Paging Implement

l: Logical address (), P: Physical address 일 때,

• MSB (Page number): (l>>12) & 0xFFFFF

• LSB (Page offset): l & 0xFFF

• Physical Address: (pMSB << 12) | LSB

Virtual Memory

1. BackGround

명령을 실행하려면 물리적 메모리에 있어야 한다는 요구 사항은 필요하고 합리적인 것인 것 같지만, 실제 프로그램을 살펴보면 대부분의 경우 전체 프로그램이 필요하지 않다. 예를 들어,

• 비정상적인 오류 상황을 처리하는 코드

• 배열, 목록 및 표에는 실제로 필요한 것보다 많은 메모리가 할당되는 경우

• 프로그램의 특정 옵션 및 기능이 거의 사용되지 않는 경우

또한, 전체 프로그램이 필요한 경우에도 모든 프로그램이 동시에 필요한 것은 아니다. 부분적으로 만 메모리에 있는 프로그램을 실행하는 것은 많은 장점이 있다.

• 프로그램은 사용가능한 실제 메모리의 양에 의해 제약 받지 않는다. 사용자는 매우 큰 가상 주소 공간에 대한 프로그램을 작성할 수 있으므로 프로그래밍 작업이 단순해짐

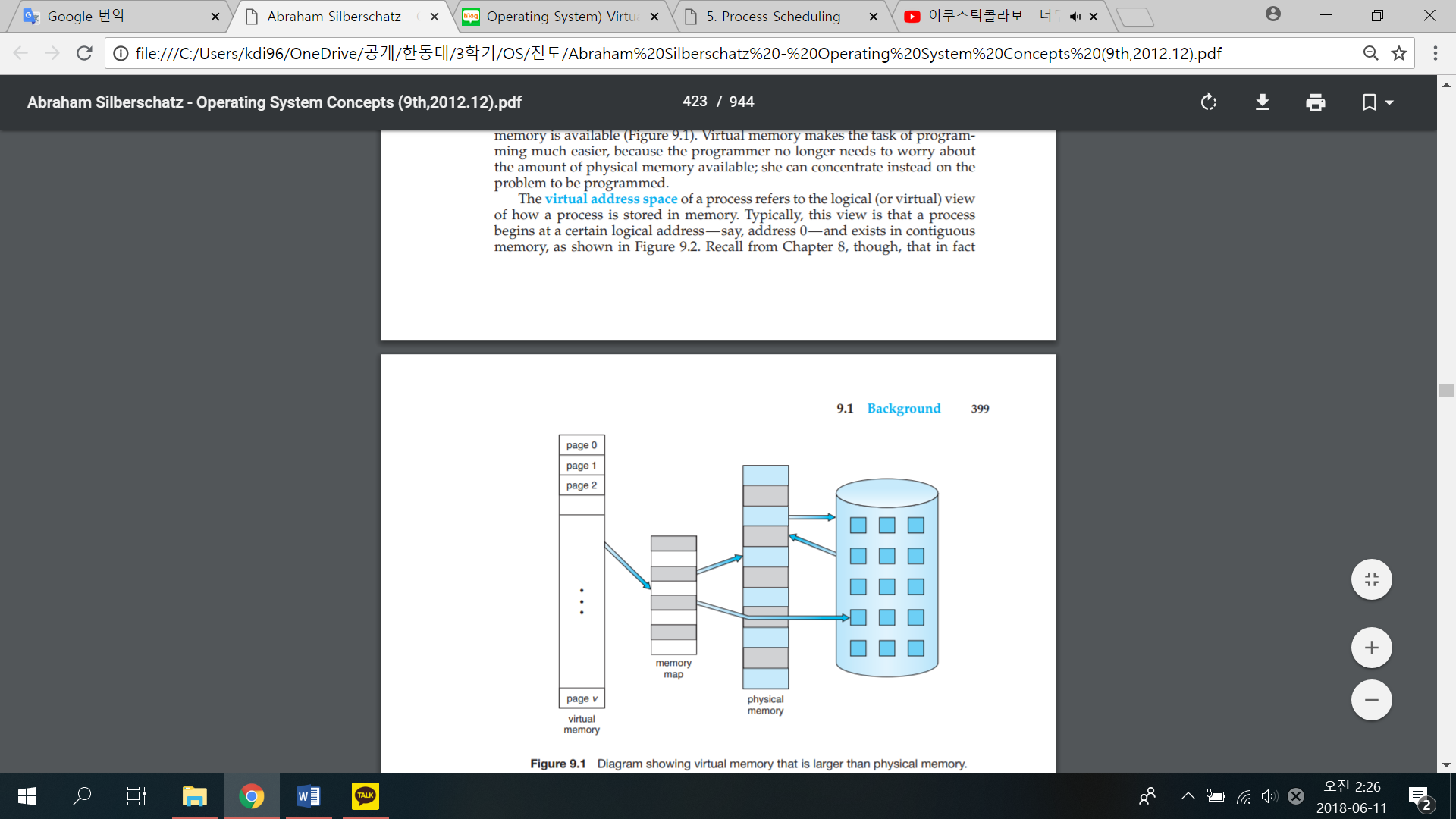
• 각 사용자 프로그램이 실제 메모리를 적게 차지할 수 있기 때문에 더 많은 프로그램이 동시에 실행될 수 있으며 그에 따라 CPU 사용률과 처리량은 증가하지만 응답 시간이나 소요 시간은 증가하지 않는다.

