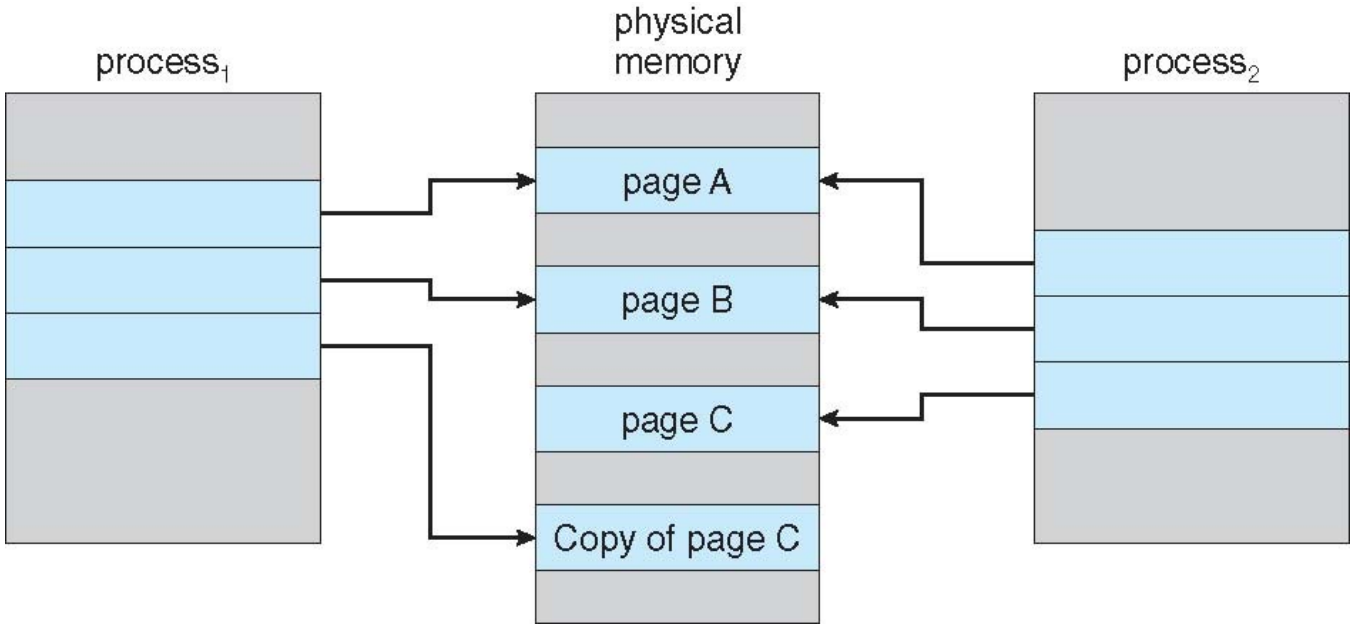
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



กำหนดให้ shared pages มีคุณสมบัติเป็น Write-protected คือห้ามเขียน

After Process 1 Modifies Page C



1. เมื่อ process ใด process หนึ่งเขียน Write-protected ก็จะเกิด Trap
2. Trap handler จะทำการ Copy Page C แล้วเปลี่ยน Page Table Entry ให้ชี้ไปยัง Page “Copy of page C” ซึ่งไม่ติด write protect
3. หลังจากนั้น P1 สามารถ modify page 3 ของมันได้

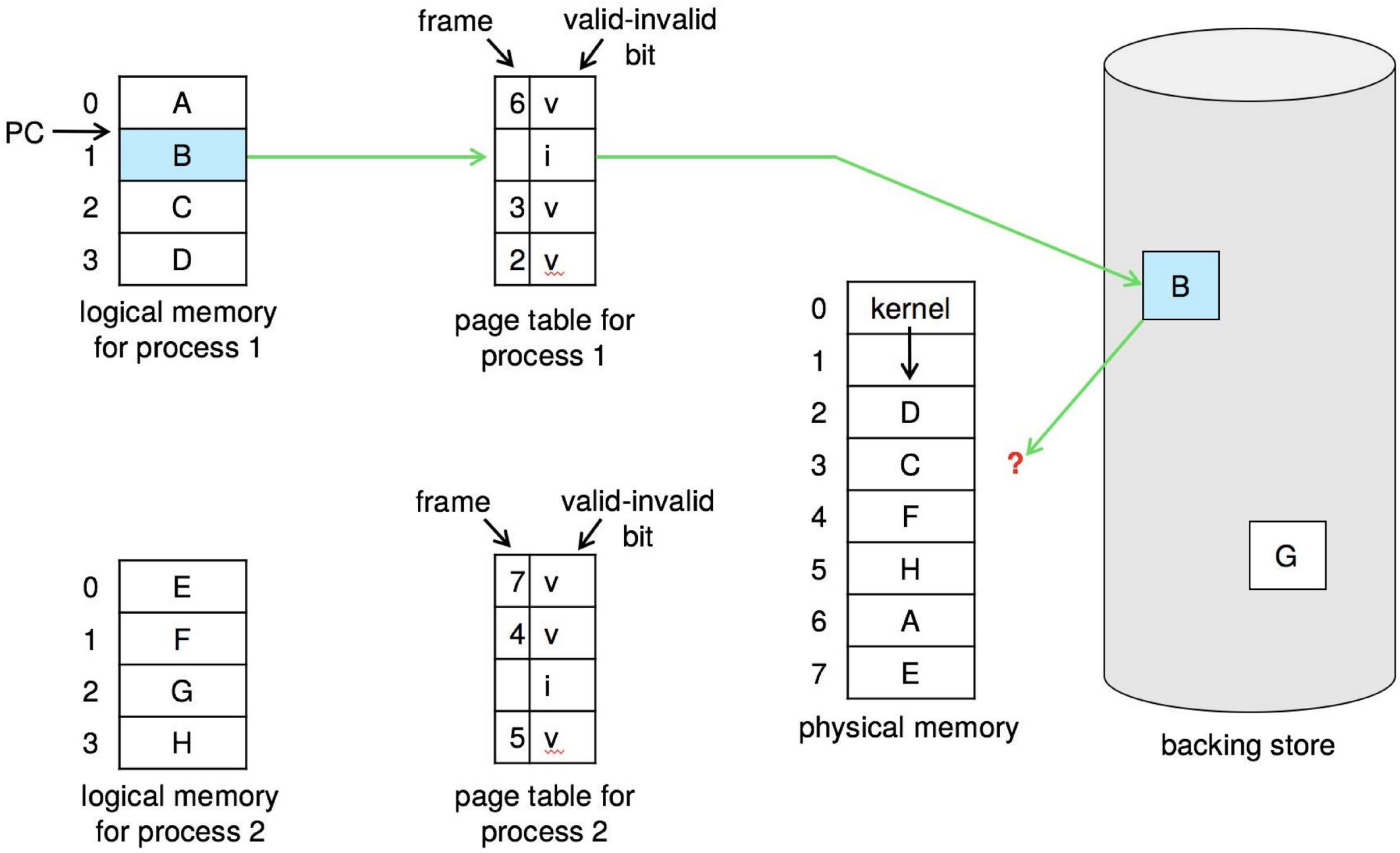
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to *include page replacement*
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- *Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory*
  - *ทำให้เกิดการแยกกันอย่างแท้จริง ของ logical และ physical memory*
  - *ทำให้เราสามารถ load program ขนาดใหญ่บนพื้นที่ physical memory จำกัดได้*

Need For Page Replacement



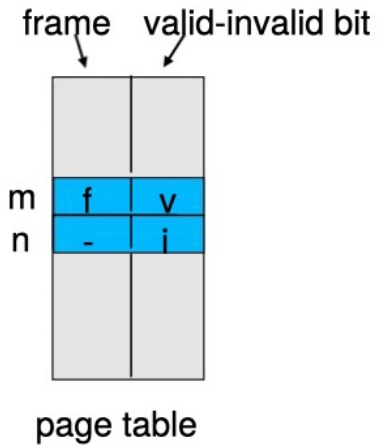
Basic Page Replacement

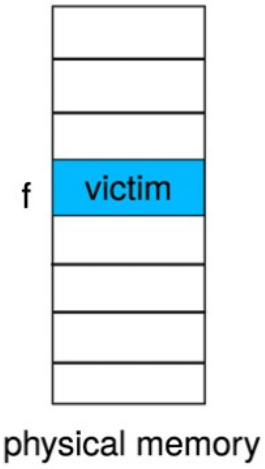
1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

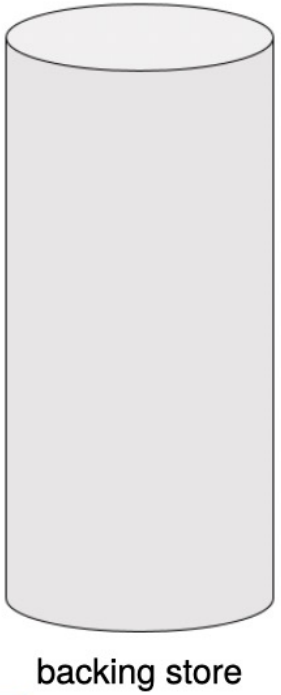
Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement

in this example
- page (m) is in memory at frame (f) and its page table entry is set to (v)alid
- page (n) is not currently in memory, so it is set to (i)nvalid
- page (n) has been accessed by the current instruction and a page fault has occurred

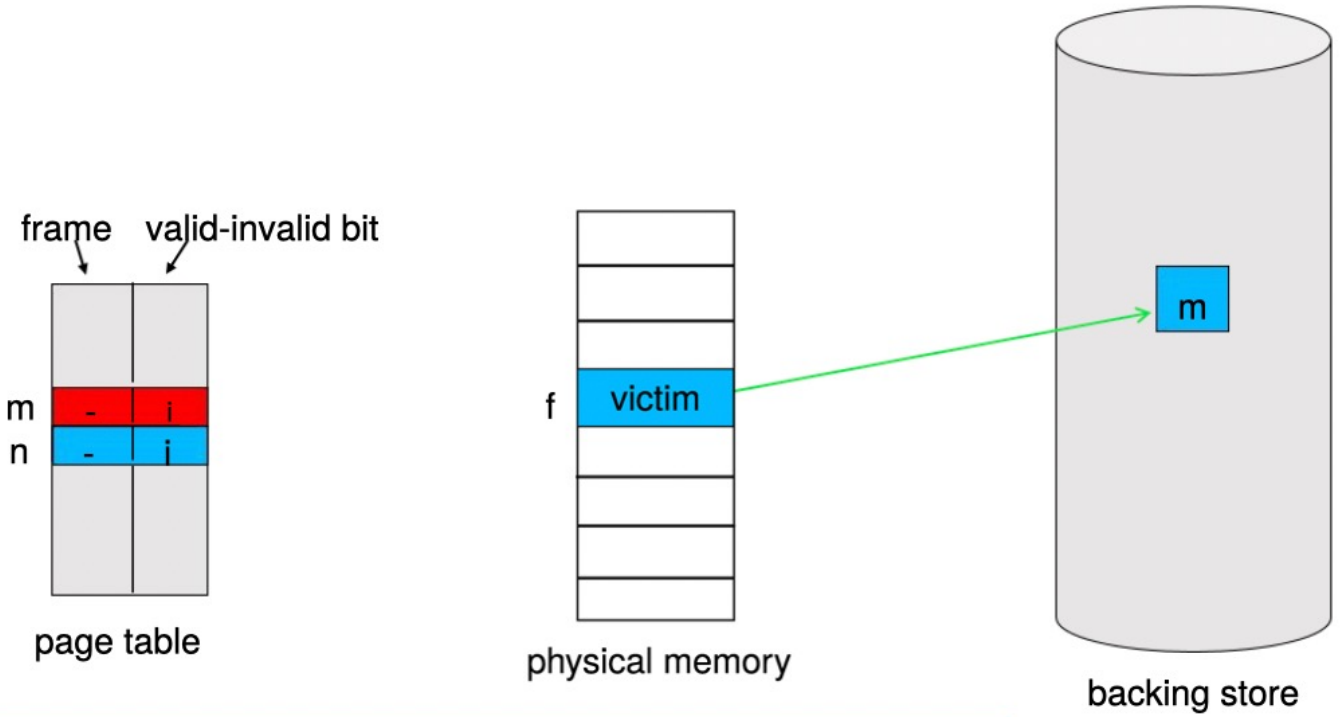






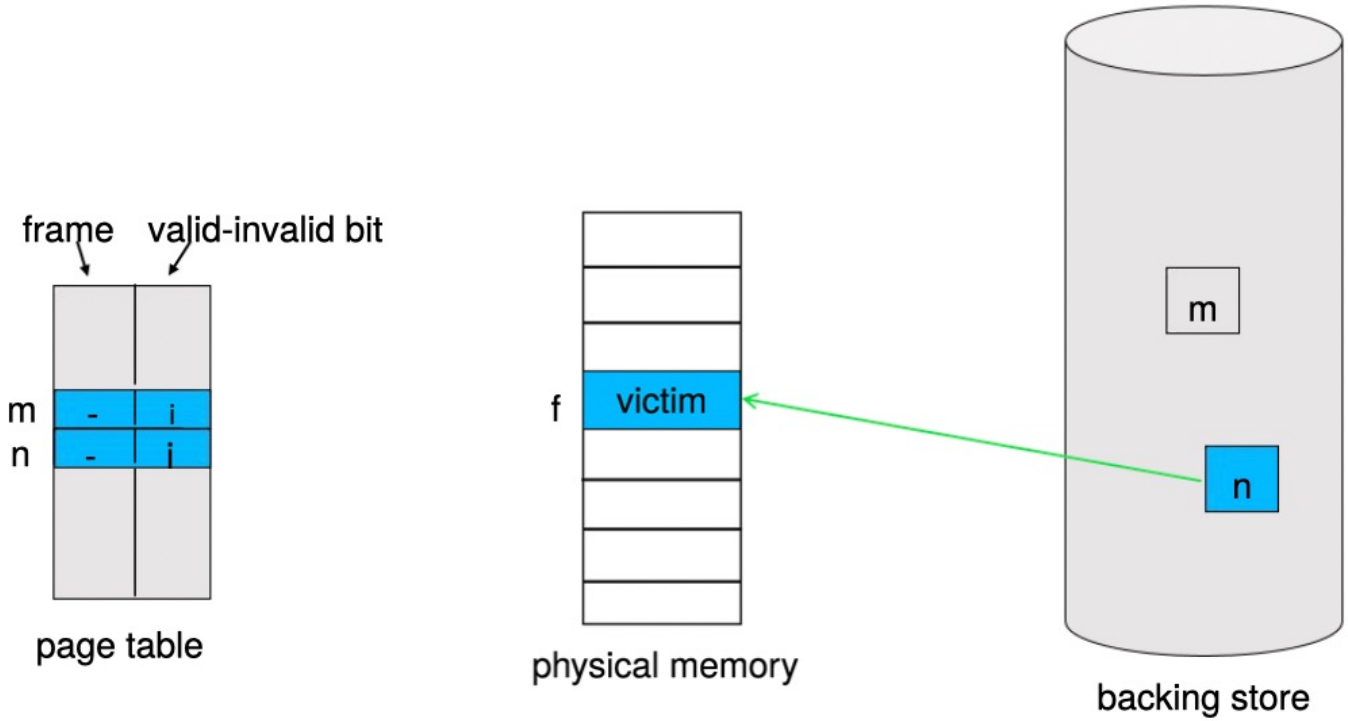
1. find a frame
   (a) if a frame is already free, select it
   (b) if there is no free frame, use a page-replacement algorithm to select a victim frame here there is no free frame so we select (f) as the victim

Page Replacement



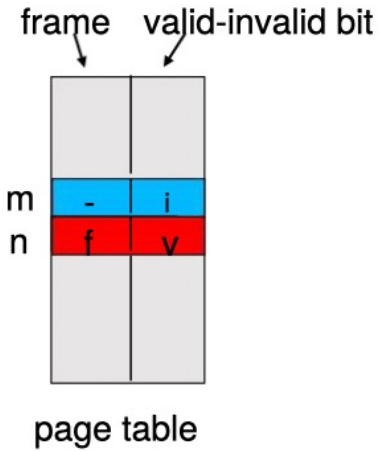
2. (f) is currently holding page (m)
   we write the page to swap space on secondary storage
   (if necessary), free the frame by marking (m) as invalid

