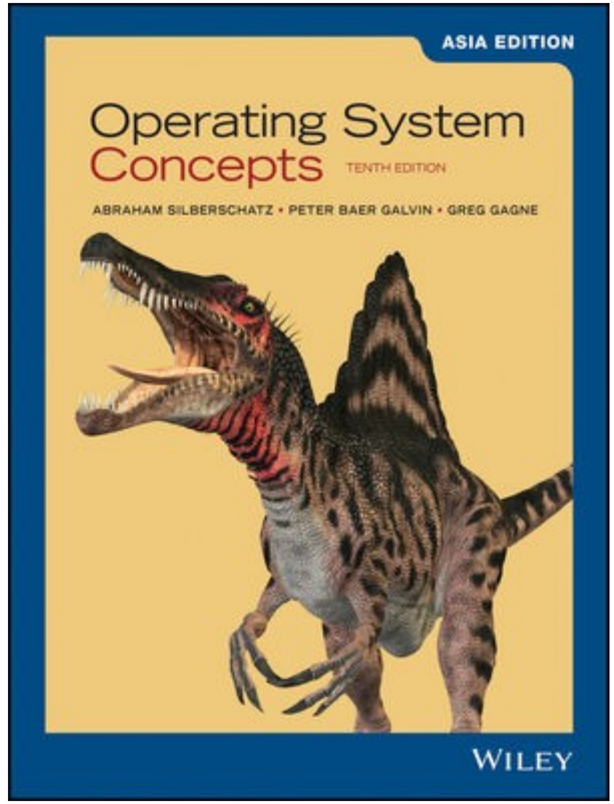
Chapter 10: Virtual-Memory Management

Dept. of Software, Gachon Univ.
Joon Yoo



Objectives

- Illustrate how pages are loaded into memory using demand paging.
- Explain concept of page faults
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe concept of thrashing and working set algorithms

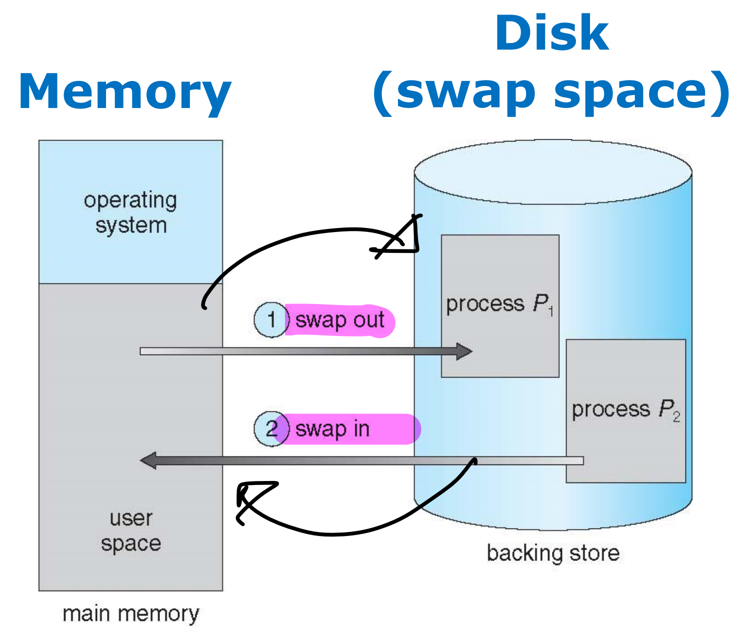
Chapter 10: Virtual-Memory Management

- **Background**
- Demand Paging
- Page Replacement
- Page Replacement Algorithms
- Thrashing
- Swapping on Mobile Systems

Chapter 9: Swapping

- Q: What if memory needed by process exceeds physical memory size?
  - In other words, can process size > physical memory?

- **Swapping**
  - A whole process can temporarily **swap out** of memory to a backing store
  - then **swap in** into memory for continued execution
    - swap device – usually HDD, SSD, or Flash



Chapter 9: Swapping

- Context switching time (=swap time) of swapping system is very high
  - e.g., user process 100MB, hard disk 50MB per second: then, 100MB / 50MB per second = 2 seconds. Swap-in + swap-out = 2 seconds x 2 = 4 seconds

- Swapping takes too much time! Modern OSes try to avoid swapping – only enable if available physical memory is small

  물리적인 메모리가 작을 경우만 허용

- Can we do better?
  - Swap only portions of process (rather than entire process) to decrease swap time: **Demand Paging**

  프로세스 전부가 아니라 "일부" 만 swapping 하는 것.

Chapter 10: Virtual-Memory Management

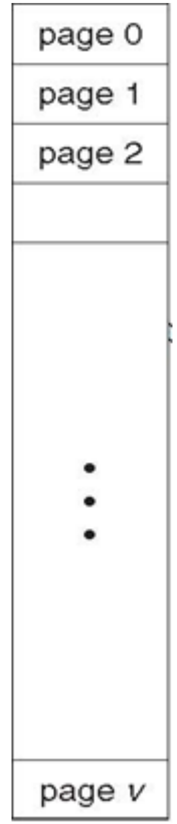
- Background
- **Demand Paging**
  - Basic Concepts
  - Page Fault
  - Performance of Demand Paging
  - Examples
- Page Replacement
- Page Replacement Algorithms
- Thrashing
- Swapping on Mobile Systems

Demand Paging Concept

- We can view a process as
  - a sequence of pages (rather than large contiguous address spaces)

- How can an executable program be loaded from disk into memory?
  - **Option 1**
    - Load the entire program into memory
  - **Option 2**
    - **Load pages only as they needed – Demand Paging**

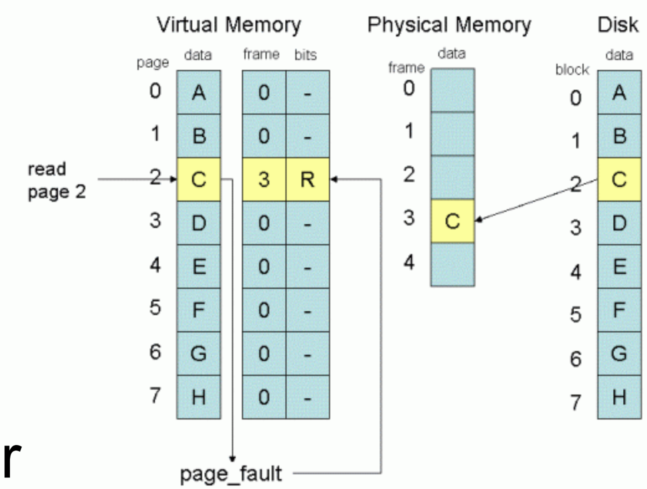
사용하는 page만 메모리에 올려놓자!



Demand Paging

- Motivation: Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures (arrays)
  - e.g., Total 500MB program. Partially load only 100MB and start program. The rest can be loaded on-demand!

- Why? (Advantages)
  - Less I/O needed from disk to load memory
    - Faster response to load programs
  - Less memory needed
    - Larger programs can be partially loaded
  - Better multi-program – more users for server