• 사용자 프로그램을 메모리에 올리거나 스왑하기 위해 더 적은 I/O가 필요하므로 각 사용자 프로그램이 더 빠르게 실행됨.

-> 프로그램 전체가 메모리에 있지 않은 프로그램을 실행하면 시스템과 사용자 모두에게 이득

정의

사용자가 물리적 메모리에서 인식하는 논리 메모리를 분리합니다.

이렇게 분리하면 물리적 메모리가 더 작은 경우 프로그래머를 위해 매우 큰 가상 메모리를 제공할 수 있습니다

※ Sparse address space

힙과 스택 사이의 큰 공백(또는 구멍)은 가상 주소 공간의 일부이지만 힙이나 스택이 커지면 실제 실제 페이지가 필요합니다. 이때, 공백이 포함된 가상 주소 공간.

2. Demanding Paging (Paging + swapping)

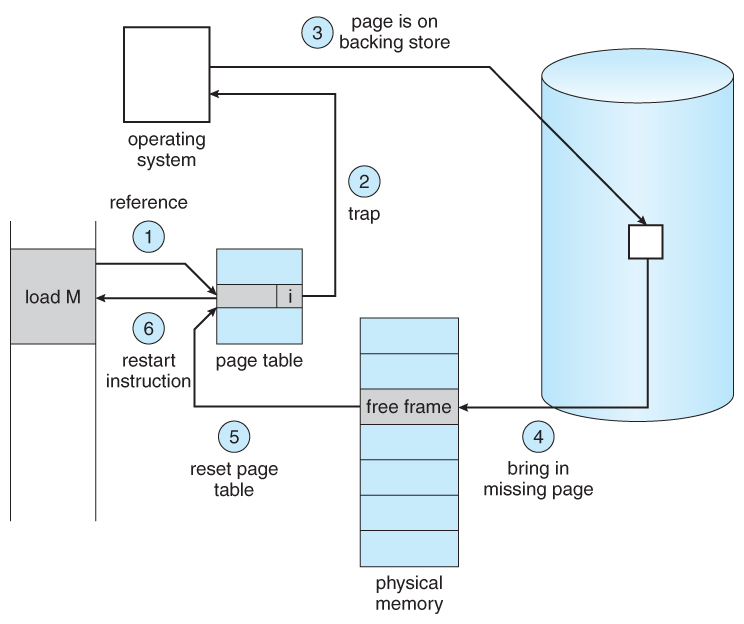
- Lazy swapper (or pager):

필요한 프로세스(페이지)가 아니면 swap하지 않는다. 즉, 프로세스를 swap 할 때 pager는 사용할 페이지를 추측해서 그 페이지만 메모리에 올린다.

(이 때, 메모리 또는 디스크상의 페이지를 구별하기 위해 H/W 지원이 필요.)

페이지를 선택했을 때, page의 유효성(page가 Physical memory에 있는지)을 봐야한다.

* Valid: 페이지가 physical memory에 존재한다. -> 그냥 실행
* Invalid: 페이지가 아예 존재하지 않거나 존재하지만 비어 있는 경우 -> Page Fault

• Page Fault: Physical memory에 page가 존재하지 않는 경우

1. PCB안에 있는 Page table을 참조하여 Invalid reference(Bad Address or protection violation)인지 단지 Physical memory에 존재하지 않는 것인지 확인한다.