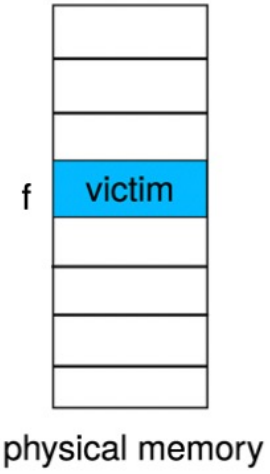
Page Replacement

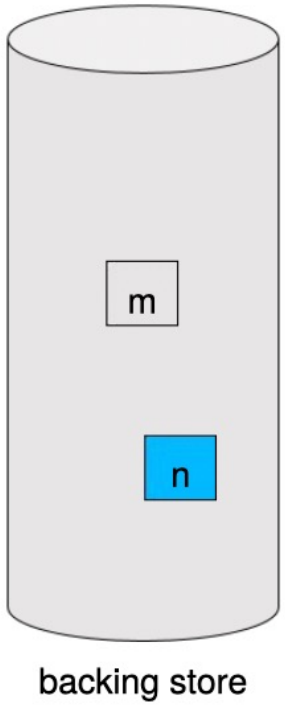


3. we locate page (n) on backing store and schedule an I/O transfer to DMA the contents into frame (f)

Page Replacement

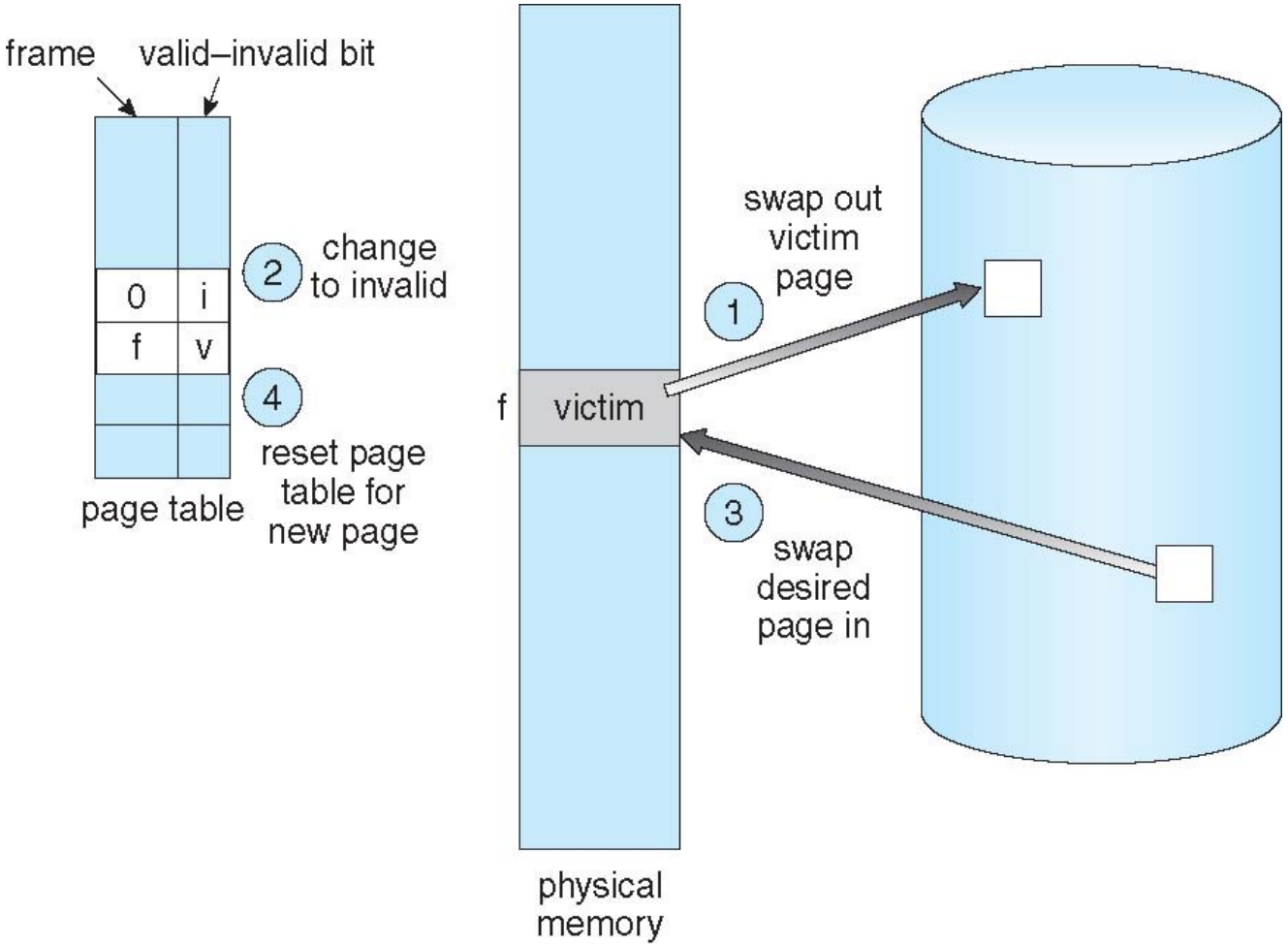






4. (a) once the transfer is complete, change the page table for (n) to point to the frame and note that it is valid
   (b) the instruction that caused the page fault is restarted, and can now access the data it needs (unless other addresses in the instruction are not currently valid!)

Page Replacement

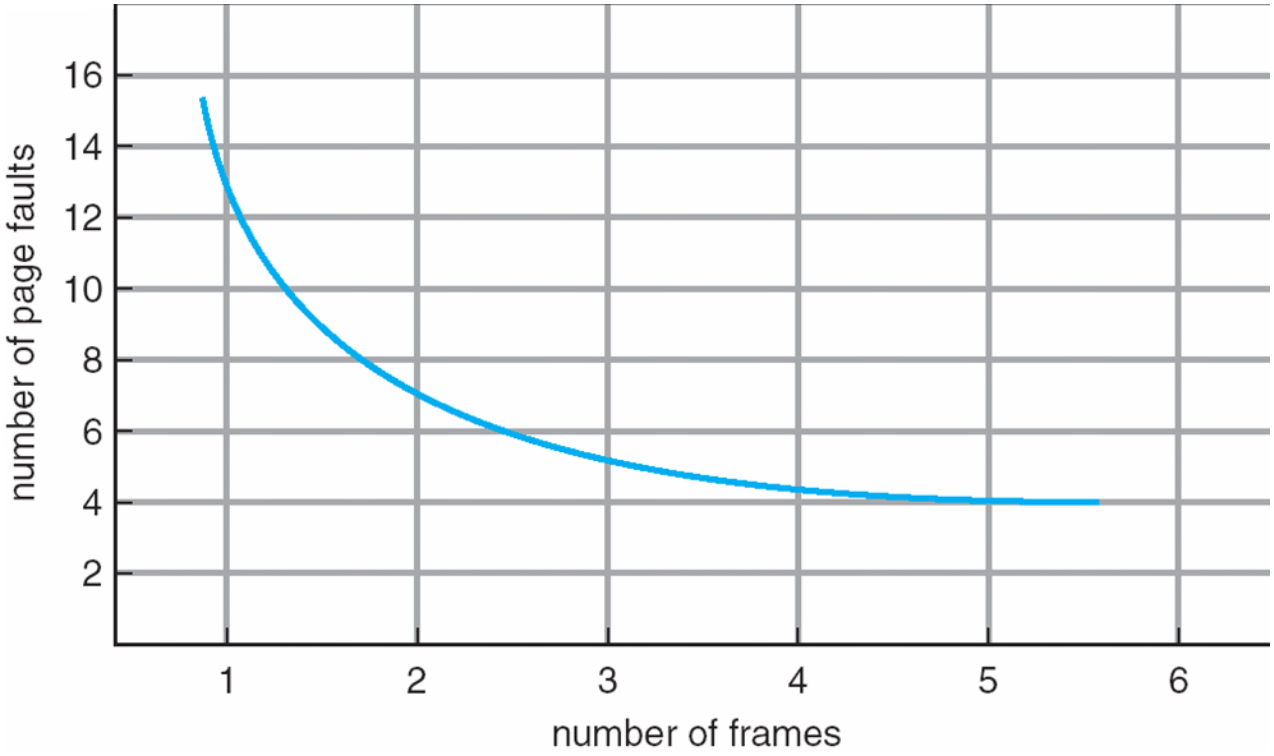


Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

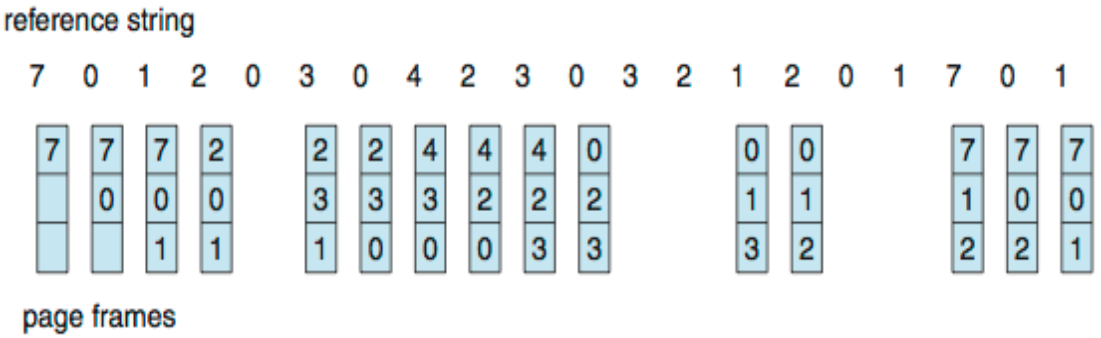
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus the Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

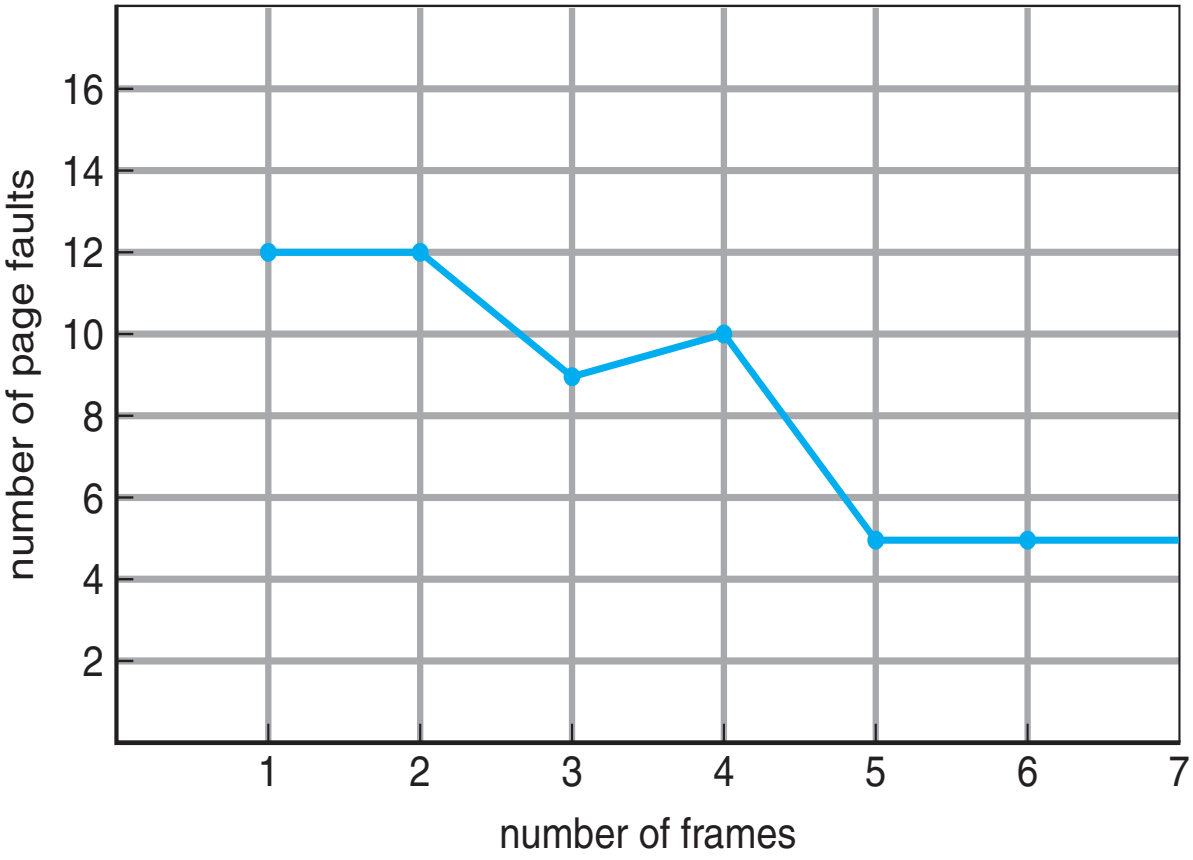
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

1,2,3,4,1,2,5,1,2,3,4,5

1	2	3	4	1	2	5	1	2	3	4	5	
1	1	1	4	4	4	5			5	5		
	2	2	2	1	1	1			3	3		
		3	3	3	2	2			2	4		

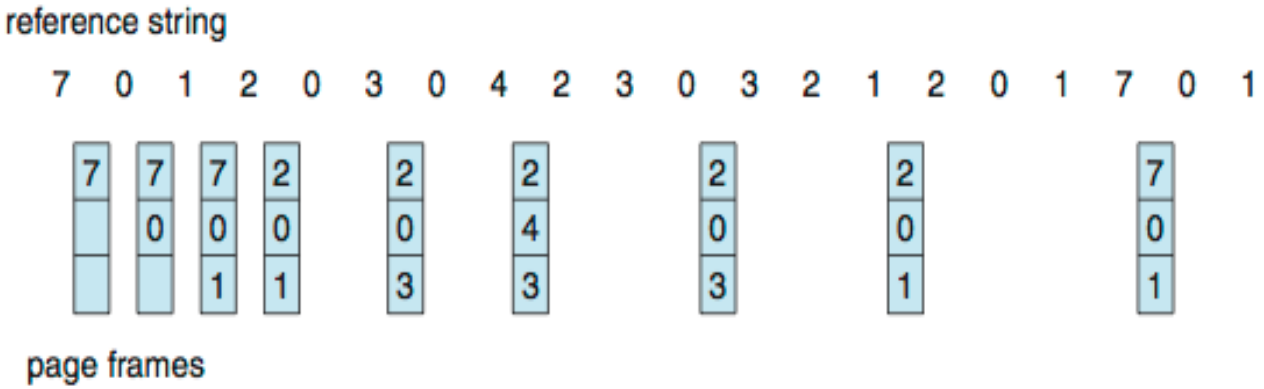
1	2	3	4	1	2	5	1	2	3	4	5	
1	1	1	1			5	5	5	5	4	4	
	2	2	2			2	1	1	1	1	5	
		3	3			3	3	2	2	2	2	
			4			4	4	4	3	3	3	

FIFO Illustrating Belady's Anomaly



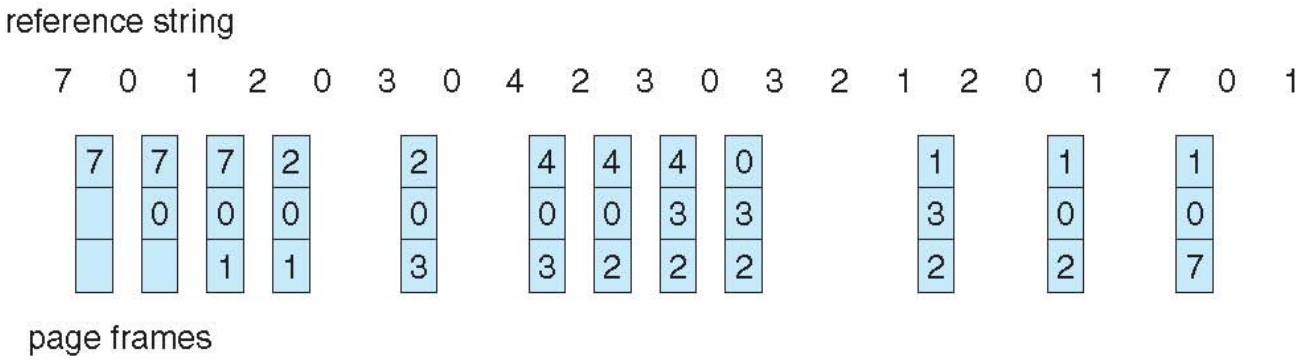
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



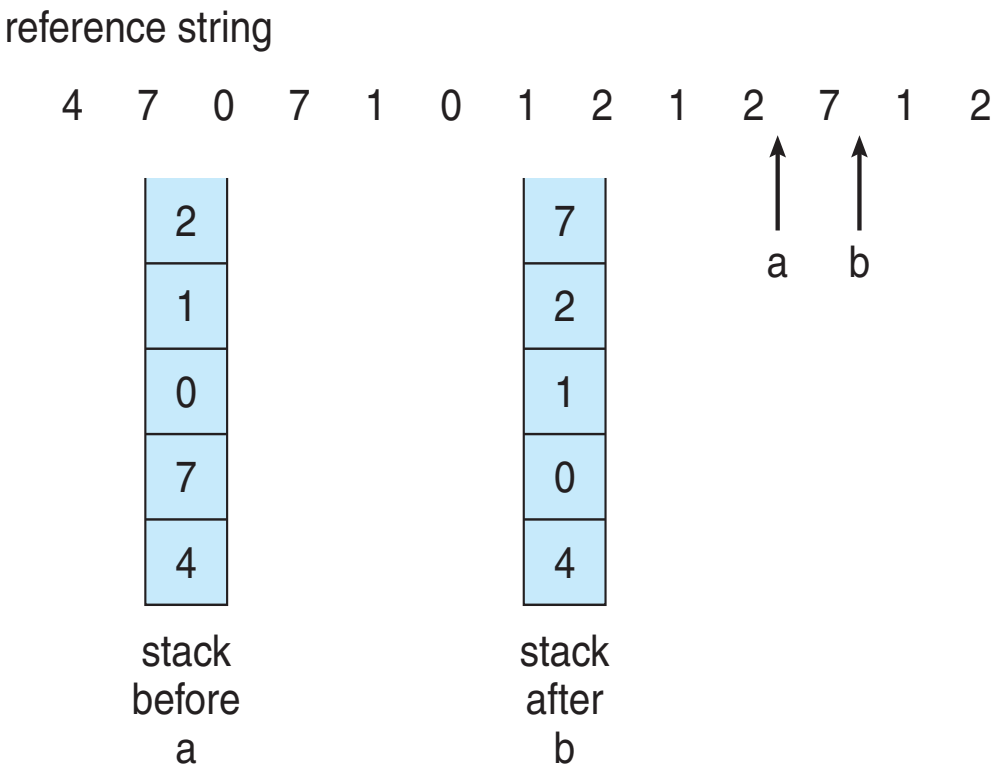
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Use Of A Stack to Record Most Recent Page References



LRU Algorithm (Cont.)