Valid-Invalid Bit

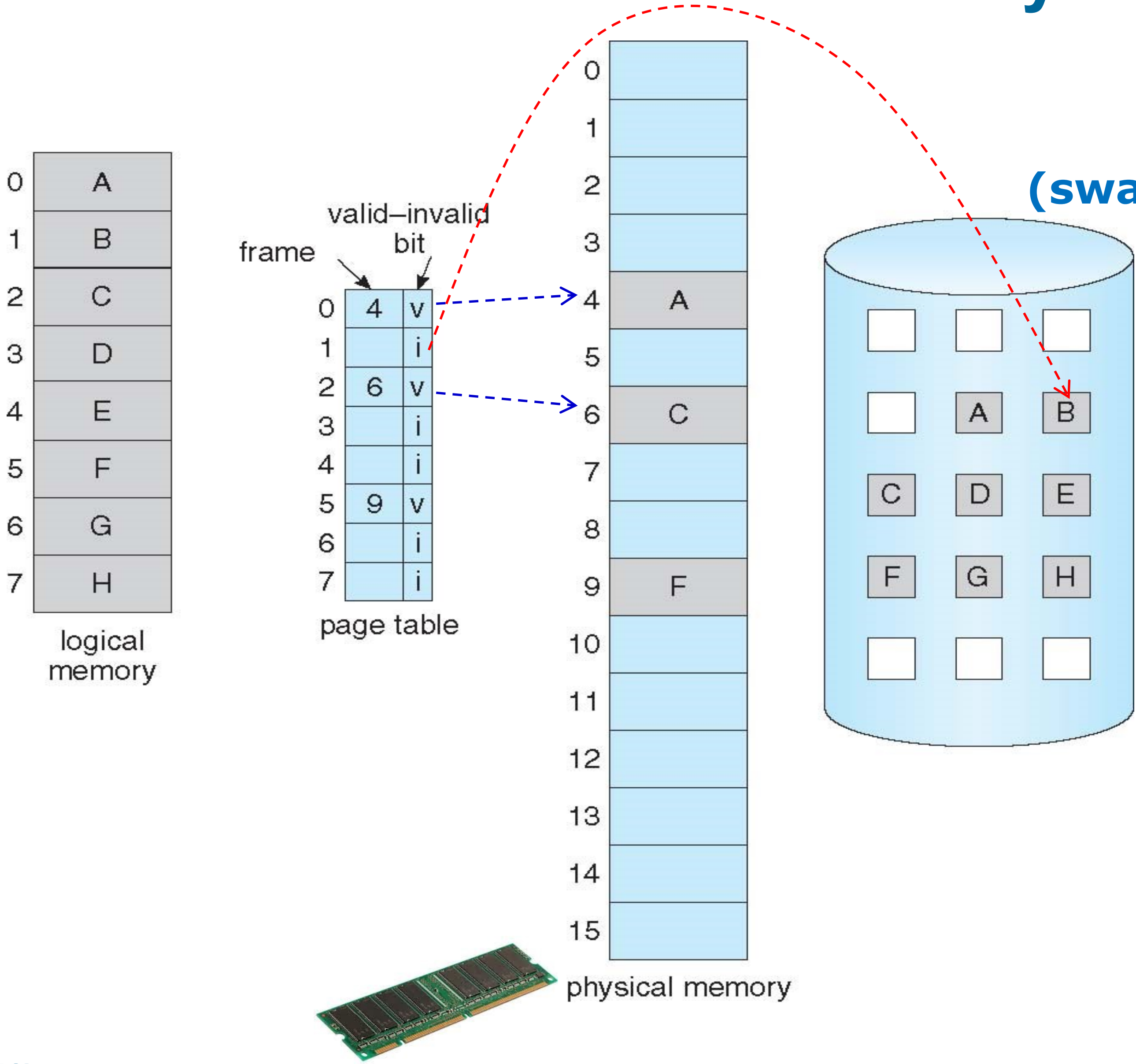
- With each page table entry a **valid–invalid bit** is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
  - or it can be represented as **1** / **0** for valid/invalid.
- Initially valid–invalid bit is set to **i** (or **0**) on all entries
- Example of a page table snapshot:
- During address translation, if valid–invalid bit in page table entry is **i** (or **0**) ⇒ **page fault**

Frame #	valid-invalid bit
	v
	i
	v
	v
	i
….	
	v
	i

page table

처음에는 전부 다 invalid. 메모리에 올라오면 그때 valid로 변경됨.

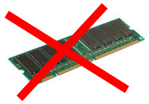
Page Table When Some Pages Are Not in Main Memory



page fault

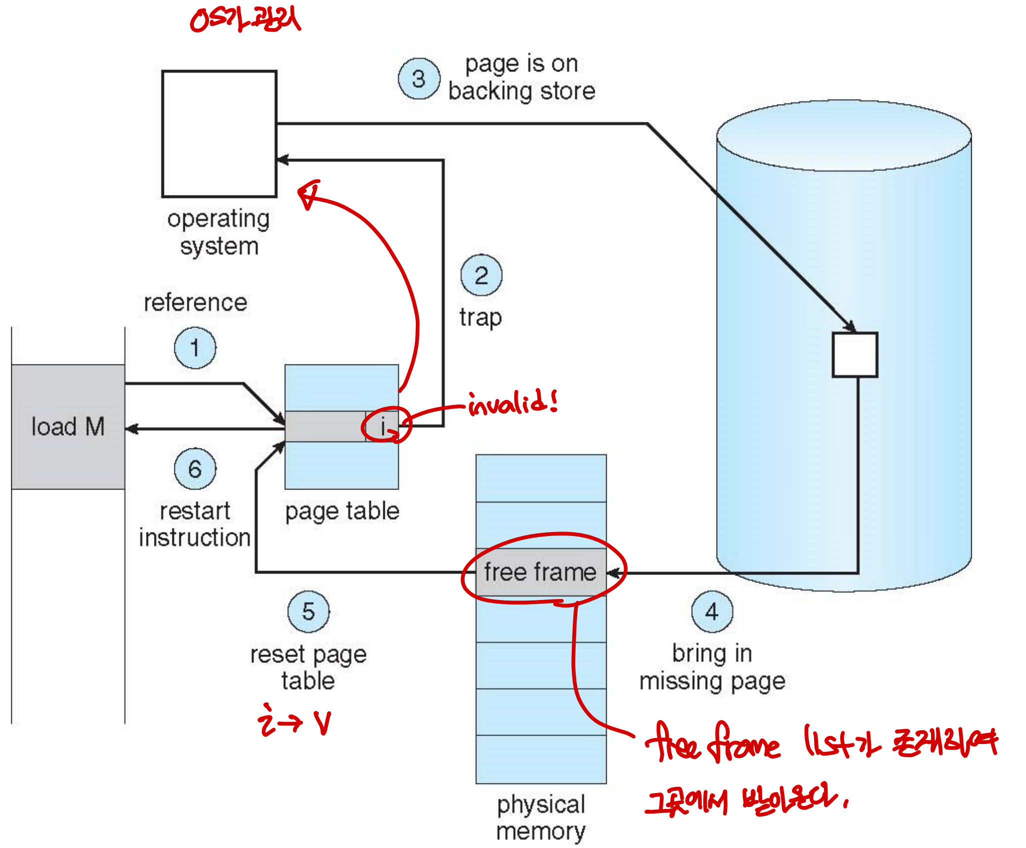
원하는 page가 memory에 없을 경우, disk에 그 page가 있다면 그것을 page-fault 라고 한다.

Page Fault

- **Page fault**  
  - If there is a reference to an <u>invalid</u> page
  - Reference to that page will <u>**trap (software interrupt)**</u> to operating system

page table

- OS handling **page fault**
  1. Operating system looks at page table to decide
  2. **PT: validation bit = i ⇒ <u>not</u> in memory (trap)**
  3. Page is in swap-space - get an empty frame from **physical memory**
  4. Bring in page from **disk** into **physical memory frame**
  5. Reset **page table** to indicate **page** now in memory
     Set **validation bit = v**
  6. **Restart** the instruction that caused the page fault



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- **Effective Access Time (EAT)** for *demand paging*

EAT = $(1 - p)$ x **memory access** + p x **page fault time**

page fault time = page fault overhead

+ swap page out [if needed]

+ swap page in

+ restart overhead

page replacement

$\boxed{\text{memory access time <<< page fault time}}$

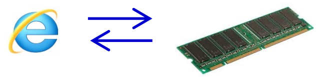
크게 고려하지 않음

오랜 시간이 걸림.

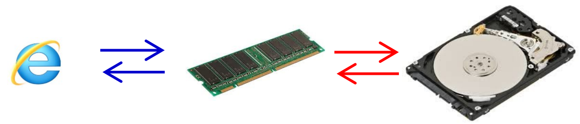
→ disk I/O 발생

Demand Paging Example

- Memory access time = **200 ns**



- Average page-fault service time = **8 ms** (=8,000,000 ns)



- EAT = (1 – p) x **200 ns** + p (**8 ms**)
  = (1 – p) x 200 + p x 8,000,000 (ns)
  = 200 + p x 7,999,800 (ns)
- If one access out of 1,000 causes a page fault (p=0.001), then
  EAT = 8,200ns (= **200ns** x 41)
  This is a slowdown by a factor of 41x!! (or 97.6% degradation)

page fault가 한번만 일어나도
엄청난 시간낭비가 발생한다.