2. Invalid reference라면 죽이고, 단지 Physical memory에 존재하지 않는 것이라면 Free frame 검색

3. Free Frame에 page를 swap-in해준다.

4. 새롭게 할당된 Page에 대해서 Page table을 업데이트한다.

valid bit: i->v,

frame# 업데이트

※ Pure Demand Paging

: 페이지가 필요할 때까지 메모리에 가져오지 않음. -> 사용 ㄴㄴ, locality of reference로 극뽁

• Performance of Demand Paging

-ma: memory access time (10~200 s)

-p: page fault 확률

-page fault time

• service page-fault interrupt

• read in the page: about 8msec -> 이것만 고려

• restart the process

문제풀이)

Memory access time: 200 nano sec

Page fault time: 8 msec

-> Page fault time에 크게 영향을 받는다.

※ COW(Copy On Write)

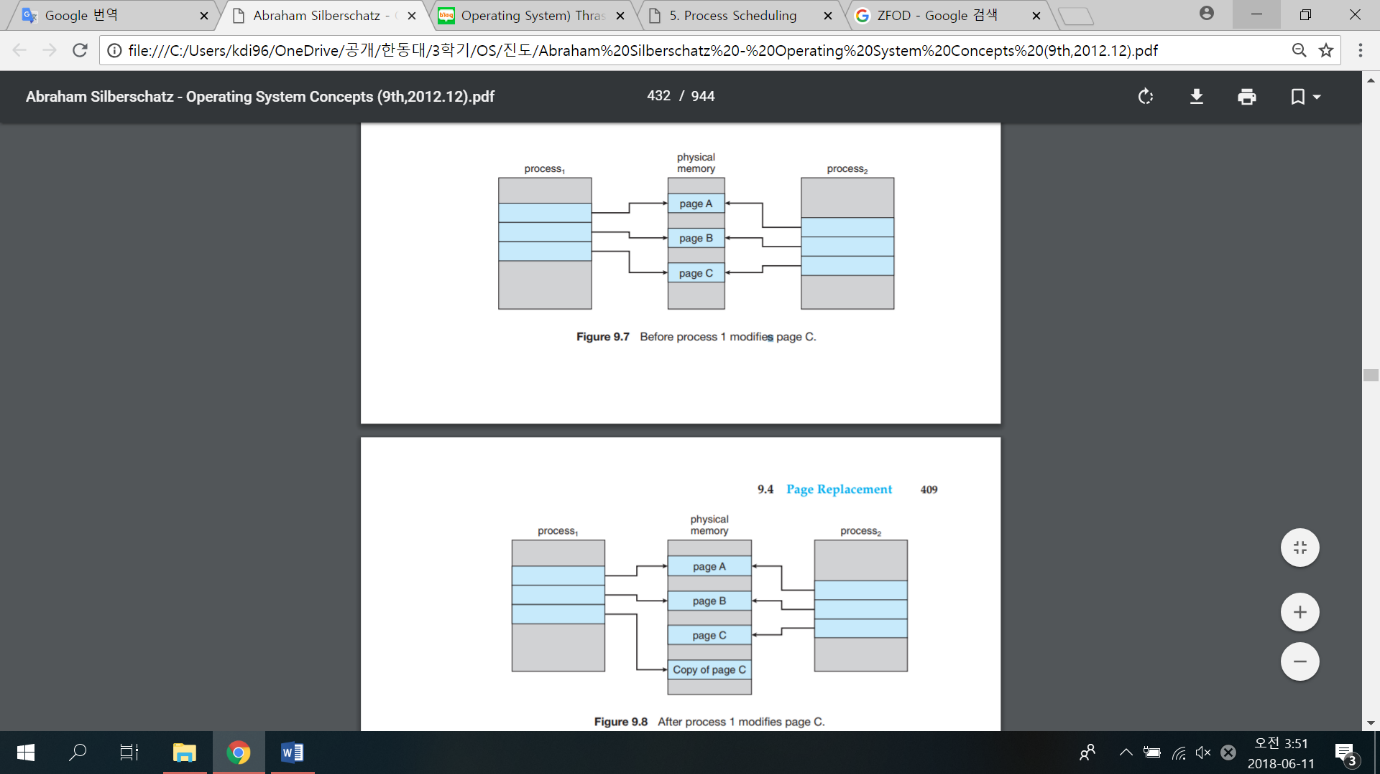