- Use Of A Stack to Record Most Recent Page References

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

Sub QUIZ 2

1,2,3,4,5

1	2	3	4	5
1	1	1	4	x
	2	2	2	y
		3	3	z

FIFO

1,2,3,2,5

1	2	3	2	5
1	1	1	1	x
	2	2	2	y
		3	3	z

LRU

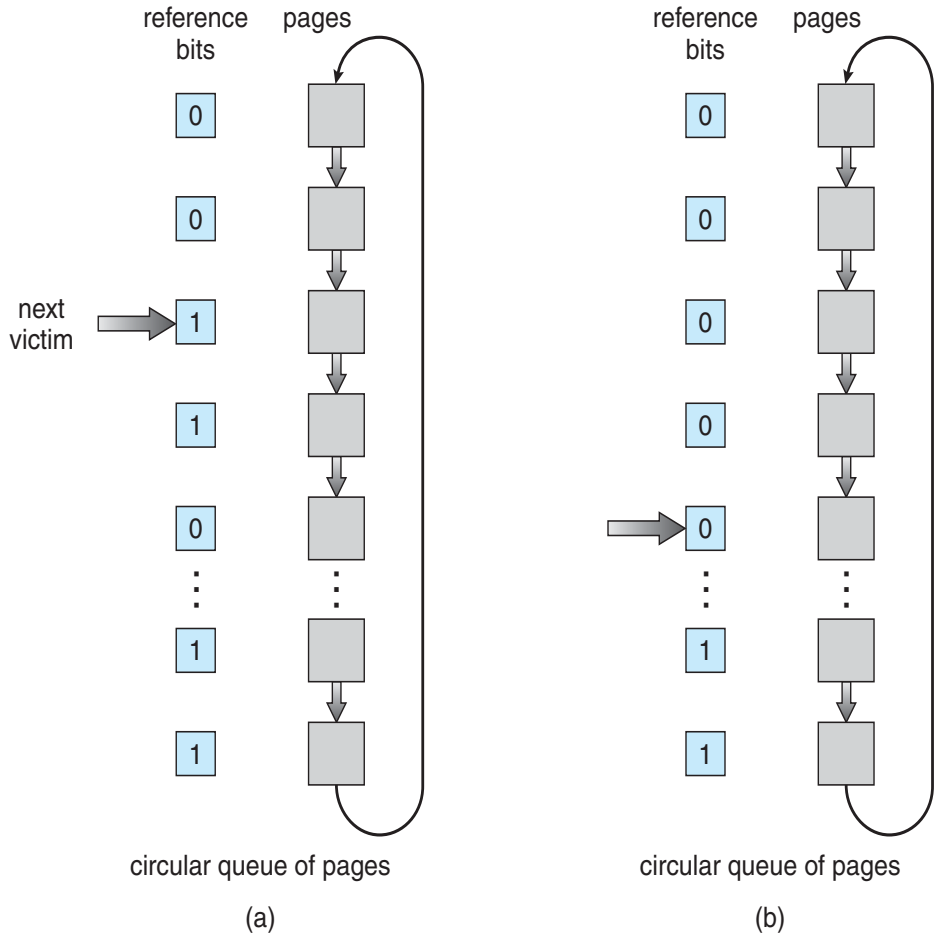
LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however

LRU Approximation Algorithms (cont.)

- เนื่องจาก **implementation** ของ **stack** ใช้เวลามาก จึงใช้วิธีประมาณ **LRU** แทน
- **Many systems provide some help, however, in the form of a reference bit.**
- *The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).*
- **Reference bits are associated with each entry ~~in page table~~ of physical memory frame. (Ref bit** เป็นคุณสมบัติของ **Frame** ในหน่วยความจำจริงแต่ละ **Frame** เมื่อมีการอ่านหรือเขียน **Page** ใน **Frame** นั้น **Ref bit** จะถูก **Set** ให้เป็น **1)**
  - เริ่มต้น **= 0**
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-chance Algorithm



Second-chance Algorithm

1. Initial มีเพจเต็มทั้ง 8 frames

0	0								
1	0								
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								

Second-chance Algorithm

2. CPU refer frames 0,1,2,3,4

0	0	1							
1	0	1							
2	0	1							
3	0	1							
4	0	1							
5	0	0							
6	0	0							
7	0	0							

Second-chance Algorithm

3. Page fault occur
4. Find a victim frame from 0 …

0	0	1	1						
1	0	1	1						
2	0	1	1						
3	0	1	1						
4	0	1	1						
5	0	0	0						
6	0	0	0						
7	0	0	0						

Second-chance Algorithm

3. Page fault occur
4. Find a victim frame from 0 ... 4 เปลี่ยนให้เป็น 0
5. พบ victim frame ที่ 5 เกิด page fault

0	0	1	0						
1	0	1	0						
2	0	1	0						
3	0	1	0						
4	0	1	0						
5	0	0	0						
6	0	0	0						
7	0	0	0						

Second-chance Algorithm

6. page fault ทำงานเสร็จ ref bit 5 set ให้เป็น 1
7. next victim คือ frame 6

0	0	1	0						
1	0	1	0						
2	0	1	0						
3	0	1	0						
4	0	1	0						
5	0	0	1						
6	0	0	0						
7	0	0	0						

Second-chance Algorithm

7. สมมุติว่ามีการอ้างอิง page 6,7,0,1
8. เกิด page fault อีกครั้ง

0	0	1	1						
1	0	1	1						
2	0	1	0						
3	0	1	0						
4	0	1	0						
5	0	0	1						
6	0	0	1						
7	0	0	1						

Second-chance Algorithm

9. จะได้ victim frame = frame 2

0	0	1	0						
1	0	1	0						
2	0	1	0						
3	0	1	0						
4	0	1	0						
5	0	0	1						
6	0	0	0						
7	0	0	0						

Second-chance Algorithm

10. page fault ทำงานเสร็จ ref bit 2 set ให้เป็น 1
11. จะได้ next victim frame = 3

รอจนกว่าจะเกิด page fault อีกครั้ง

แต่ถ้า frame 3 ถูก refer ก่อนก็อาจถูก set reference bit เป็น 1 และอาจไม่เป็น victim frame ในอนาคตก็ได้

0	0	1	0						
1	0	1	0						
2	0	1	1						
3	0	1	0						
4	0	1	0						
5	0	0	1						
6	0	0	0						
7	0	0	0						

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
  - reference bit จะถูก set โดย HW เมื่อมีการอ่านค่าใน frame
  - modify bit จะถูก set โดย HW เมื่อมีการเปลี่ยนแปลงค่าใน frame
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
  - reference bit จะถูก set โดย HW เมื่อมีการอ่านค่าใน frame
  - modify bit จะถูก set โดย HW เมื่อมีการเปลี่ยนแปลงค่าใน frame
- Take ordered pair (reference, modify) (หรือ (r,m)):
  1. จะวน circular list หา (0, 0) ก่อน ถ้าไม่มีก็ไปข้อถัดไป ถ้ามีก็ใช้เป็น victim frame
  2. จะวน circular list หา (0, 1) ถ้าไม่มีก็ไปข้อถัดไป ถ้ามีก็ใช้เป็น victim frame (ต้อง swap ค่าเก่าใน frame ไปเก็บใน swap space)
  3. จะวน circular list หา (1, 0) ถ้าไม่มีก็ไปข้อถัดไป
  4. จะวน circular list หา (1, 1) ถ้ามีก็ใช้เป็น victim frame (ต้อง swap ค่าเก่าใน frame ไปเก็บใน swap space)
- หลังจากอ่านค่าเข้ามาใน victim frame แล้ว ถ้า page fault เพราะอ่านค่า ให้ set (r,m) ของ victim frame เป็น (1,0) ถ้าเขียนค่าให้ set เป็น (0,1)

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- (เป็นการนับจำนวนการ reference ของแต่ละ frame ตั้งแต่นำเข้ามา)
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**:
  - Replaces page with smallest count
  - แทนที่ page ที่ถูกอ้างอิงน้อยที่วุด
- **Most Frequently Used** (**MFU**) **Algorithm**:
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used
  - แทนที่ page ที่ถูกอ้างอิงมากที่สุด เพราะ page ที่มีค่าอ้างอิงน้อย น่าจะเพิ่งถูก swap in เข้ามา

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Page-Buffering Algorithms

- Page Buffering Algorithm คือการกลุ่ม (pool) ของ free frame ไว้ล่วงหน้าจำนวนหนึ่งก่อนที่จะเกิด page fault เพื่อให้ OS สามารถอ่านค่าเพจใหม่เข้ามาใน free frame ได้เลย
- เลือกเพจจำนวนหนึ่งที่จะนำมาใช้ free frame

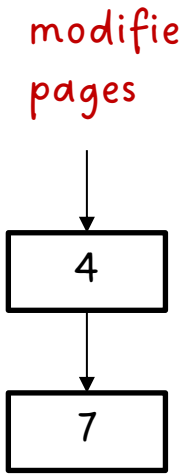
frame	Page
0	P0 pg1
1	P0 pg0
2	P1 pg1
3	P0 pg5
4	P1 pg2
5	P1 pg3
6	P0 pg3
7	P0 pg8

1. ก่อนมี pf สร้าง free pool
2. เลือก victim ใส่ใน free pool สาม frames
Frames 7, 2, 4 (ถ้า p1 pg1, p1 pg2, p0 pg3 dirty ก็ เขียนลง swap space)
3. Update page tables of p0 & p1
4. เมื่อเกิด pf สามารถอ่าน page ใหม่เข้า free frame ได้เลย ทาให้สามารถ pf ได้เร็ว (ไม่ต้องรอ ให้เขียนลง swap แล้วค่อยอ่าน page ใหม่เข้ามา ณ เวลาที่เกิด pf นั้น)

Page-Buffering Algorithms

- ในระหว่างที่สร้าง pool ของ free frame ก็ให้สร้าง list ของ frame ที่ถูก modified ไว้ด้วย (modified page list ในภาพ)
- ให้ OS ทยอยเขียน frame ใน modified page list ลงสู่ swap space และ set ให้ dirty bit ของ frame เหล่านั้นเป็น 0

frame	Page
0	P0 pg1
1	P0 pg0
2	P1 pg1
3	P0 pg5
4	P1 pg2
5	P1 pg3
6	P0 pg3
7	P0 pg8



1. ก่อนมี pf สร้าง list ของ modify pages
2. พอ I/O ว่างงาน ก็เขียน swap space
3. update page table ของ p0 และ p1 ให้ dirty bit ของ p1 pg1 p1 pg2 p0 pg8 เป็น 0 (จะได้ไม่ต้องเขียนลง swap space ตอน pf)
4. เมื่อเขียน List ของ modified page ลง swap Space แล้ว frame เหล่านี้ก็เป็น free frame ได้

Page-Buffering Algorithms

- ในกรณีที่มีการอ้างอิงข้อมูลใน frame ที่ถูกทำให้เป็น free frame แล้ว (แต่ยังไม่ได้ถูกเอาไปใช้) page buffering algorithm ก็สามารถคืน frame นั้นให้กับ page table ของ process ที่ใช้งาน frame นั้นก่อนหน้าได้

frame	Page
0	P0 pg1
1	P0 pg0
2	P1 pg1
3	P0 pg5
4	P1 pg2
5	P1 pg3
6	P0 pg3
7	P0 pg8

frame	Page
0	P0 pg1
1	P0 pg0
2	P1 pg1
3	P0 pg5
4	P1 pg2
5	P1 pg3
6	P0 pg3
7	P0 pg8

1. ไม่ต้องลบเนื้อหาใน free frames
2. มีการใช้ frame 2 ก่อน pf

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- **Proportional allocation** – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly

    เวลาที่ process หนึ่งประมวลผลแต่ละครั้งอาจต่างกัน ขึ้นอยู่กับจำนวน frame ที่มีให้ใช้
  - But greater throughput so more common

    ใช้แบบนี้กันมาก เพราะทำให้อัตราการจบการประมวลผลของ process สูง
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance

    เวลาในการรัน process แต่ละครั้งจะสม่ำเสมอและใกล้เคียงกัน
  - But possibly underutilized memory

    การใช้งาน memory อาจไม่คุ้มค่าเพราะอาจมี frame ที่ไม่ถูกใช้งาน
  - อะไรคือข้อดีและข้อเสียของ global และ local replacement

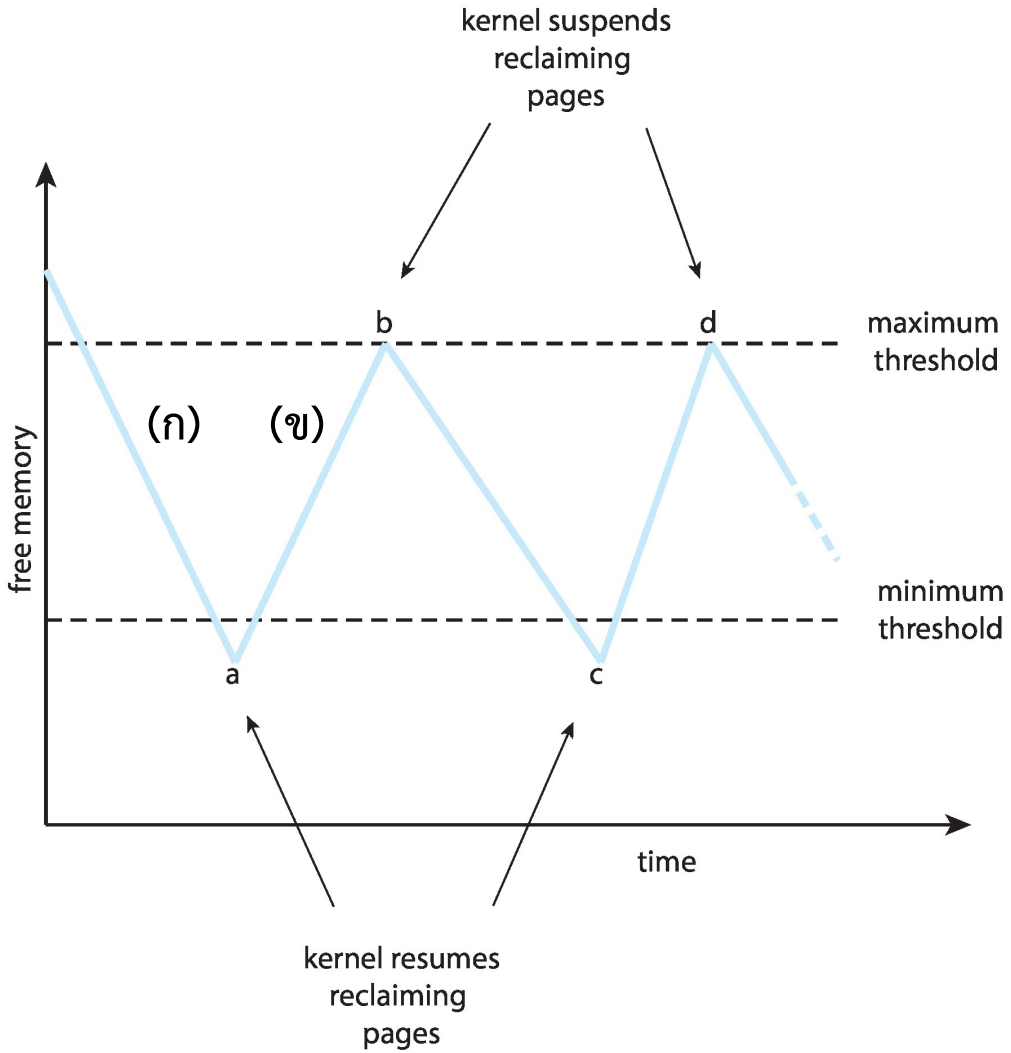
Reclaiming Pages

- A strategy to implement global page-replacement policy

  เป็นกลยุทธในการแทนที่ page แบบ global
- OS จะ reclaim pages โดยใช้ page buffering algorithm สร้าง free-frame list ไว้ก่อนที่จะมีการแทนที่จริงเมื่อเกิดเหตุการณ์ page fault
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.