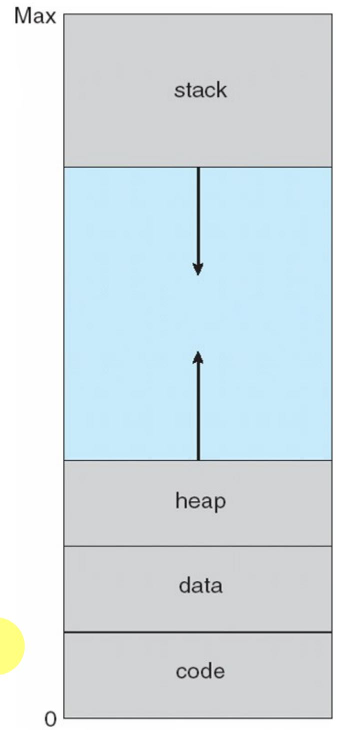
- If we want performance degradation < 10 % (or EAT < 220ns)
  - 220 > EAT = 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

10% 성능 향상을 위해 p의 최댓값이 .... = 0.0000025 ... ㅋㅋ!

- **<u>It is important to keep the page-fault rate low in a demand-paging system!</u>**

Demand paging usage: Heap/Stack

- Heap: grow upward in memory as it is used for dynamic memory allocation
- Stack: grow downward in memory through successive function calls
- Do not need to allocated physical frames for heap/stack at the beginning - will require actual physical frames only if the heap and stack grows
- Ideally implemented via **demand paging**
- Easy to extend memory space of process!

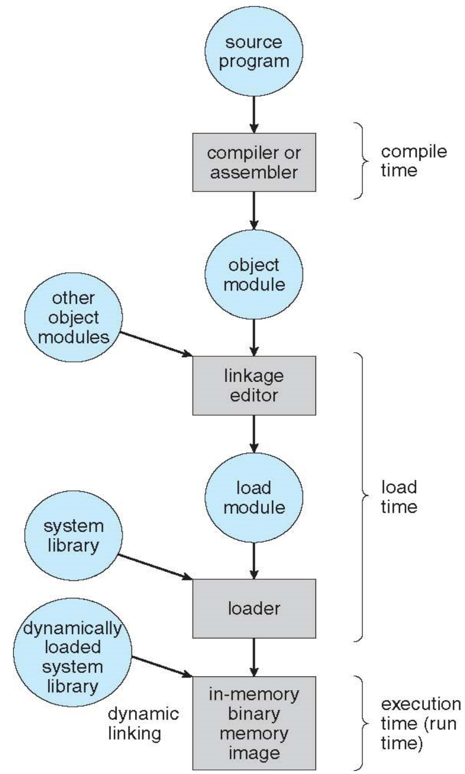


Demand paging usage: Dynamically linked libraries (DLL)

- **Dynamically linked libraries (DLL)** – Chapter 9.1.5

DLL에 해당하는 library는 해당하는 코드가 실행될 때 그제서야 실행됨!

  - Library linking is postponed until execution time
  - A **stub** is included in binary image for each DLL reference
    - **stub**: piece of small code that indicates how to locate appropriate routine.

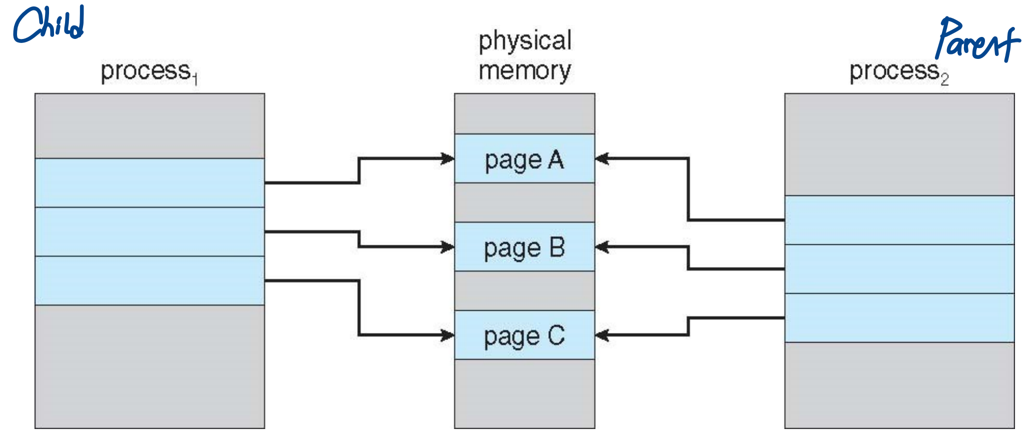


fork(): Copy-on-Write (Ch. 10.3)

- **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
  - when there are copy-on-write pages
  - or when child executes exec()

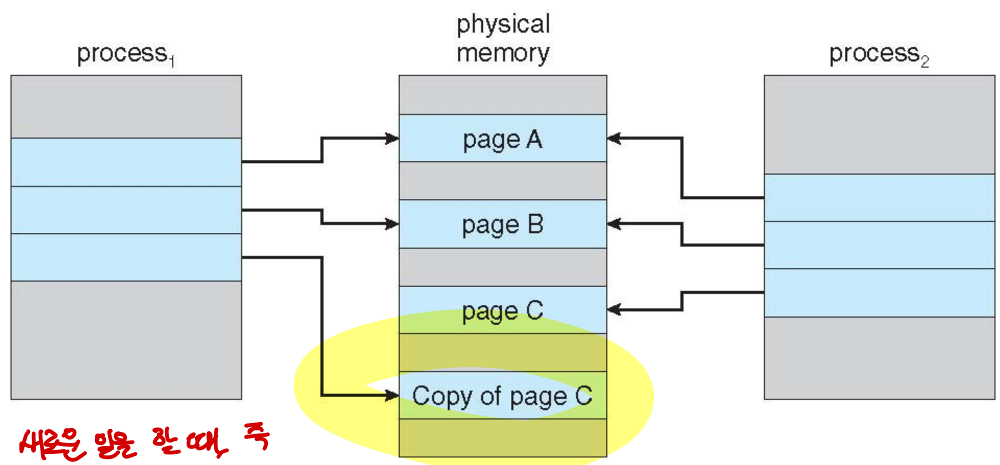
→ demand paging!

Before Process 1 Modifies Page C



어차피 fork();하면, child = parent 이므로 그냥 child를 위한 physical memory를 따로 주지 않고 parent가 쓰던 메모리 공간을 share 한다.

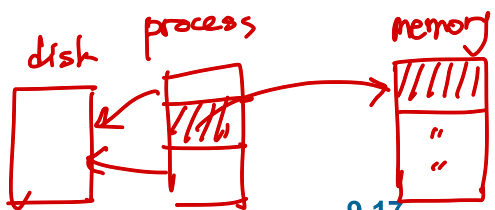
After Process 1 Modifies Page C



새로운 일을 할 때, 즉 'write'를 할 때 그제서야 copy해서 공간을 따로 주게 된다.