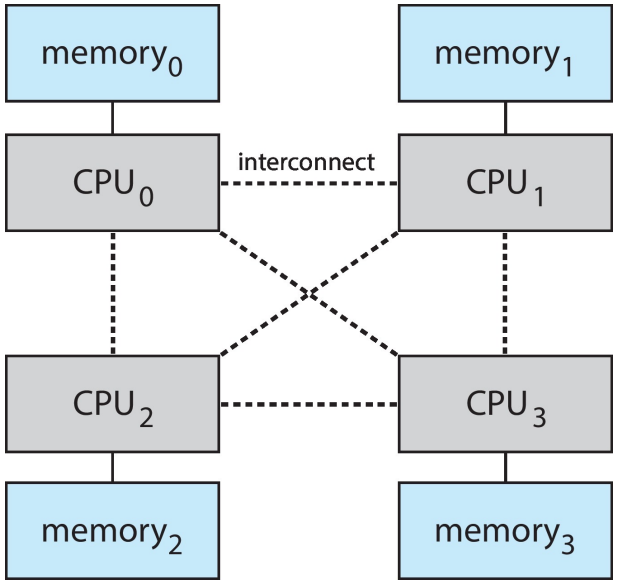
  Page buffering algor ที่ใช้นโยบายนี้จะดำเนินการ reclaim page และเพิ่ม frame เข้าไปใน free-frame list เมื่อจำนวน frame ใน list ต่ำกว่าค่า Threshold
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example



Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Linux มี free frame list แยกสำหรับแต่ละ NUMA node

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
      - เมื่อการใช้งาน CPU ต่ำเนื่องจากมีการทำ page fault บ่อย ทำให้ OS เพิ่มจำนวน process เพื่อหวังจะเพิ่ม CPU utilization
      - เข้าใจผิดอย่างแรง
    - Another process added to the system
      - เมื่อ เพิ่ม process ทีให้เกิด page fault เพิ่มขึ้นอีก

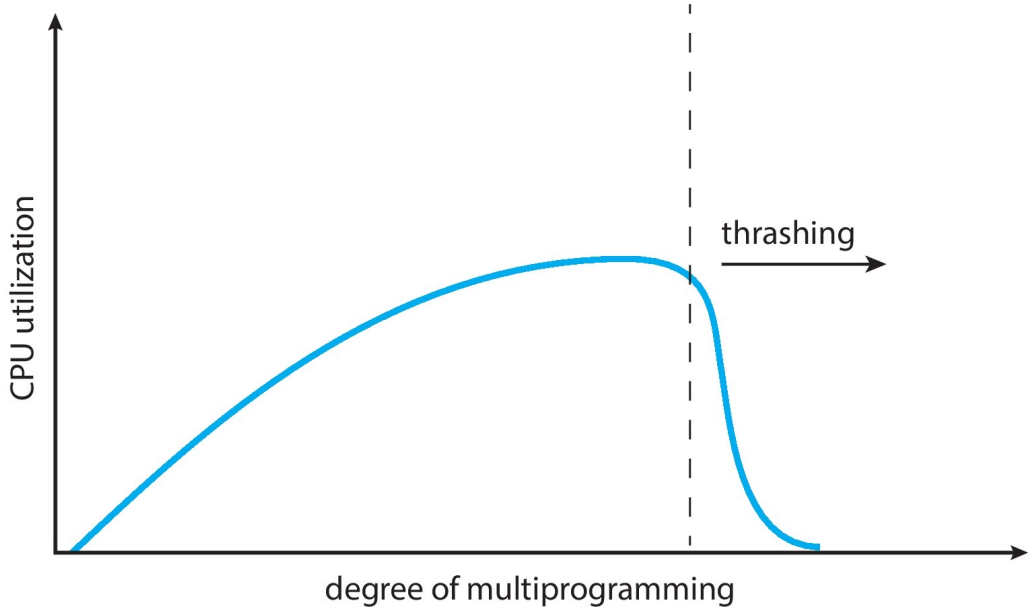
ตัวอย่าง Trashing

frame	Page
0	P0 pg1
1	P0 pg0
2	P1 pg1
3	P0 pg5
4	P1 pg2
5	P1 pg3
6	P0 pg3
7	P0 pg8

- สมมุติ P0 ต้องการ Pg 9 และ OS โหลดมันเข้ามาใน frame 2 ส่งผลให้ Pg1 ของ P1 ถูกแทนที่
- แต่หลังจากนั้น ปรากฏว่า P1 ต้องการ Pg 3 ทำให้ OS ต้องโหลดมันขึ้นมาใน frame 4 ซึ่งทำให้ Pg2 ของ P1 ถูกแทนที่
- แต่หลังจากนั้น P1 กลับต้องการค่าจาก Pg2 ขึ้นมา OS เลยต้องโหลด Pg2 ของ P1 ไปไว้ใน frame 7
- การอ้างอิง memory ของ Process ที่อ้างอิง page ที่เพิ่งถูกนำออกไปจาก physical memory ซ้ำๆแบบนี้ทำให้เกิด trashing

Thrashing (Cont.)

- **Thrashing**. A process is busy swapping pages in and out



Demand Paging and Thrashing

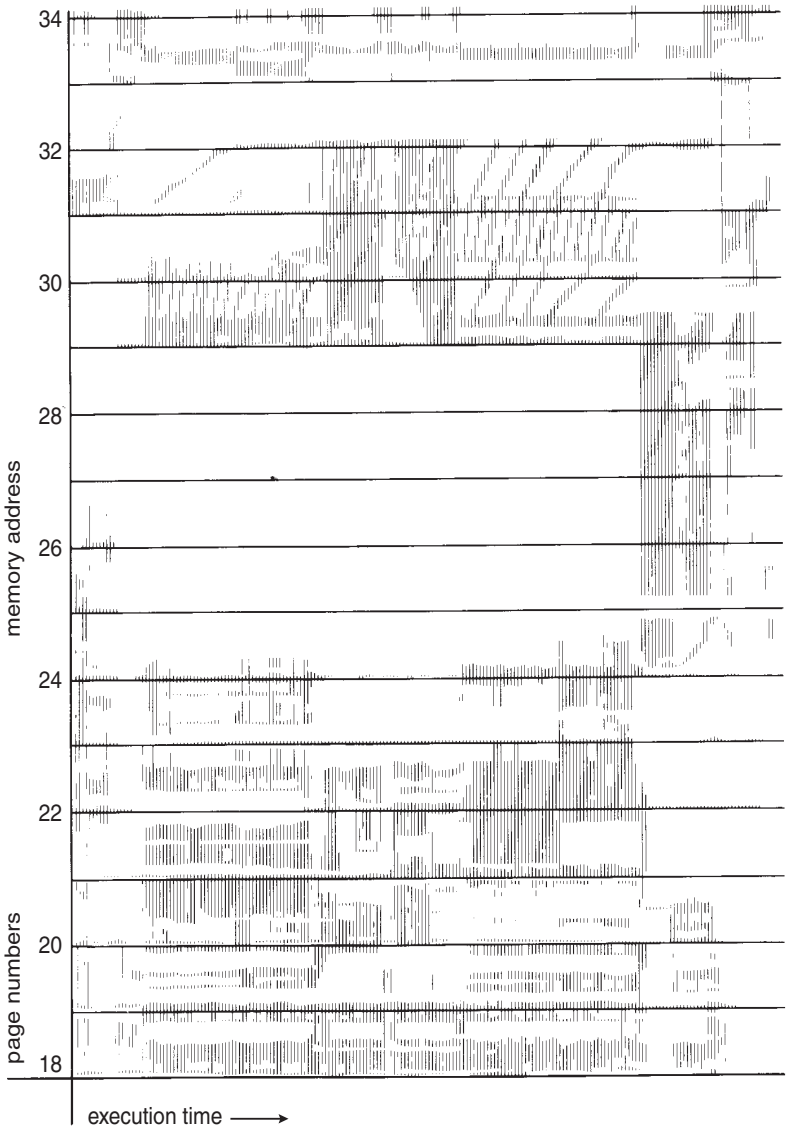
- Why does demand paging work?

  Locality model

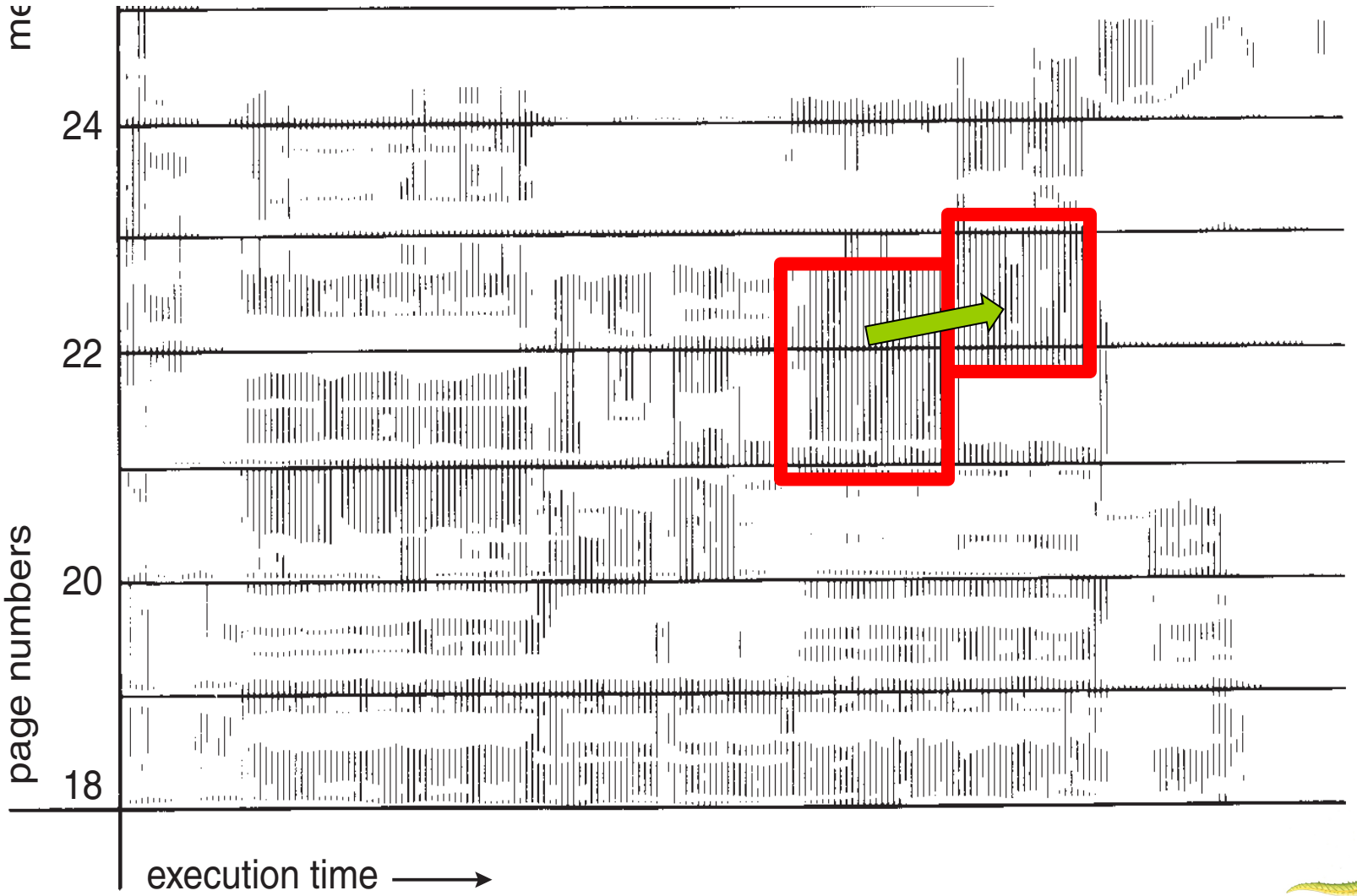
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size
- Limit effects by using local or priority page replacement

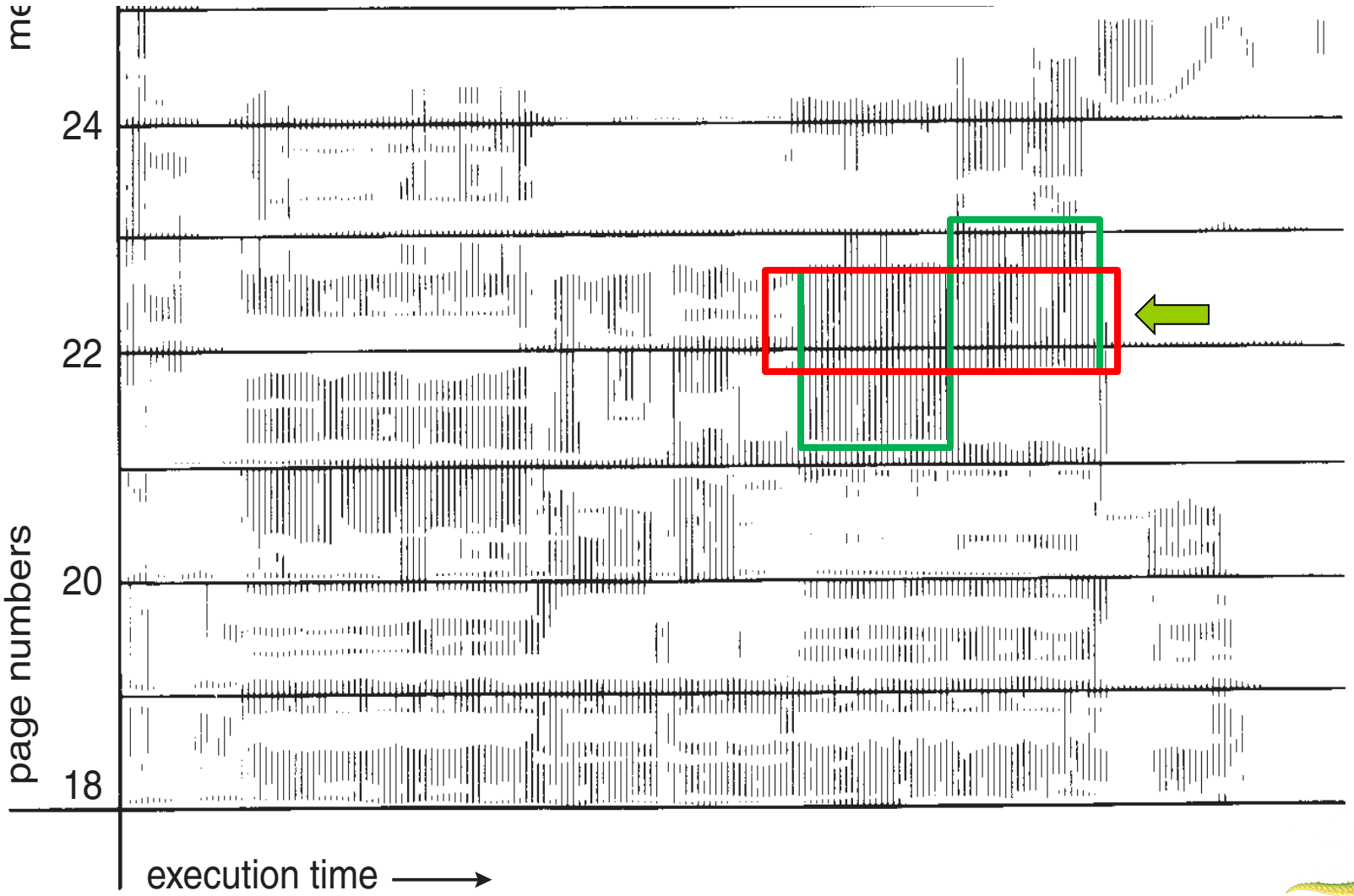
Locality In A Memory-Reference Pattern



Locality Migration



Locality Overlap

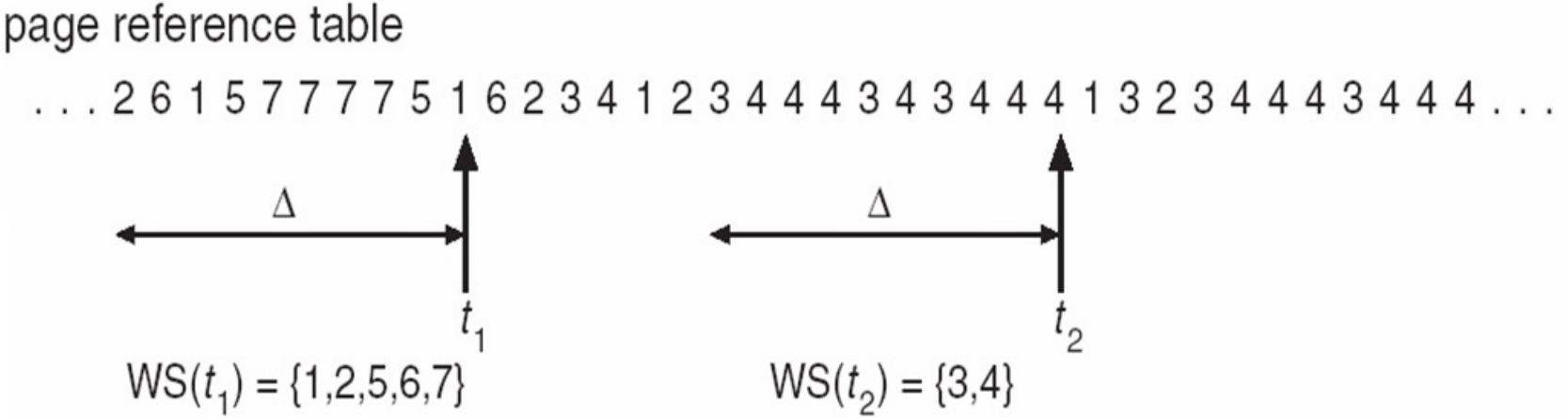


Working-Set Model

- The working set is an approximation of the program's locality.
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

Working-Set Model (Cont.)

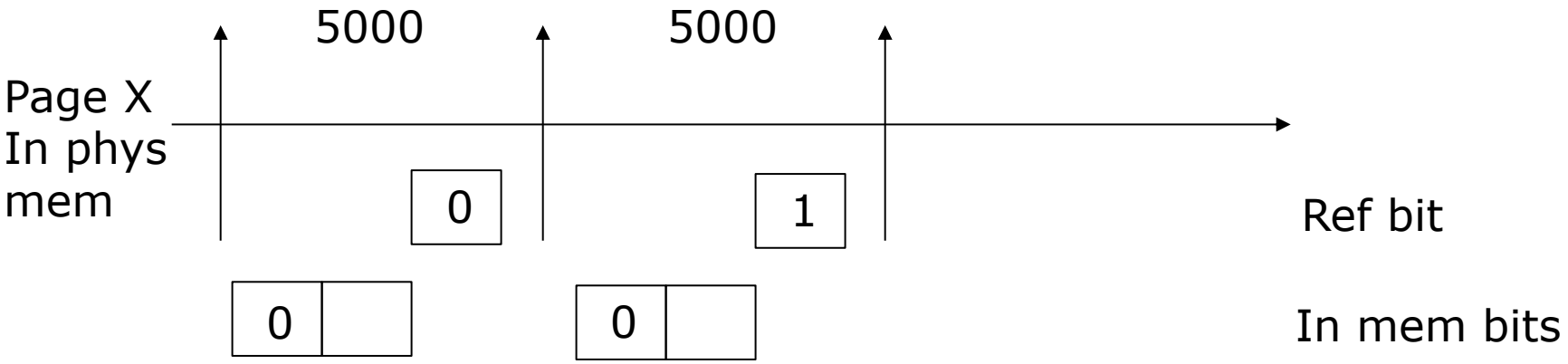
- if $D > m \Rightarrow$ Thrashing
- ถ้าขนาดรวมของ working set ของทุก process มากกว่าขนาดของ memory ก็จะเกิด trashing
- Policy if $D >$ m, then suspend or swap out one of the processes
- สามารถแก้ปัญหานี้ได้โดย หยุดบาง process ชั่วคราว หรือ swap เนื้อหาของบาง process ออกไป



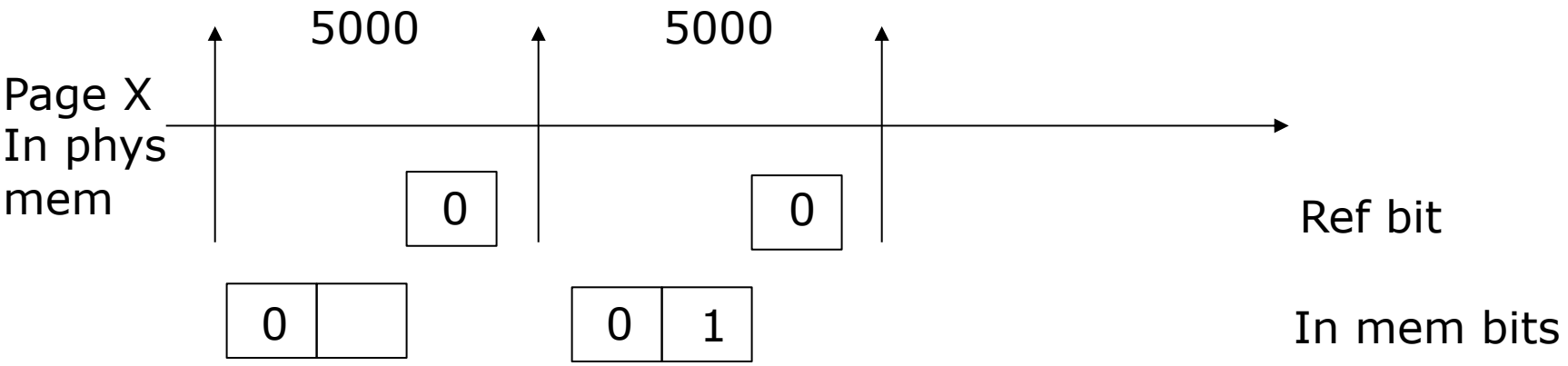
Keeping Track of the Working Set

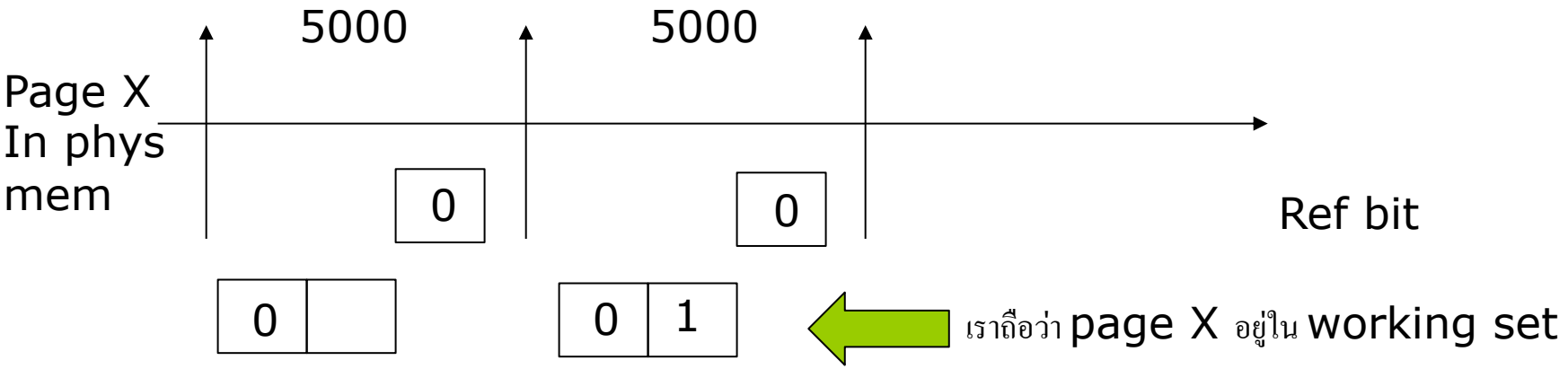
- Approximate with interval timer + a reference bit
- OS ประมาณว่า Page ใดอยู่ใน Working Set บ้างโดยใช้ interval timer และ reference bit ดังตัวอย่างต่อไปนี้
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
  - วิธีนี้ช่วยตรวจจับการอ้างอิงเพจได้ในช่วง 10,000 time unit
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units
- การเพิ่มจำนวน In Memory bits เป็น 10 และ Interrupt ทุก 1000 time units ทำให้ record การอ้างอิงได้ละเอียดขึ้น ทำให้เราสามารถตรงจได้ว่าในช่วงเวลาย่อยใดที่มีการอ้างอิง

Keeping Track of the Working Set



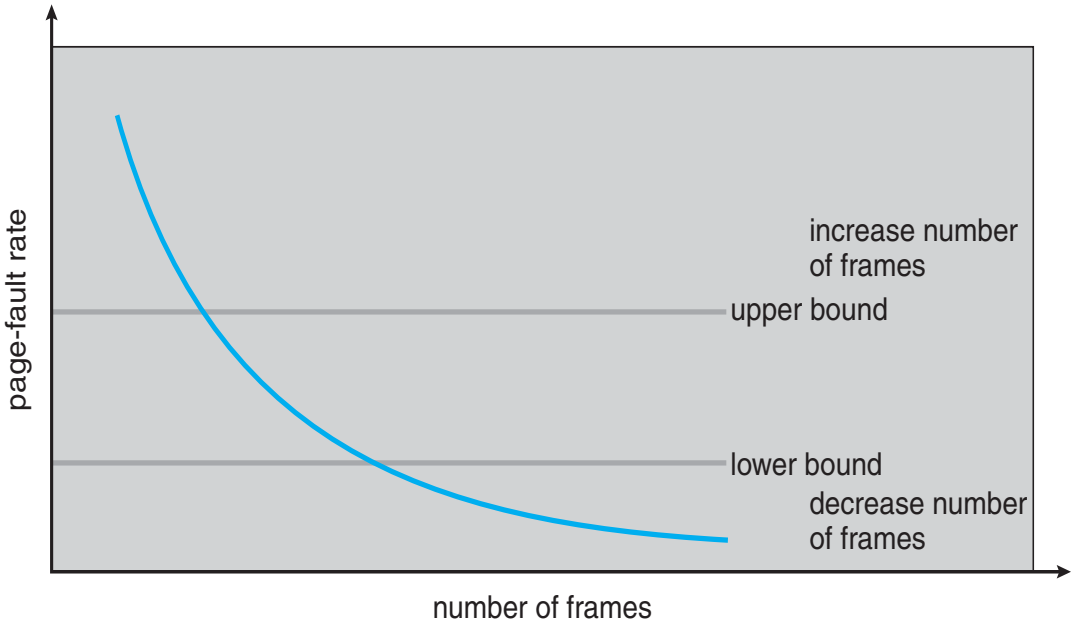
Keeping Track of the Working Set





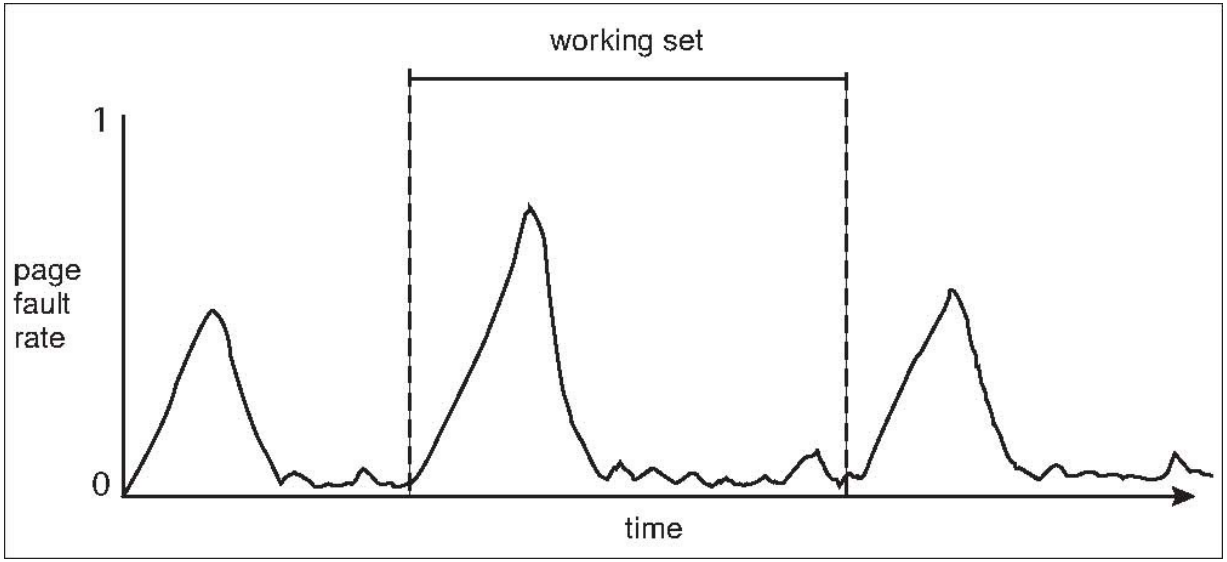
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Other Considerations

- Prepaging
- Page size
- TLB reach
- Program structure
- I/O interlock and page locking

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and *α* of the pages is used
  - Is cost of ***s * α*** save pages faults > or < than the cost of prepaging ***s * (1- α)*** unnecessary pages*?*
  - ***α*** near zero $\Rightarrow$ prepaging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

Program Structure

- Program 1

```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
          data[i,j] = 0;
```

1 Page

128 x 128 = 16,384 page faults

0,0	0,1	0,2	…	…	…	0,126	0,127
1,0	1,1	1,2	…	…	…	1,126	1,127
2,0	2,1	2,2	…	…	…	2,126	2,127
126,0	126,1	126,2	…	…	…	126,126	126,127
127,0	127,1	127,2	…	…	…	127,126	127,127

Program Structure

- Program structure
  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

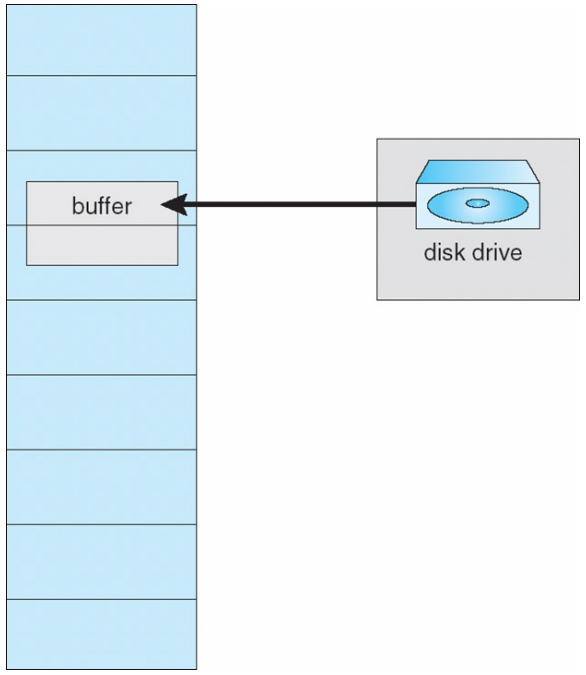
128 page faults

1 Page

0,0	0,1	0,2	…	…	…	0,126	0,127
1,0	1,1	1,2	…	…	…	1,126	1,127
2,0	2,1	2,2	…	…	…	2,126	2,127
126,0	126,1	126,2	…	…	…	126,126	126,127
127,0	127,1	127,2	…	…	…	127,126	127,127