Benefits of Virtual Memory (using Demand Paging)

- **Protection, Transparency, Sharing**
  - Protection: A process can only access the virtual address (pages); A bug in one process can corrupt memory in another
  - Transparency: A process sees a contiguous virtual memory space (pages) cannot access physical memory space directly
- **Resource exhaustion**
  - Sum of sizes of all processes often greater than physical memory
  - **Demand paging**
    - e.g., If a process is 20 pages, we can execute with 10 frames and the rest in the swap space (disk) – use demand paging and page-replacement



Chapter 10: Virtual-Memory Management

- Background
- Demand Paging
- **Page Replacement**
  - Basic Page Replacement
  - Modify (Dirty) Bit
- Page Replacement Algorithms
- Thrashing
- Swapping on Mobile Systems

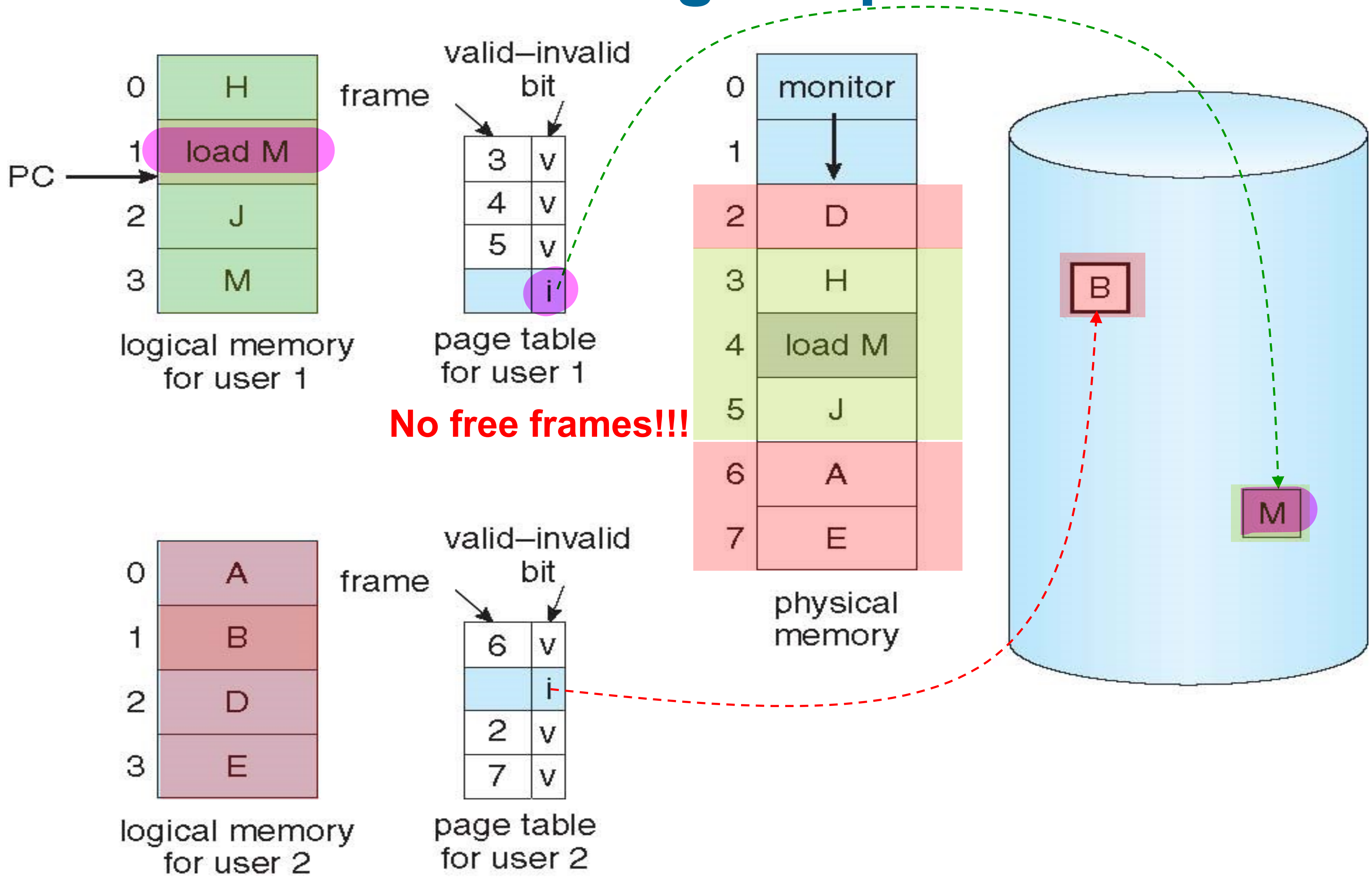
What Happens if There are no Free Frames?

- All memory may be in use
  - Multiprogramming – multiple processes share the memory
  - Page fault → OS finds the desired page on the disk → find the free-frame
  - If NO free frames on the free-frame list ? – i.e., physical memory is full!

Swaping을 사용해도 되지만...

- **Page replacement**
  - Find some page in memory, but not really in use, **page out**
  - Performance – want an algorithm which will result in minimum number of page faults
    - **Page replacement Algorithms**

Need For Page Replacement

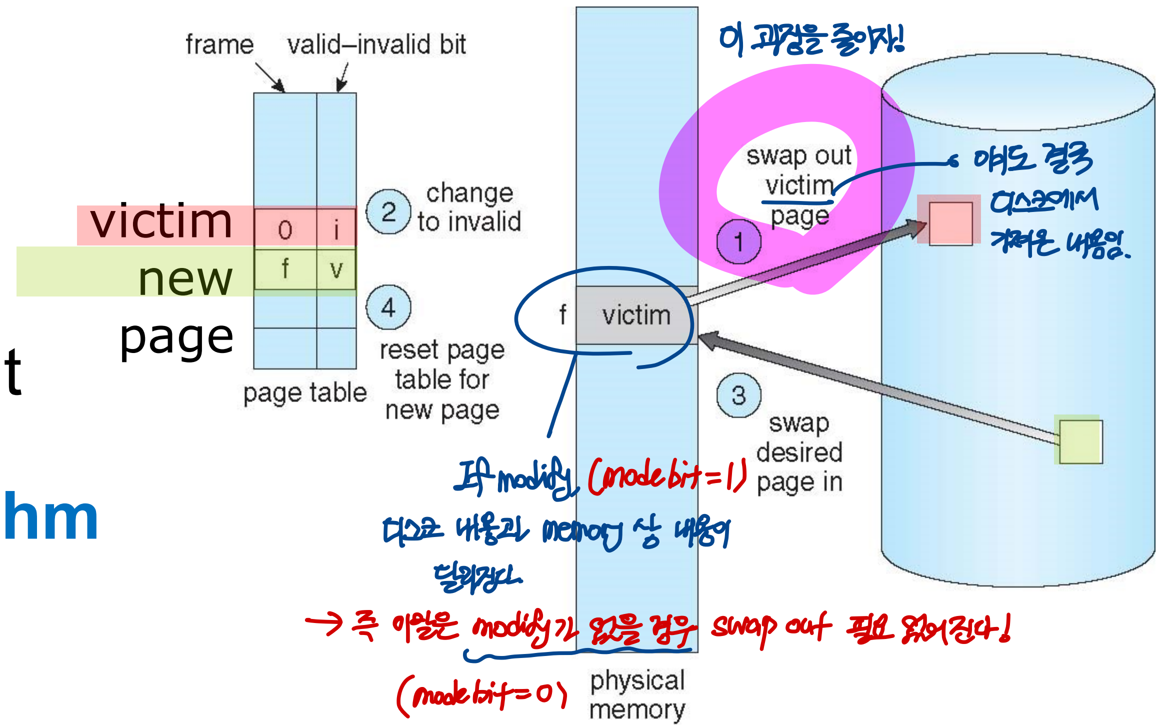


Basic Page Replacement

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**



이 과정을 줄이자!

어차피 결국 디스크에서 가져온 내용임.

If modify (mode bit=1)

디스크 내용과 memory 상 내용이 달라진다.

→ 즉 이말은 modify가 없을 경우 swap out 필요 없어진다!

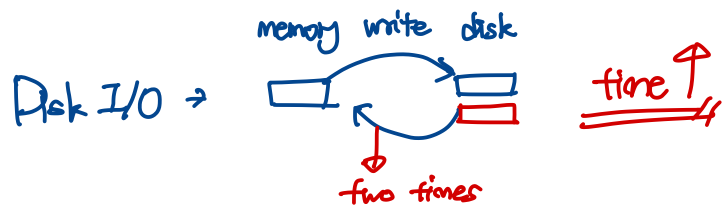
(mode bit=0)

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

새로운 frame 공간을 위해 page 하나를 제거하는데 _ 그것이 victim frame.

그곳에 page를 넣는다.

4. Restart the process



Modify Bit (= Dirty Bit)

- If no frames are free → **TWO** page transfers are required
  - One page out and one page in (see previous slide)
- Can we reduce the number of disk writes (page-outs)?
- Solution: **Modify bit** **(dirty bit)**
  - Initially set to zero
  - Set the bit (to 1) whenever any word or byte in the page is modified
    - only modified pages are written to disk – unmodified pages can just be deleted without writing on disk (it has **not** changed!)