I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



การจัดการ Memory ของ Linux Kernel

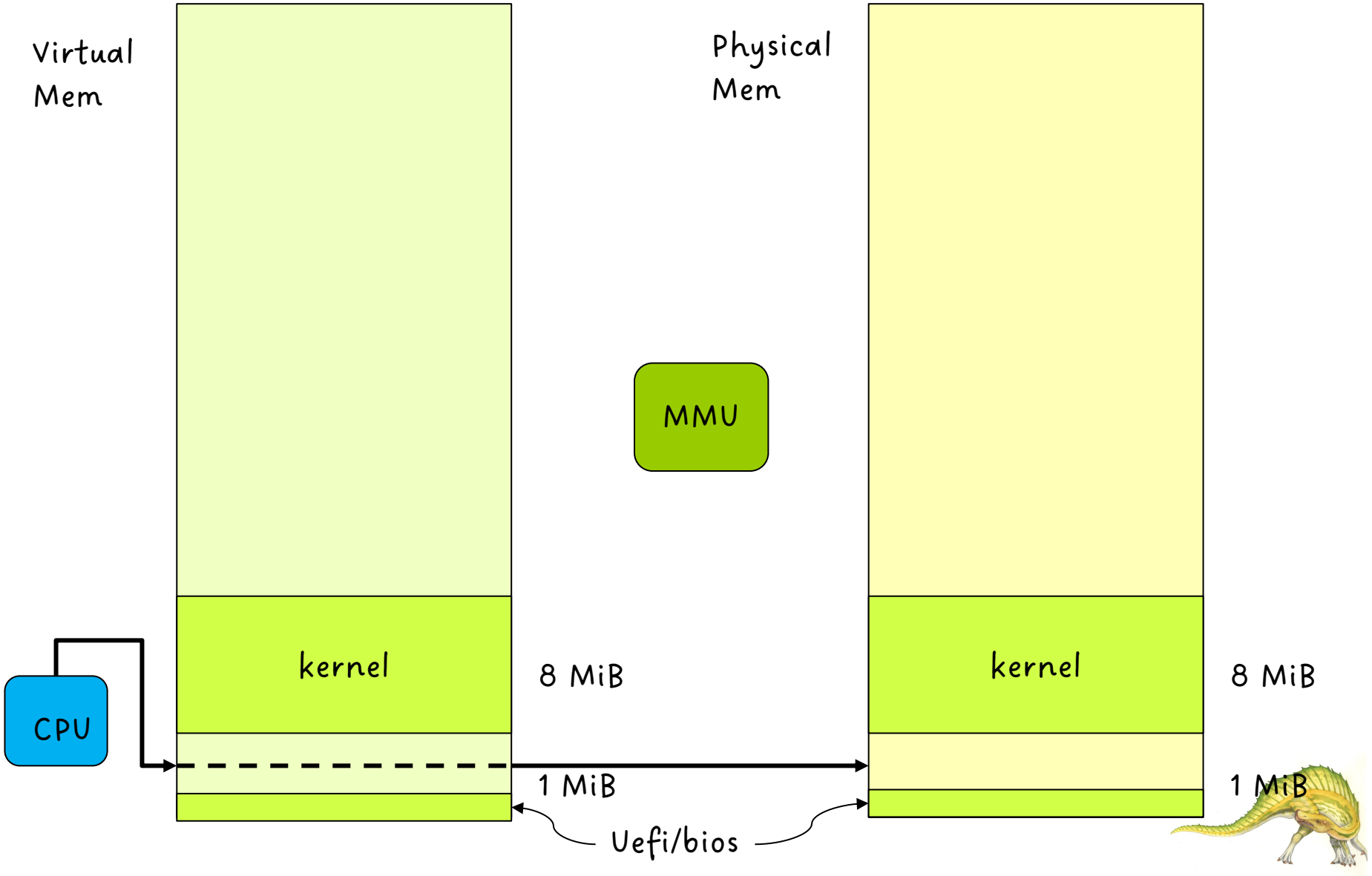
UEFI load Kernel to Memory

- เมื่อคอมพิวเตอร์เริ่มต้นการประมวลผล ซีพียูจะทำงานอยู่ใน Real Mode ซึ่งรันโปรแกรมเดียวและเข้าถึงหน่วยความจำ Physical Memory โดยตรง และการอ้างอิง Memory Address จะเป็นการอ้างอิง Physical Memory Address
- ซีพียูจะโหลดโปรแกรม UEFI เข้าสู่หน่วยความจำและประมวลผล UEFI
- โปรแกรม UEFI จะโหลดโปรแกรม OS Kernel เข้าสู่ Physical Memory
- ในกรณีที่มี Grub UEFI จะโหลด Grub แล้ว Grub โหลด OS Kernel เข้าสู่ Memory
- Linux Kernel เริ่มประมวลผล Boostrapping

https://www.kernel.org/doc/gorman/html/understand/understand006.html

memory - How linux kernel create, initialize and setup page table for user process? - Stack Overflow

UEFI/BIOS


Virtual Mem
Physical Mem
MMU
kernel
8 MiB
kernel
8 MiB
CPU
1 MiB
1 MiB
Uefi/bios

ขั้นตอน 1. Bootstraping (1)

- เมื่อ OS Kernel เริ่มการประมวลผล เนื้อหาของเคอร์เนิลจะอยู่ในที่ตำแหน่ง 1 – 9 MiB (ขนาด 8MiB) ของ Physical Memory
- ในโค๊ดของ kernel ค่า memory address ของ variables และ routines จะเป็น virtual address (vaddr) ที่เท่ากับตำแหน่งของ physical memory ที่มันจะถูกโหลดเข้ามาอยู่ + ค่าคงที่ (Page_Offset สมมุติว่า = 4096)
- ถัดจากนั้น Kernel ทำการ enable paging (กำหนดค่าใน CR0 register)
- ในขั้นตอนนี้ เมื่อ CPU รันโค๊ดของ kernel MMU จะแปลง virtual address ให้เป็น physical address โดยใช้วิธีการ Map โดยตรง (Direct Mapping) คือ

phyaddr = vaddr – Page_Offset

https://www.kernel.org/doc/gorman/html/understand/understand006.html

memory - How linux kernel create, initialize and setup page table for user process? - Stack Overflow

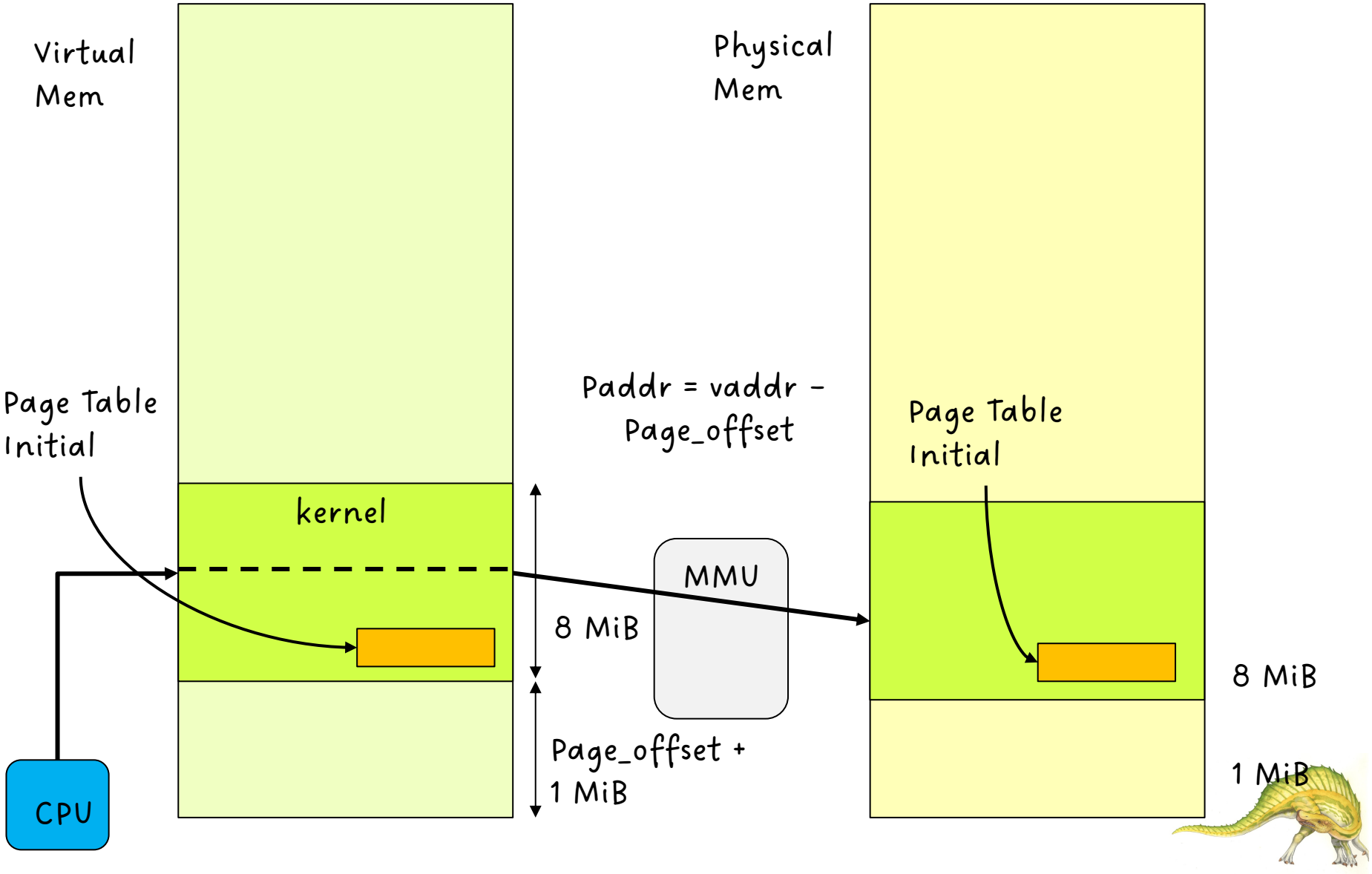
ขั้นตอน 1. Bootstraping (2)

- Bootstrapping code ของ OS Kernel จะสร้าง Page Table ขั้นต้นขึ้น ซึ่งเมื่อ Kernel enable paging ในอนาคต MMU จะมีข้อมูลใน Page Table สำหรับแปลง vaddr ให้เป็น paddr ที่เก็บโปรแกรมและ data ของ kernel ในพื้นที่ 8 MiB ที่ kernel อยู่ใน Physical Memory ได้

https://www.kernel.org/doc/gorman/html/understand/understand006.html

memory - How linux kernel create, initialize and setup page table for user process? - Stack Overflow

Bootstrapping



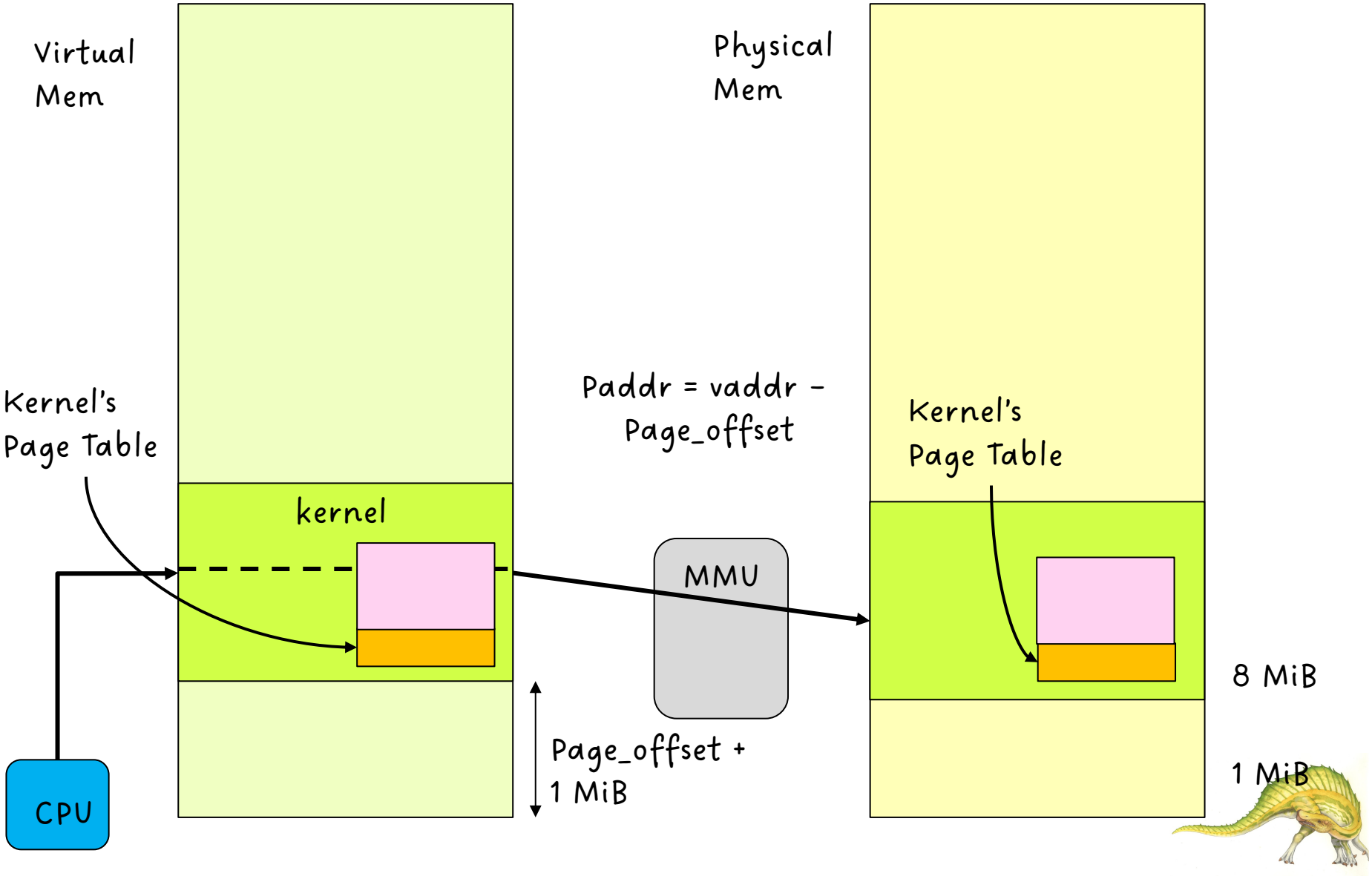
ขั้นตอน 2. Finalising Paging

- OS kernel (ใช้การแปลง vaddr เป็น paddr แบบ Direct Mapping) เรียกฟังก์ชั่น paging_init() เพื่อ
  - สร้าง Page Table ต่อให้สมบูรณ์ โดยสร้าง page table entries (PTE) สำหรับ ZONE_DATA และ ZONE_NORMAL ของเคอร์เนิล
  - กำหนดให้ทุก Process เห็น PTE ของ kernel
  - สร้าง PTE สำหรับ map kernel ไปที่ High Virtual Memory Address
- OS Kernel กำหนดให้ MMU เริ่มใช้งาน page table กำหนดให้ CR3 ชี้ไปยัง Page Table ของ Kernel
- OS Kernel กำหนดค่าใน PTE ของมันเพื่อให้มันเข้าถึงได้เท่านั้น