  → can reduce the page-transfer overhead

Chapter 10: Virtual-Memory Management

- Background
- Demand Paging
- Page Replacement
- **Page Replacement Algorithms**
  - FIFO Page Replacement
  - Optimal Page Replacement
  - LRU Page Replacement
- Thrashing
- Swapping on Mobile Systems

Page Replacement Algorithms

- **Page-replacement algorithm**
  - Which frame should be replaced?
  - Want lowest page-fault rate
  - Evaluate algorithm by computing the number of **page faults**

- FIFO page replacement
- Optimal page replacement
- LRU (Least-Recently-Used) algorithm
  - clock, reference-bit, second-chance algorithm

First-In-First-Out (FIFO) Algorithm

- When a page must be replaced, <u>the oldest page</u> is chosen

애초에 page-fault가 발생하지 않으면 page-replacement도 필요 없다

reference string

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

page frames

12 Page Replacements!!
(15 Page Faults)

- How to track ages of pages?
  - Just use a FIFO queue

Replacement Algorithm	Page replacements	Page faults
FIFO	12	15
OPT		
LRU		

Problem of FIFO Algorithm

- Example 2
  - Reference string: 1, 2, 3, 4, 1, 2, 5, <u>**1**</u>, <u>**2**</u>, 3, 4, <u>**5**</u>

  -3 frames (3 pages can be in memory at a time per process)

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

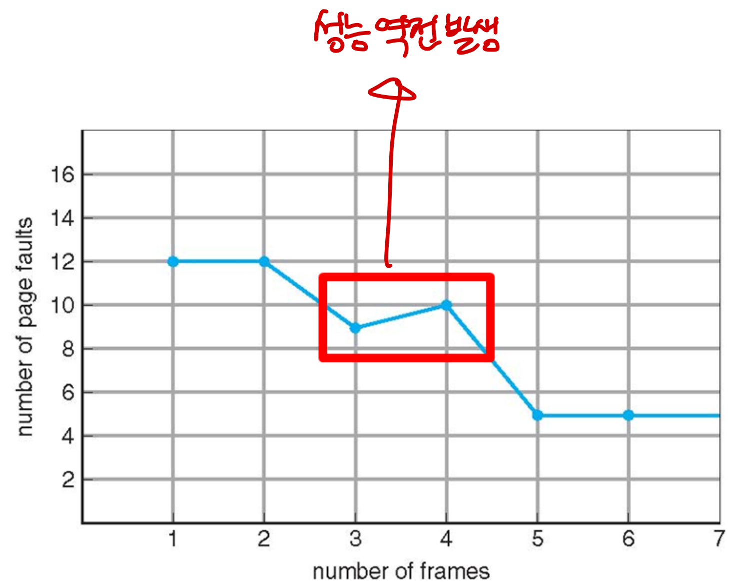
9 page faults

Reference string: 1, 2, 3, 4, <u>**1**</u>, <u>**2**</u>, 5, 1, 2, 3, 4, 5

-4 frames:

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

10 page faults



Why did this happen???

FIFO Illustrating Belady's Anomaly

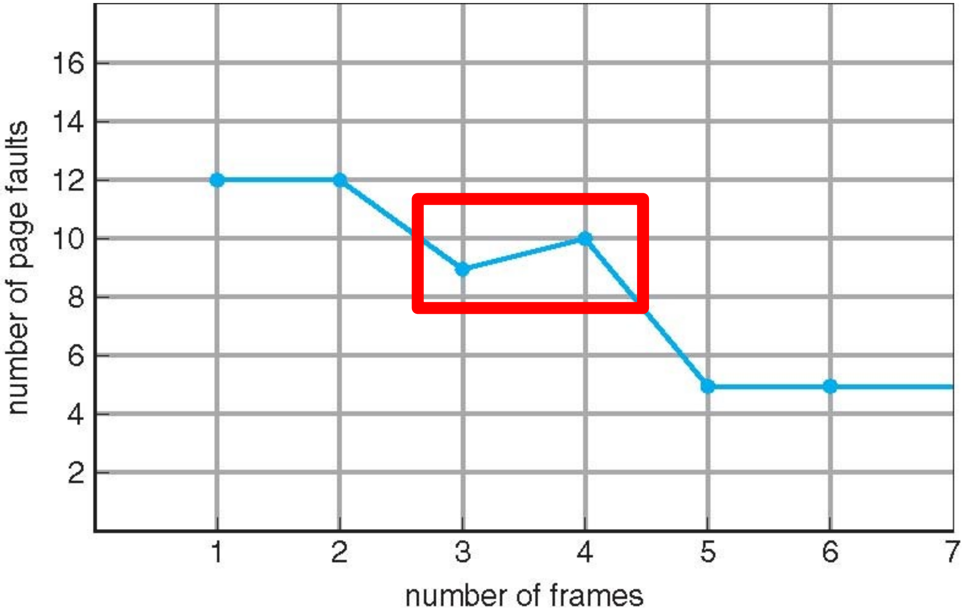
FIFO replaces the **oldest** page – is this always good?

An old page may contain a **frequently used variable**

Belady's anomaly

예외적인 경우가 종종 발생할 수 있다.

Sometimes, page-fault may increase as the number of allocated frames increase!



Optimal Algorithm

오래 안 "쓸" page. 미래에 안쓸 page...

- Algorithm that has the lowest page-fault rate
- Replace page that **will *not*** be used for longest period of time
  - This is a design to guarantee the lowest page-fault rate for a fixed number of frames

6 Page Replacements!!
(9 Page Faults)

Replacement Algorithm	Page replacements	Page faults
FIFO	12	15
OPT	6	9
LRU		

Optimal Algorithm

- Unfortunately, the optimal page-replacement algorithm is difficult to implement
  - Why? ? 어떻게 알아
    - Require **future knowledge** of the reference string – We do not know the future! (Notes: Have we seen this before?) → scheduling SJF

- Mainly used for comparison studies
  - Evaluating a new algorithm
  - e.g. “the new algorithm is within 12.3% of optimal at worst and within 4.7% on average”

Least Recently Used (LRU) Algorithm

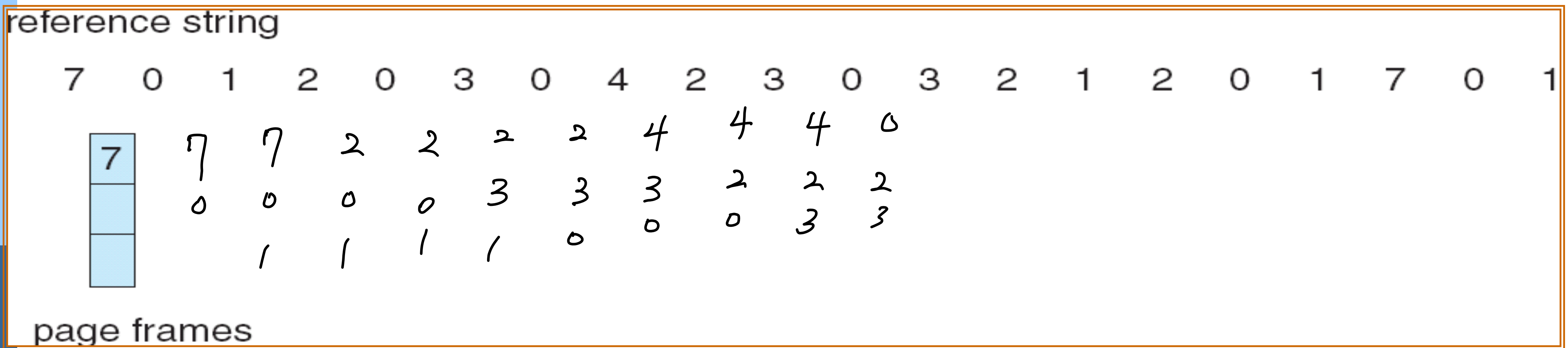
- FIFO: Use time when a page **was brought** into the memory
- OPT: Use time when a page is to be **used**
- Let us predict the future by using the past!

가장 오래동안 쓰지 않은 것

- **Least Recently Used (LRU) Algorithm**
  - Use past knowledge rather than future
  - **Replace** page that has **not been used** in the most amount of time
  - Generally good algorithm and frequently used

Least Recently Used (LRU) Algorithm

- LRU replacement associates with each page the time of that page's last use
- Replace the page that has not been used for the longest period of time

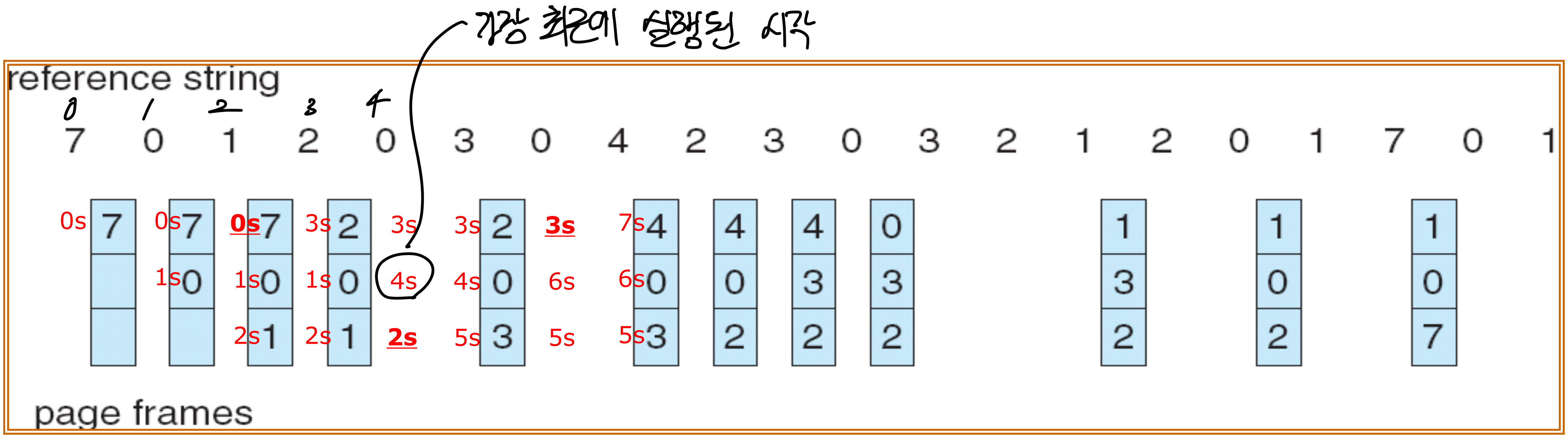


9 Page Replacements!!
(12 Page Faults)

Replacement Algorithm	Page replacements	Page faults
FIFO	12	15
OPT	6	9
LRU	9	12

Implementation of LRU Algorithm

- The major problem is **how to implement LRU replacement**
  - Use ***counter***
  - LRU implementation requires H/W support (due to speed)
- Counter implementation example



Implementation of LRU Algorithm

- *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 – replace page with **smallest** timer value
    - Con1: Need to **search** to replace page (complexity **O(N)** with N frames)
    - Con2: Counter must be updated for each memory reference

LRU Implementation

- Updating of **counter fields** must be done for ***every*** memory reference
- LRU needs special hardware and still slow
  - Counter approach use too much resources
- LRU Approximation: **Reference bit**
  - With each page associate a bit, initially = 0
  - When a page is referenced, then bit set to 1
  - Replace **<u>any</u>** with **<u>reference bit = 0</u>** (if one exists)
  - In this method, although we can not know the **<u>exact order</u>** of page reference, we can know that whether the page is **<u>referenced or not</u>**
  - Basis for many page-replacement algorithms that ***approximate*** LRU replacement

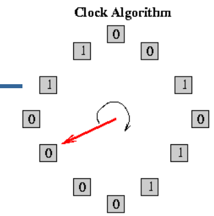
한번이라도 참조되었다면 1로 변경

한번도 참조 안된 것들을 하나 골라 대체한다.

LRU Approximation: Additional-Reference-Bits Algorithm

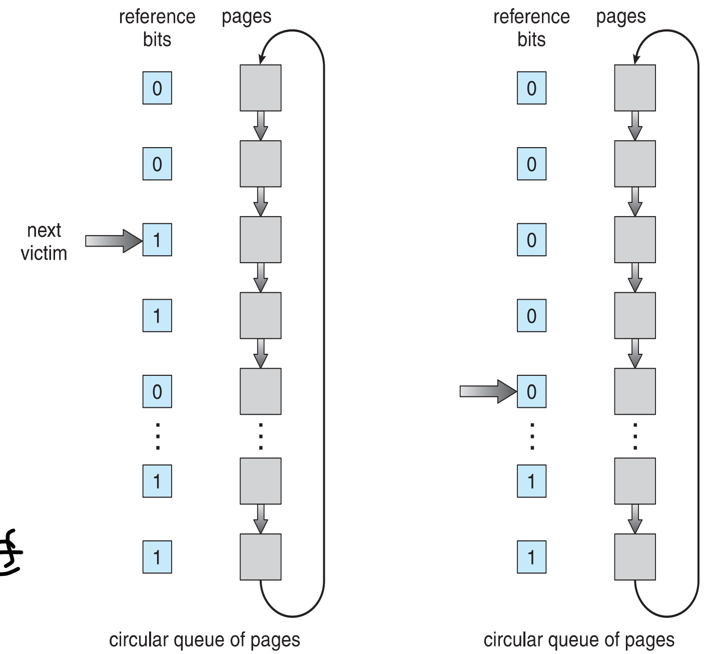
- 8-bit byte for each page in a table in memory
- At regular intervals (e.g., 100ms), the reference bit for each page is shifted right by 1-bit
- Example 100ms 마다, 참조될 경우 1, 아니면 0. 그래서 가장 작은 값을 replace 한다.
  - 00000000: page has not been used for 8 time periods
  - 11111111: page has been used at least once in each period
  - 11000100: page has been used more recently than (01110111)
- Page with lowest integer number is LRU – it can be replaced

Second-chance algorithm (Clock algorithm)



- Use one reference bit per page using **circular list**
- Pointer (hands on clock) indicates which page is to be replaced next
- If page to be replaced has
  - reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - move on to next page
  - reference bit = 0 -> replace it
    - this was the second chance
    - if this page was reference for in the last cycle, then reference bit should have been 1

두번 연속0이면 replace



(a) (b)