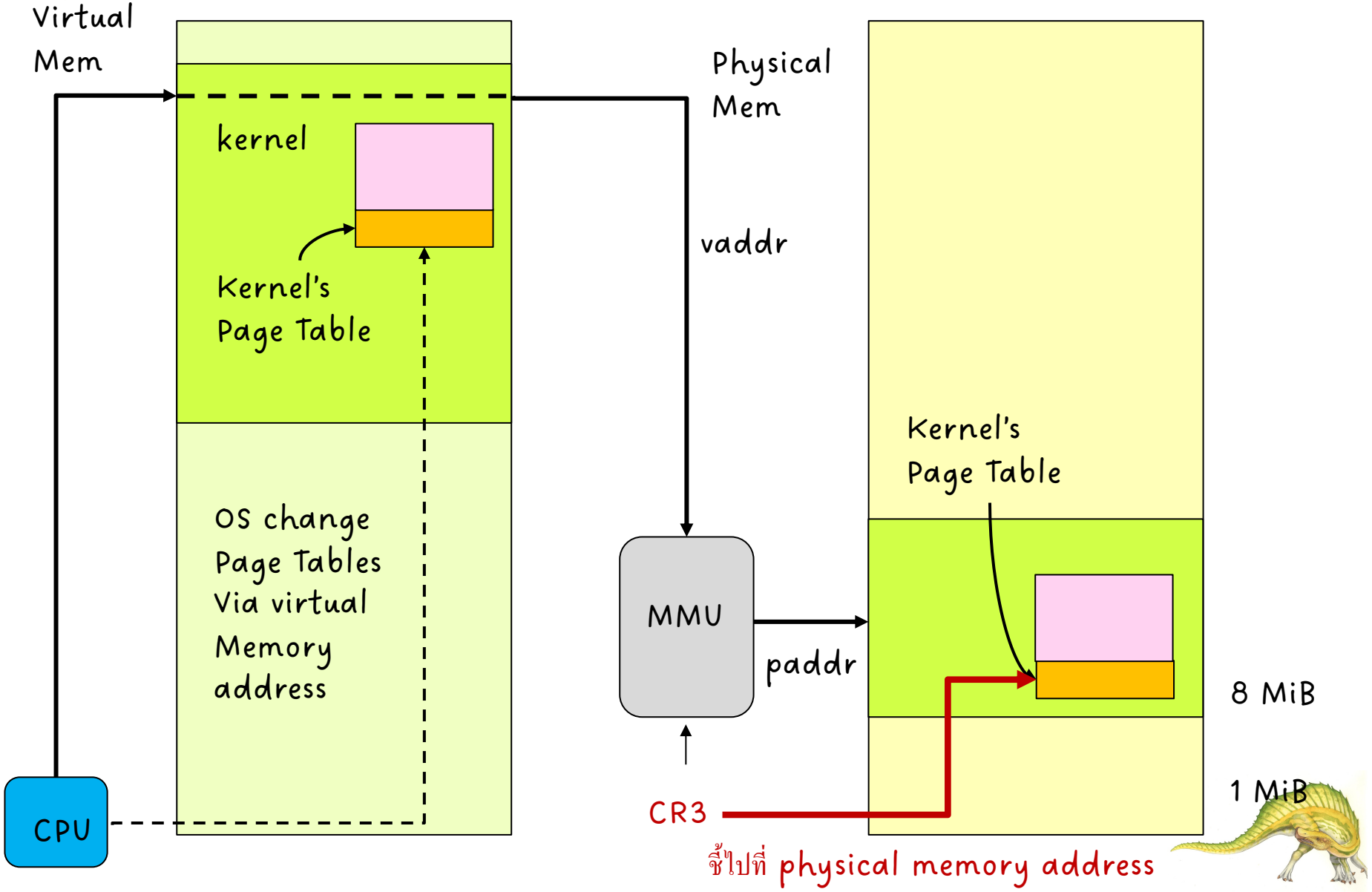
https://www.kernel.org/doc/gorman/html/understand/understand006.html

memory - How linux kernel create, initialize and setup page table for user process? - Stack Overflow

Finalize Kernel Paging



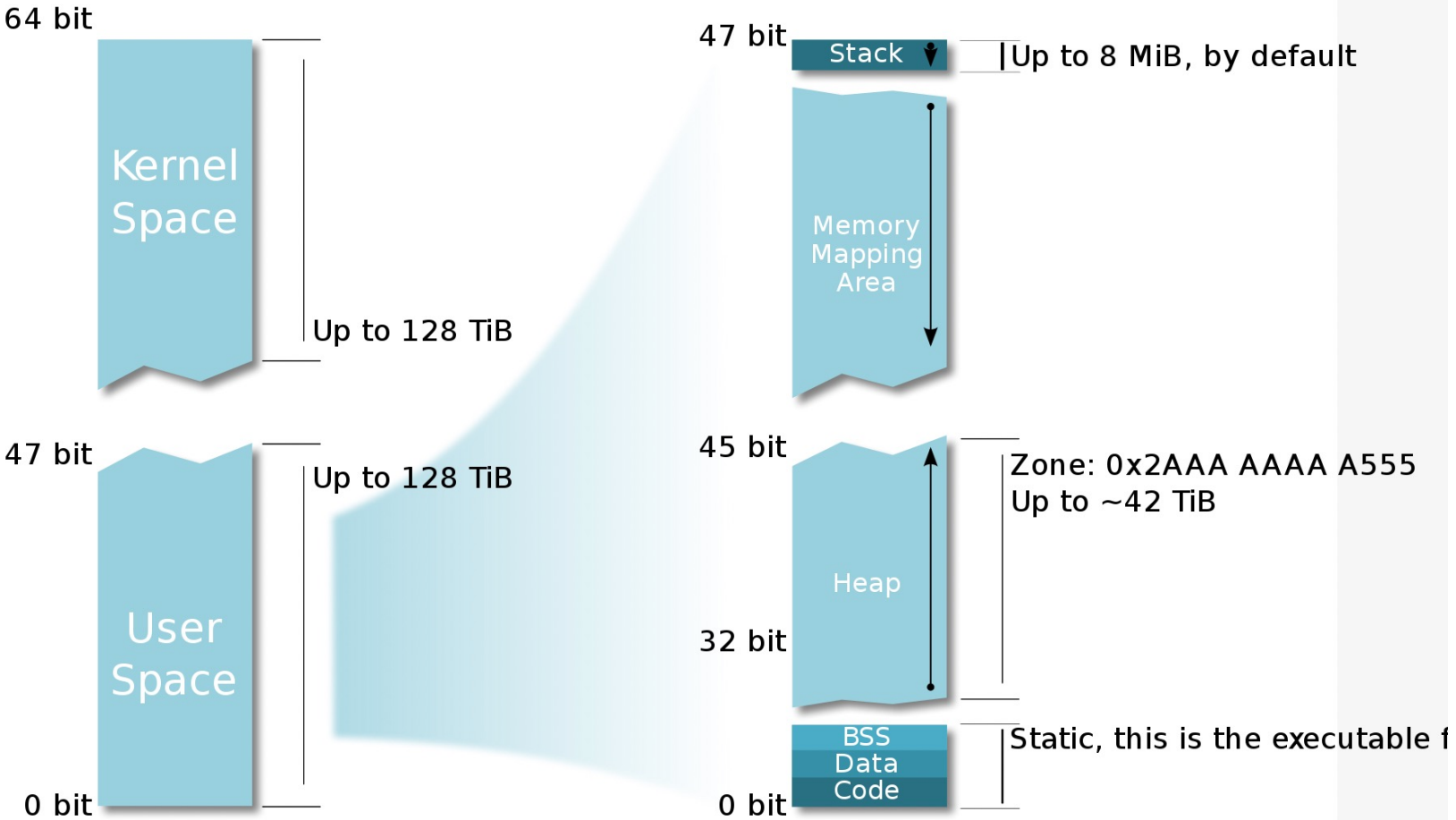
After Finalize Paging



Linux Virtual Memory Layout

- เมื่อเริ่มต้นการทำงาน Linux จะ load kernel เข้าสู่ main memory ใน high logical memory address และจองพื้นที่นั้นไว้ใช้ตลอดเวลา
- Kernel's logical memory ตั้งแต่ `ffff800000000000` ถึง `ffffffffffffffff`
- User's logical memory ตังแต่ `0000000000000000` ถึง `00007fffffffffff`
- OS สร้างและจัดการ page tables ใน kernel memory space

Linux Memory Layout (64 bits)



So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!

https://en.wikibooks.org/wiki/The_Linux_Kernel/Memory#:~:text=Linux%20kernels%20split%20the%204GB,final%201GB%20starting%20at%200xc0000000.

Linux Memory Layout (64 bits)

17 bits=0

47 bits

Start addr	Offset	End addr	Size	VM area description
0000000000000000	0	00007fffffffffff	128 TB	user-space virtual memory, different per mm
0000800000000000	+128 TB	ffff7fffffffffff	~16M TB	... huge, almost 64 bits wide hole of non-canonical virtual memory addresses up to the -128 TB starting offset of kernel mappings.
				Kernel-space virtual memory, shared between all process
ffff800000000000	-128 TB	ffff87ffffffffff	8 TB	... guard hole, also reserved for hypervisor
ffff880000000000	-120 TB	ffff887fffffffff	0.5 TB	LDT remap for PTI
ffff888000000000	-119.5 TB	ffffc87fffffffff	64 TB	direct mapping of all physical memory (page_offset_base
ffffc88000000000	-55.5 TB	ffffc8ffffffffff	0.5 TB	... unused hole
ffffc90000000000	-55 TB	ffffe8ffffffffff	32 TB	vmalloc/ioremap space (vmalloc_base)
ffffe90000000000	-23 TB	ffffe9ffffffffff	1 TB	... unused hole
ffffea0000000000	-22 TB	ffffeaffffffffff	1 TB	virtual memory map (vmemmap_base)
ffffeb0000000000	-21 TB	ffffebffffffffff	1 TB	... unused hole
ffffec0000000000	-20 TB	fffffbffffffffff	16 TB	KASAN shadow memory

17 bits=1

47 bits

				Identical layout to the 56-bit one from here on:
ffffc0000000000	-4	TB	fffffdffffffffff	2 TB	... unused hole vaddr_end for KASLR
fffffe0000000000	-2	TB	fffffe7fffffffff	0.5 TB	cpu_entry_area mapping
fffffe8000000000	-1.5	TB	fffffeffffffffff	0.5 TB	... unused hole
ffffff0000000000	-1	TB	ffffff7fffffffff	0.5 TB	%esp fixup stacks
ffffff8000000000	-512	GB	ffffffeeffffffff	444 GB	... unused hole
ffffffef00000000	-68	GB	fffffffeffffffff	64 GB	EFI region mapping space
fffffffff0000000	-4	GB	fffffffff7ffffff	2 GB	... unused hole
fffffffff8000000	-2	GB	fffffffff9ffffff	512 MB	kernel text mapping, mapped to physical address 0
fffffffff8000000	-2048	MB			
fffffffffa0000000	-1536	MB	ffffffffffefffff	1520 MB	module mapping space
ffffffffff000000	-16	MB			
FIXADDR_START	~-11	MB	ffffffffff5fffff	~0.5 MB	kernel-internal fixmap range, variable size and offset
ffffffffff600000	-10	MB	ffffffffff600fff	4 kB	legacy vsyscall ABI
ffffffffffe00000	-2	MB	ffffffffffffffff	2 MB	... unused hole

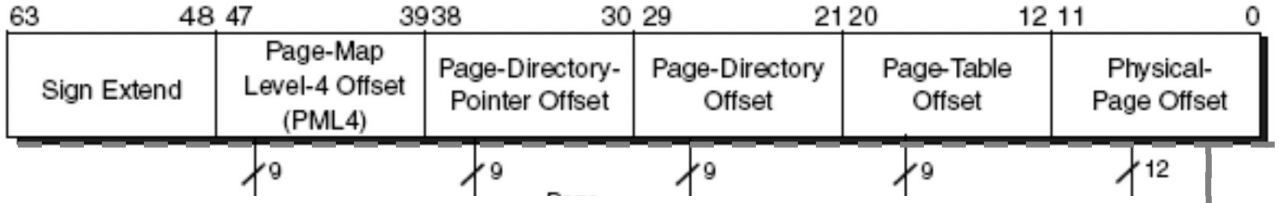
โครงสร้าง Page Table

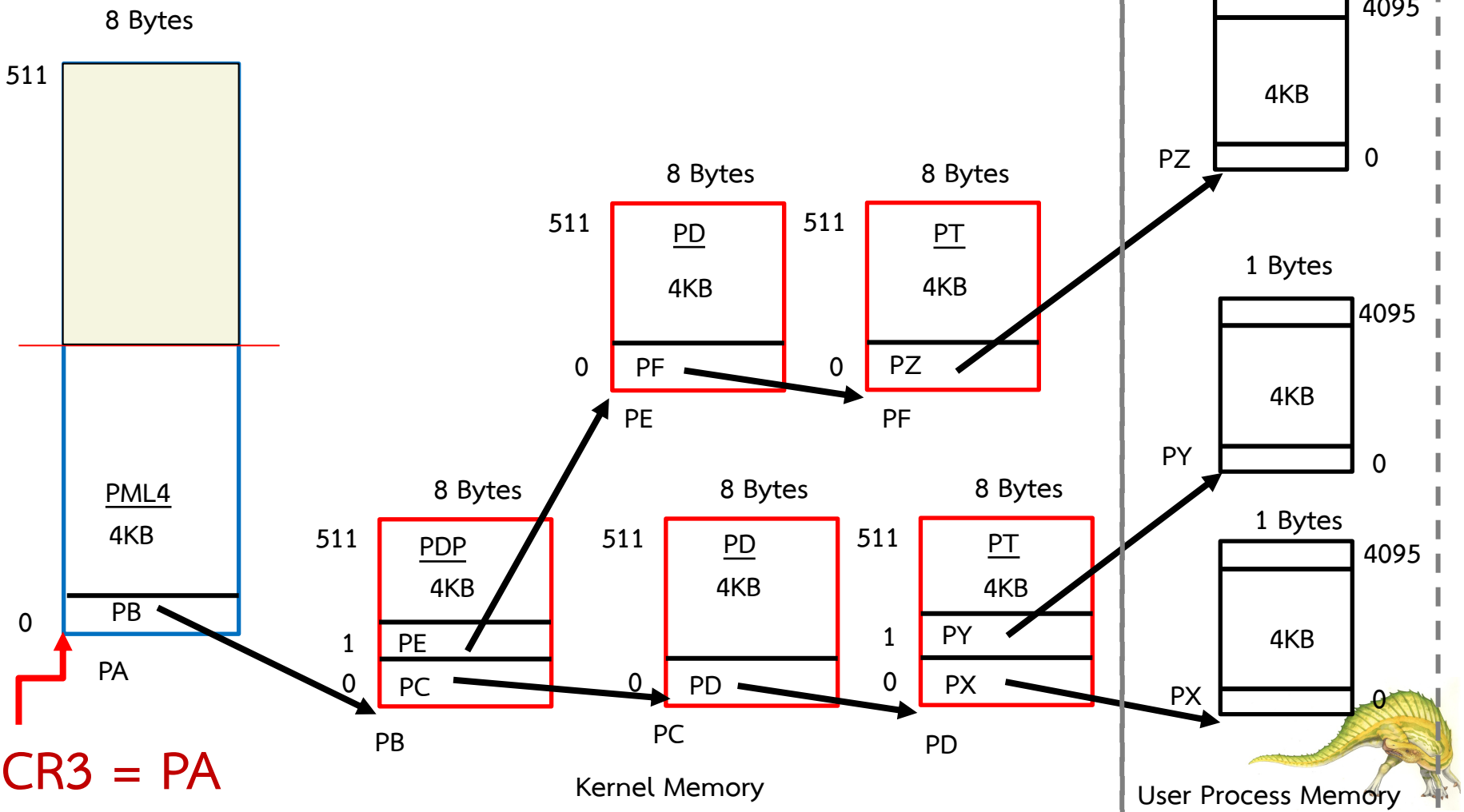
- แบบดั้งเดิม: Traditional ทุก Process จะมี logical memory space ทั้งหมด (รวมทั้งส่วนของ Kernel) แต่ user process จะเข้าถึงได้ใน address ที่เป็น user space เท่านั้น
- แบบใหม่: Kernel Page Table Isolation (KPTI) แยก user page table กับ kernel page table ออกจากกัน เพื่อป้องกันการโจมตี (จาก meltdown attack)

โครงสร้าง Page Table แบบดั้งเดิม

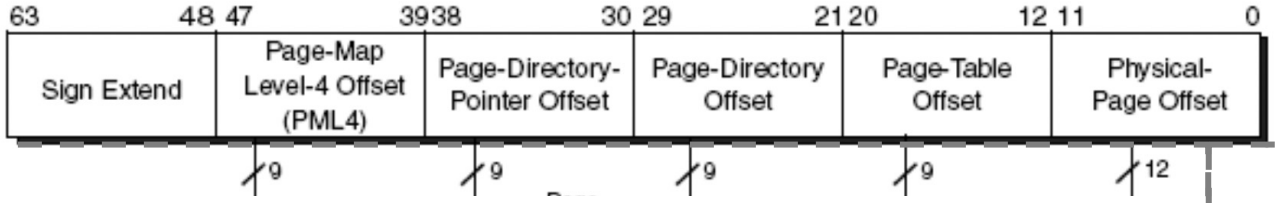
- Page Table ของทุก Process จะมี ส่วนที่รวมเนื้อหาของ kernel อยู่ด้วย ในส่วนที่ cover high memory address
- เนื้อหาของส่วน kernel ในทุก process จะเหมือนกัน
- เมื่อ process ประมวล OS จะกำหนดค่า cr3
  - เมื่อ process ใช้ system call ไม่ต้องเปลี่ยน cr3
  - เมื่อเกิด interrupt ในระหว่างการประมวลผล process ไม่ต้องปลี่ยน cr3
  - kernel เปลี่ยน cr3 เมื่อ scheduler เลือก process ใหม่เข้าใช้ cpu
- ในระหว่างที่ kernel ประมวลผล มันจะสามารถเข้าถึง memory ใน user space ได้