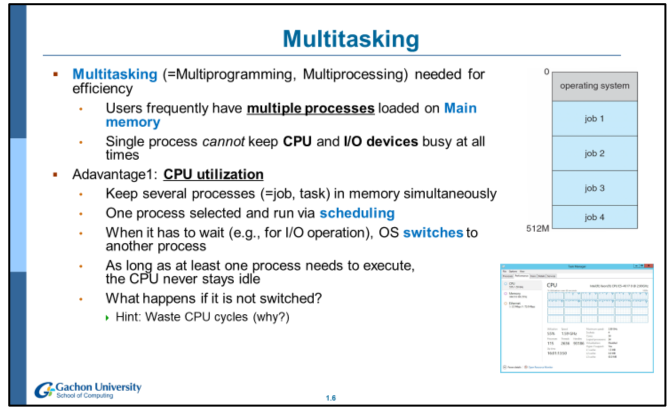
Chapter 10: Virtual-Memory Management

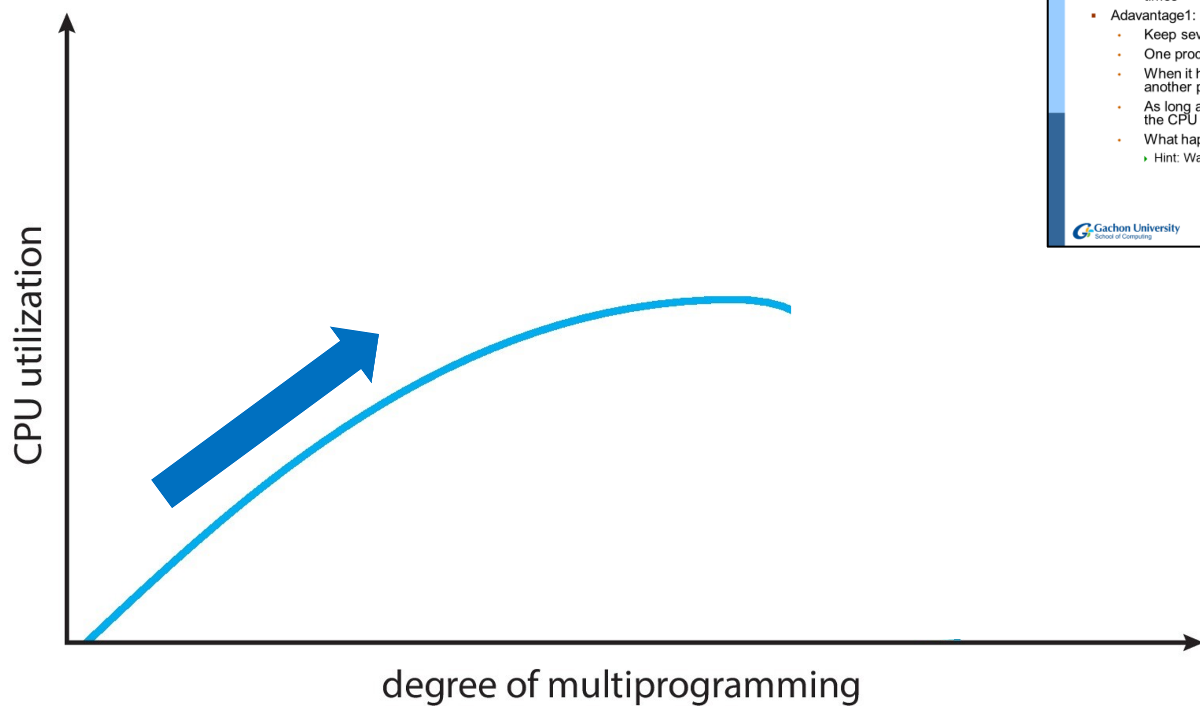
- Background
- Demand Paging
- Page Replacement
- Page Replacement Algorithms
- **Thrashing**
- Swapping on Mobile Systems

Degree of Multiprogramming

- **Degree of multiprogramming**: Number of programs in memory.
- If CPU utilization is low, operating system thinks that it needs to **increase** the degree of multiprogramming
  - Another process added to the system. But, ...

Ch. 1

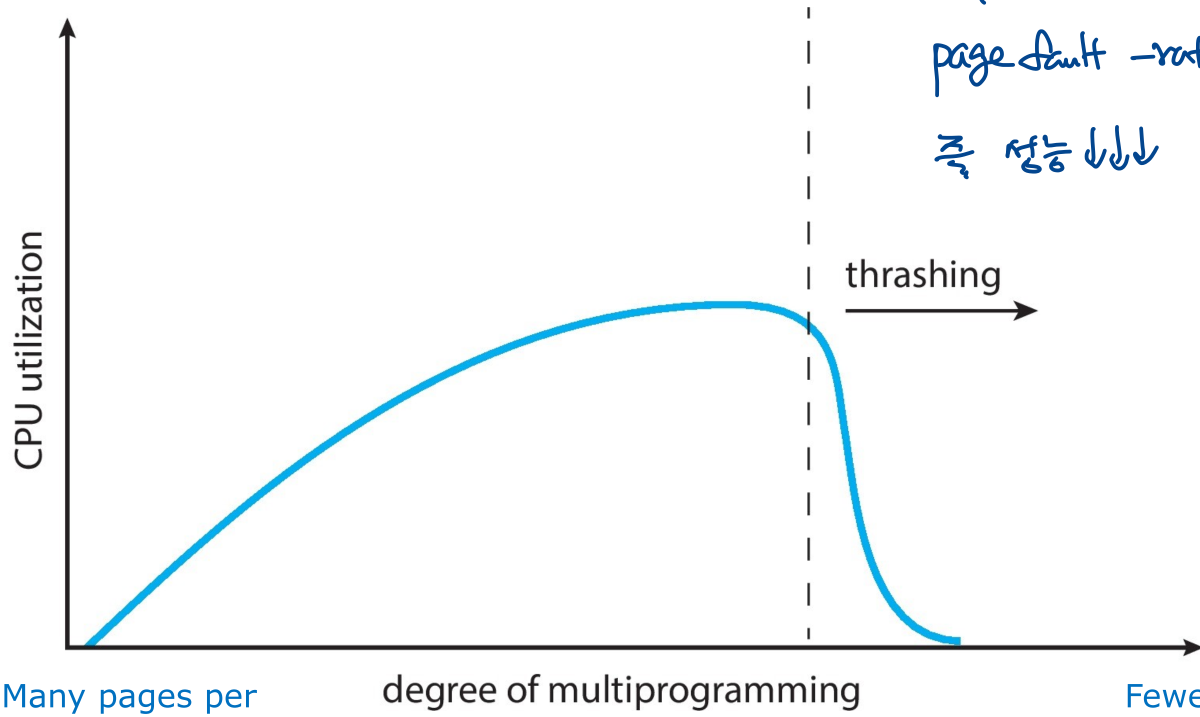




Thrashing

- **Thrashing.** A process is busy dealing with page faults
  - 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

너무 많이 실행시키면, frame이 부족해서
page fault -rate가 많이 올라간다.
즉 성능 ↓↓↓



Many pages per process

Fewer pages per process – more page faults – lower CPU utilization

Demand Paging and Thrashing

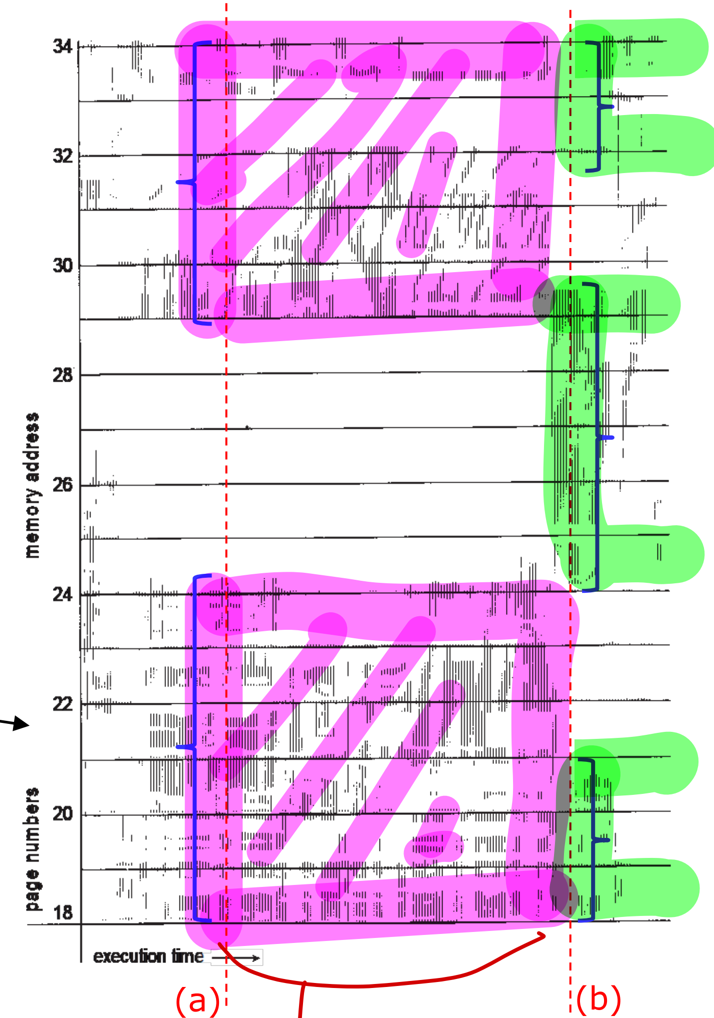
- To prevent thrashing
  - provide a process with as many frames as it **needs**
  - How many does it **need**?
- Locality model

메모리의 특정 지역을 특정 시간대에 점령하고 있다!

  - Process migrates from one locality to another (e.g., from (a) to (b))
  - Localities may overlap
- Why does thrashing occur?