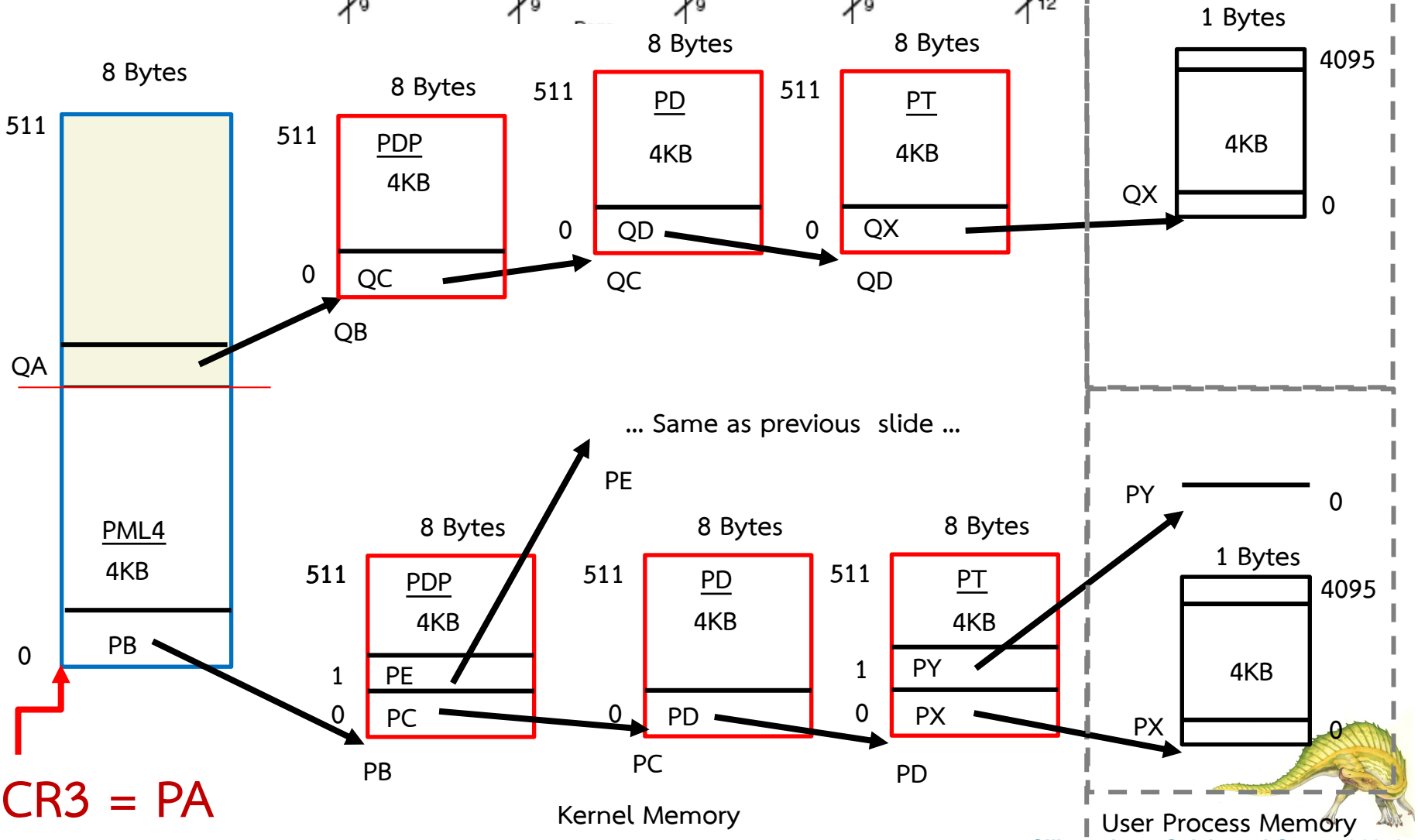
Page Table Structure in User space





Page Table Structure in User space & Kernel space

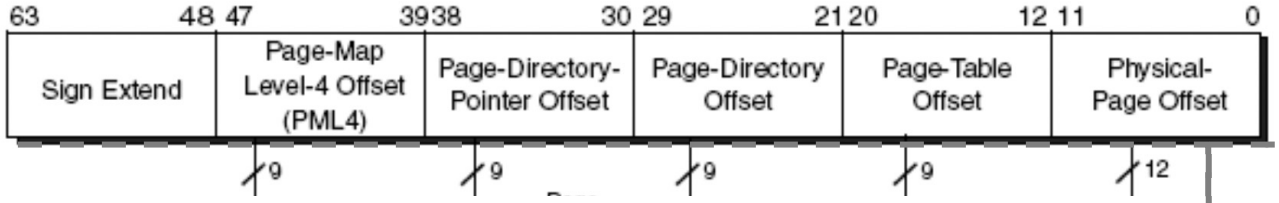




โครงสร้าง Page Table แบบ KPTI

- Kernel Page Table Isolation (KPTI) เป็นการแยก Page Table ของ kernel ออกจาก Page Table ของ Process
- เมื่อ process ใช้ system call OS จะต้องเปลี่ยน cr3 ให้ชี้ไปที่ Kernel Page Table และเปลี่ยนกลับเมื่อ resume การทำงานของ Process
- เมื่อเกิด interrupt ในระหว่างการประมวลผล process OS ต้องปลี่ยน cr3 ให้ชี้ไปที่ Kernel Page Table และเปลี่ยนกลับเมื่อ resume การทำงานของ Process
- ในกรณี system call ก่อน OS เปลี่ยน cr3 มันจะต้อง copy ค่าการ mapping ไปยังหน่วยความจำของ user process ที่ kernel จำเป็นต้องเข้าถึงมายัง kernel page table เพื่อให้ kernel สามารถใช้ user memory ได้(เท่าที่จำเป็นเท่านั้น)

Page Table Structure in User space



8 Bytes

511

เก็บค่า address ของ KPT (KP) ไว้ใช้ในพื้นที่ส่วนนี้

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

PT
4KB

0

PZ

PF

8 Bytes

511

PD
4KB

0

PD

PC

8 Bytes

511

PT
4KB

1

PY

0

PX

PD

Kernel Memory

1 Bytes

4095

4KB

PZ

0

1 Bytes

4095

4KB

PY

0

1 Bytes

4095

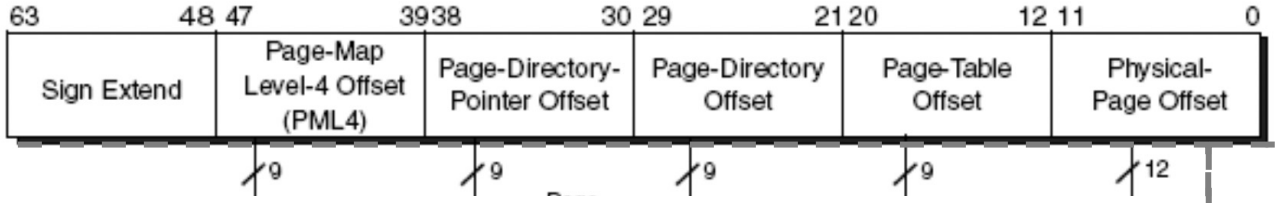
4KB

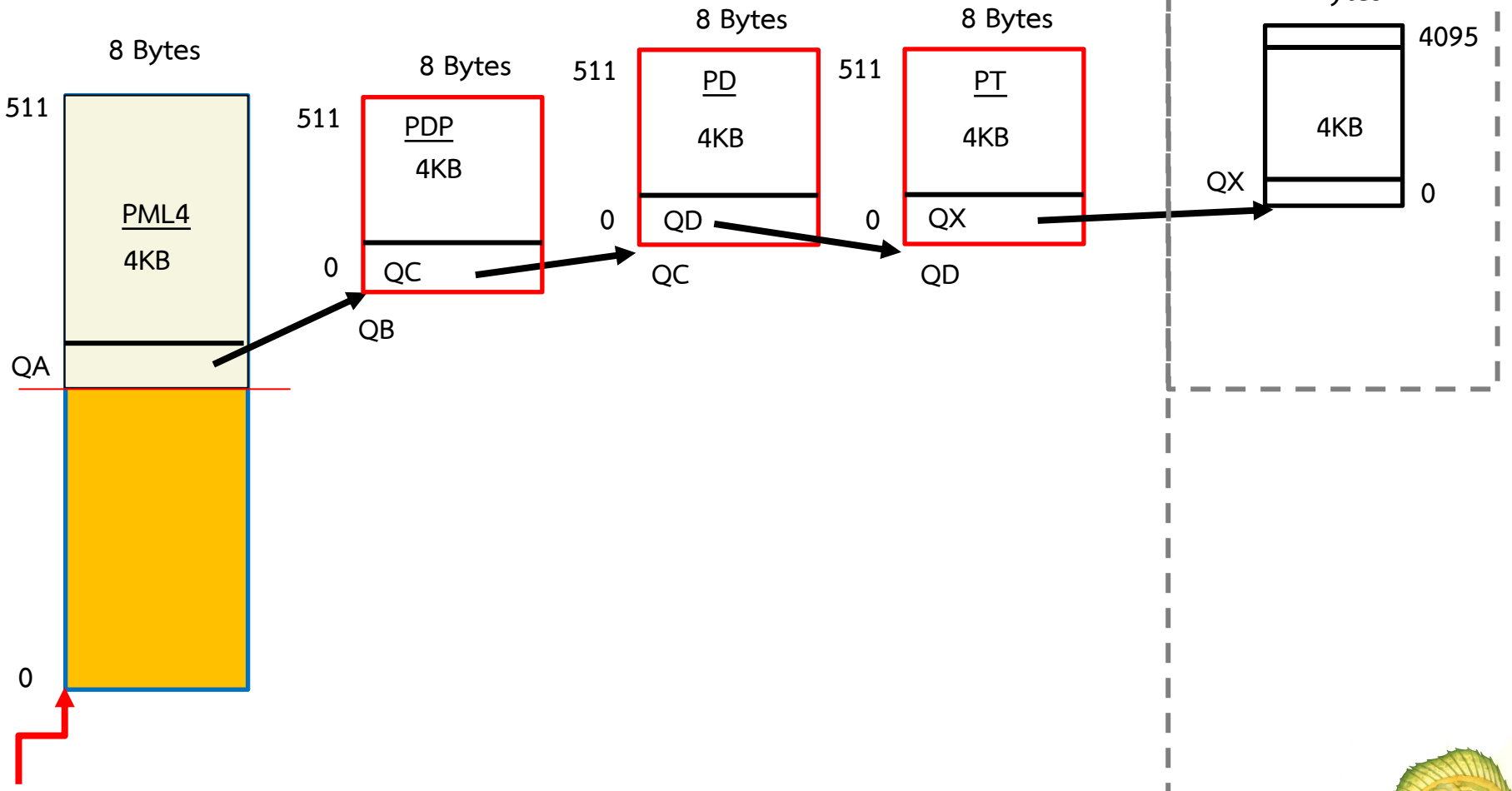
PX

0

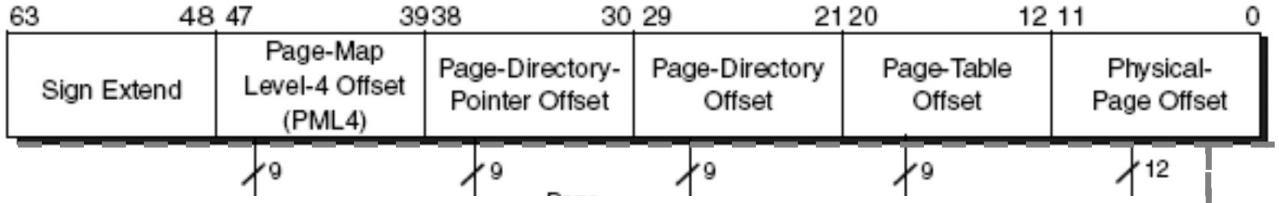
User Process Memory

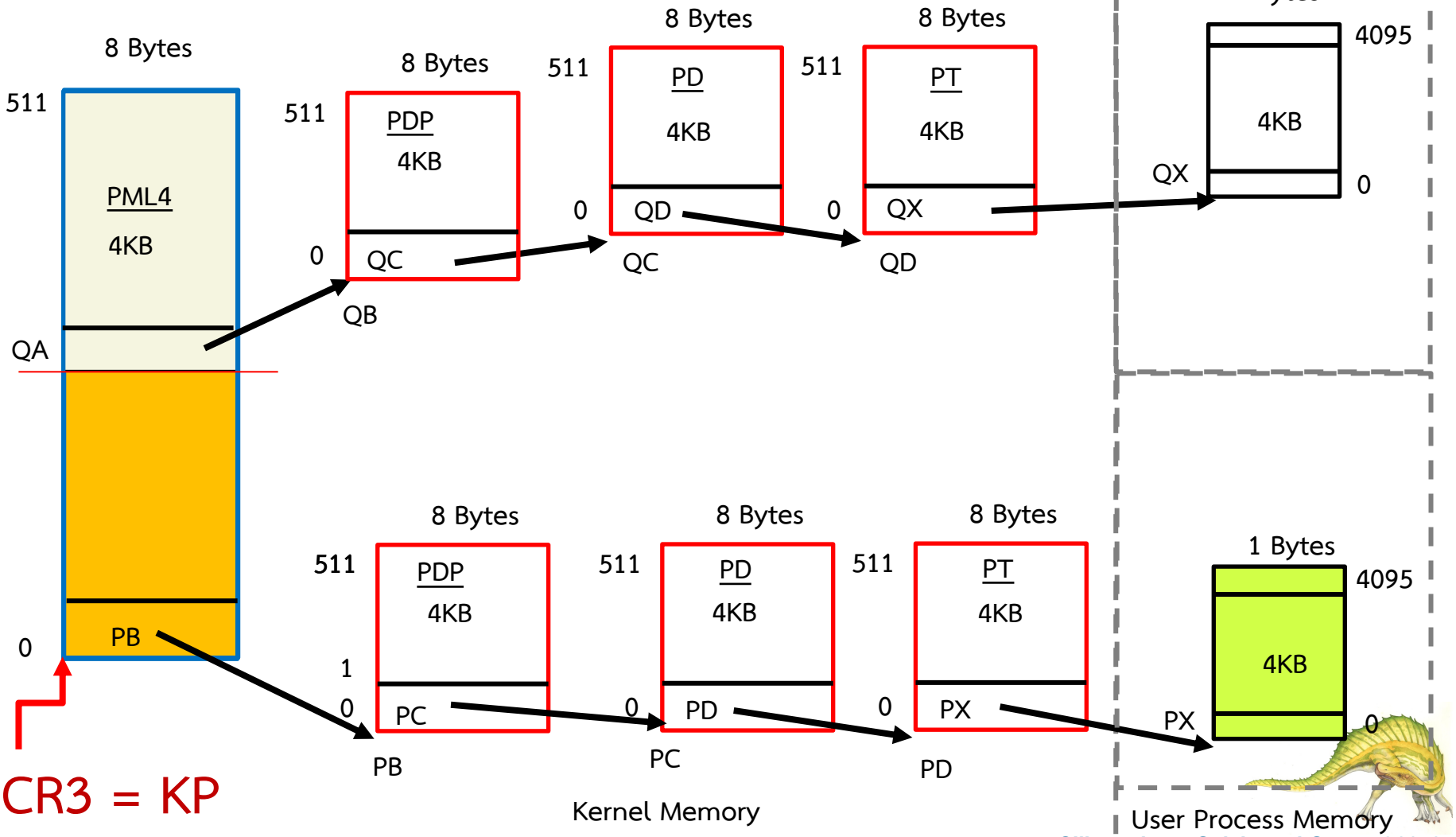
เปลี่ยน CR3 ชี้ Page Table Structure in Kernel space





Copy เฉพาะ Mapping ที่ต้อใช้มายัง Kernel space





End of Chapter 10