Σ size of locality > free frames

필요한 메모리 – locality 보다 free frame가 적으면 thrashing 발생



Locality In A Memory-Reference Pattern

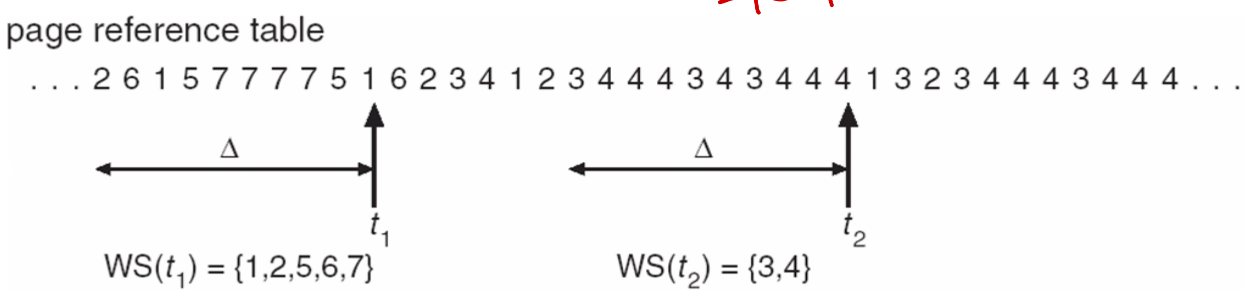
비슷한 곳을 계속 사용!

Working-Set Model

앞으로 계속 사용할 것 같은 곳을 메모해 놓는 것

- **working-set window** (Δ): the most recent Δ pages
  - an approximation of the program's locality
  - Example: Δ=10

최근에 사용한 페이지는 무엇인가?



  - if Δ too small will not encompass entire locality

너무 작으면 locality를 다 저장하지 못한다

  - if Δ too large will encompass several localities

예전 데이터 구간도 생각해야 하므로 낭비

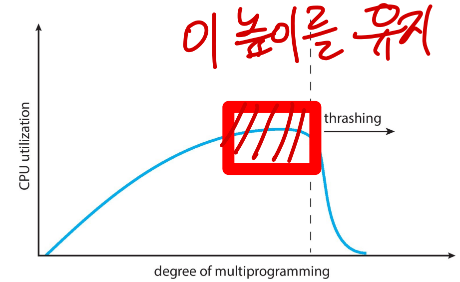
  - if Δ = ∞ ⇒ will encompass entire program

의미없음.

Working-Set Model (Cont.)

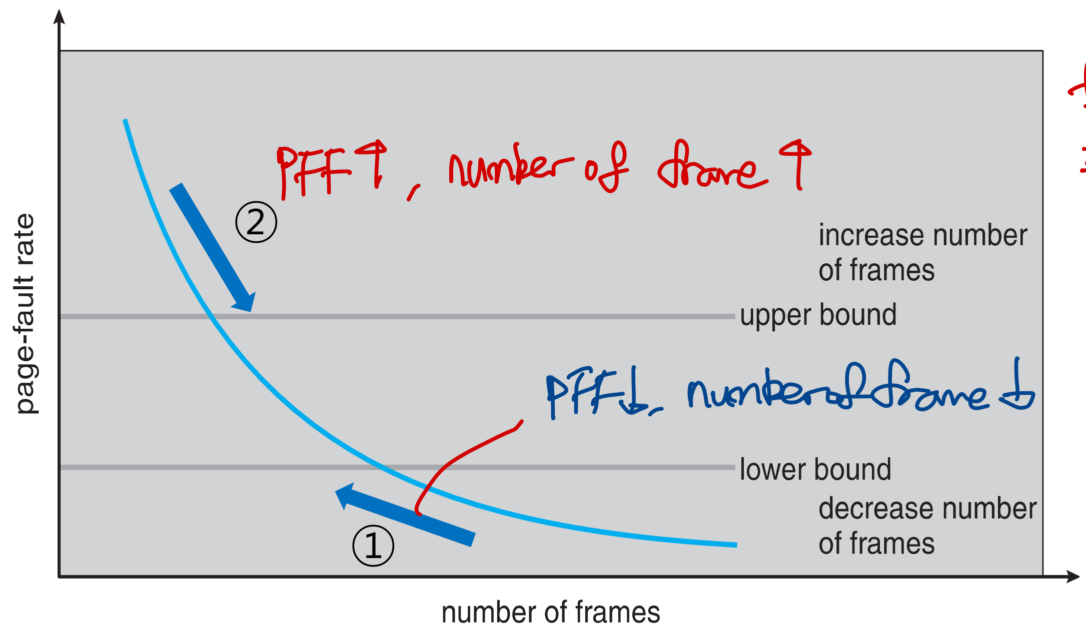
- $D = \Sigma\, WSS_i \equiv$ total demand frames 요구하는 frame 총 개수
  - WSS_i: working set size for each process
- Policy
  - if D < m (m: available frames), then another process can be initiated
  - if D > m, then Thrashing! ⇒ suspend or swap out a process ⇒ degree of multiprogramming (thus thrashing) reduces
  - Suspended process can restart later
- Prevents thrashing while keeping degree of multiprogramming as high as possible – optimizes CPU utilization

이 높이를 유지



Page-Fault Frequency (PFF)

- More direct approach than working set model
- Establish “acceptable” **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame (①)
  - If actual rate too high, process gains frame (②)
- If page fault rate increases and not enough free frames, then process swapped out ⇒ degree of multiprogramming (thus thrashing) reduces



frame도 마냥 많다고 좋지 않다.
즉 적절한 bound를 유지해야 한다!

Chapter 10: Virtual-Memory Management

- Background
- Demand Paging
- Page Replacement
- Page Replacement Algorithms
- Thrashing
- **Swapping on Mobile Systems**

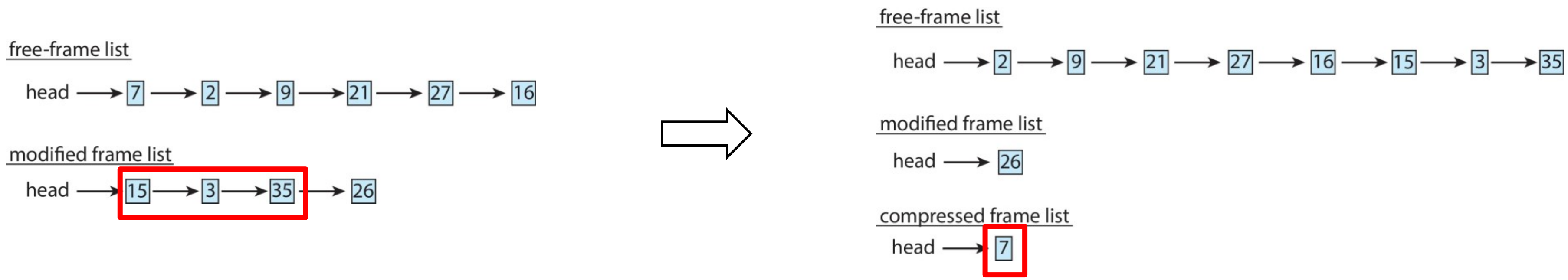
Swapping on Mobile Systems (Ch. 9.5.3)

- Typically, mobile systems do not support swapping pages, since they use **Flash memory** instead of disks
  - flash memory space constraint
  - flash memory becomes unreliable and shows poor throughput after some number of writes
- Apple iOS: ask application to voluntarily relinquish allocated memory

  - Read-only data (e.g., code) are removed from main memory, later reload if necessary
  - Some applications may be terminated by OS
- Android: Similar to iOS, but Android writes application state to flash memory before App termination for quick restart.

Memory Compression (Ch. 10.7)

- **Memory compression** in mobile systems
  - Rather than paging out modified frames to swap space, compress several frames into a single frame



- **Compression ratio**: amount of reduction achieved by compression algorithm
  - High compression ratio: slower, more computation

Summary

- Virtual memory is how we stuff large programs into small physical memories.
- We perform this by using demand paging, to bring in pages only when they are needed.
- But to bring pages into memory, means kicking other pages out, so we need page replacement algorithms.

https://www.istockphoto.com/kr/%EC%9D%BC%EB%9F%AC%EC%8A%A4%ED%8A%B8/qa