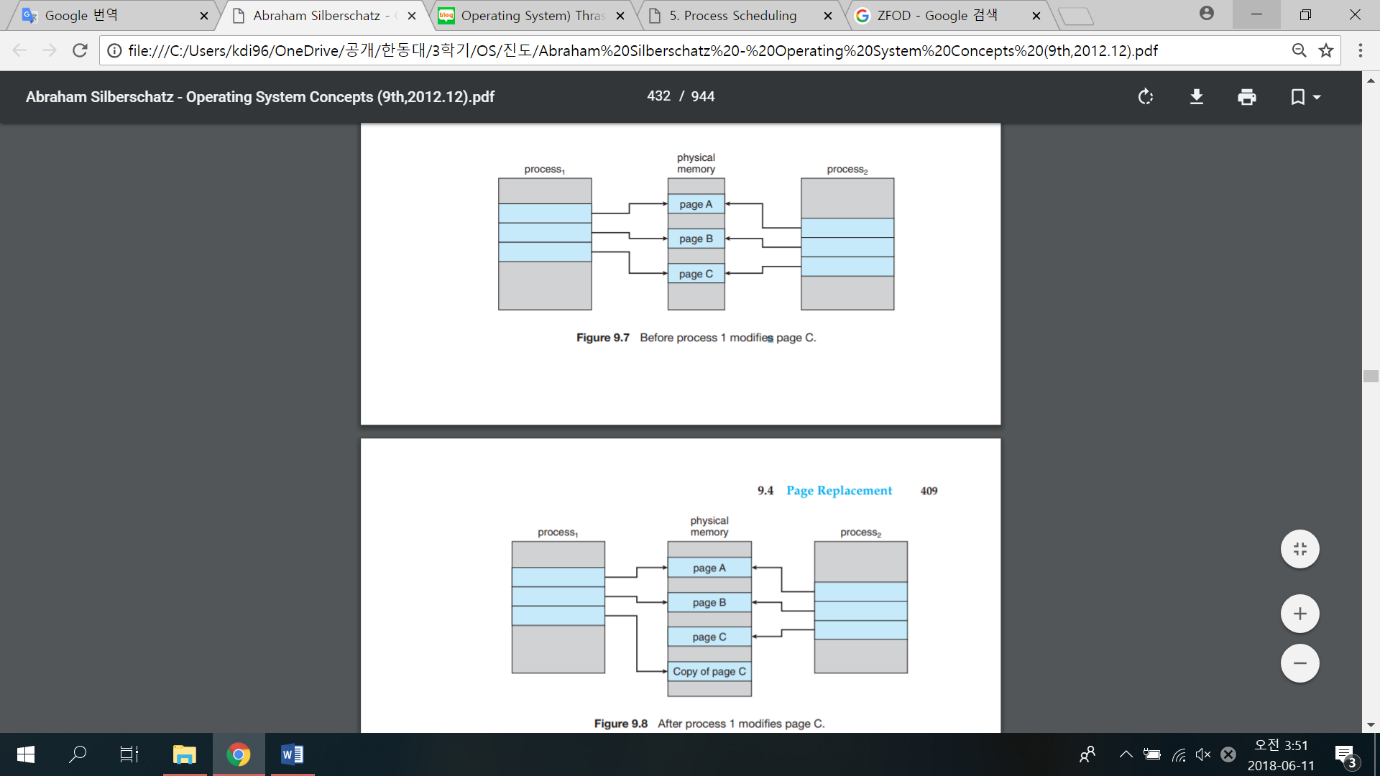
프로세스를 만들 때, 페이지를 복사하지 않고 공유하게 한다 -> 프로세스 생성시간 단축

그러다 두 프로세스 중 하나라도 공유된 페이지를 쓰면(변경하면) 새로운 페이지를 만들어준다.

※ ZFOD(Zero-fill-on-demand)

페이지가 copy-on-write를 사용하여 복제될 것으로 판단되면 사용 가능한 페이지가 할당될 위치를 기록하는 것이 중요합니다. 많은 운영 체제는 이러한 요청에 대해 사용 가능한 페이지 풀을 제공합니다. 이러한 비어 있는 페이지는 일반적으로 프로세스의 스택 또는 힙이 확장되어야 하거나 관리할 COW 페이지가 있을 때 할당됩니다.

운영 체제는 일반적으로 Zero-fill-on-demand로 알려진 기술을 사용하여 이러한 페이지를 할당합니다. Zero-fill-on-demand 페이지는 할당되기 전에 이전 내용을 지웁니다.



3. Page Replacement

목표: Page Fault를 최소한을 줄이기 위함

1. Page-Replacement Algorithm

페이지 폴트 처리 중 비어 있는 프레임이 없다면, 현재 사용되지 않는 프레임을 찾고 교체

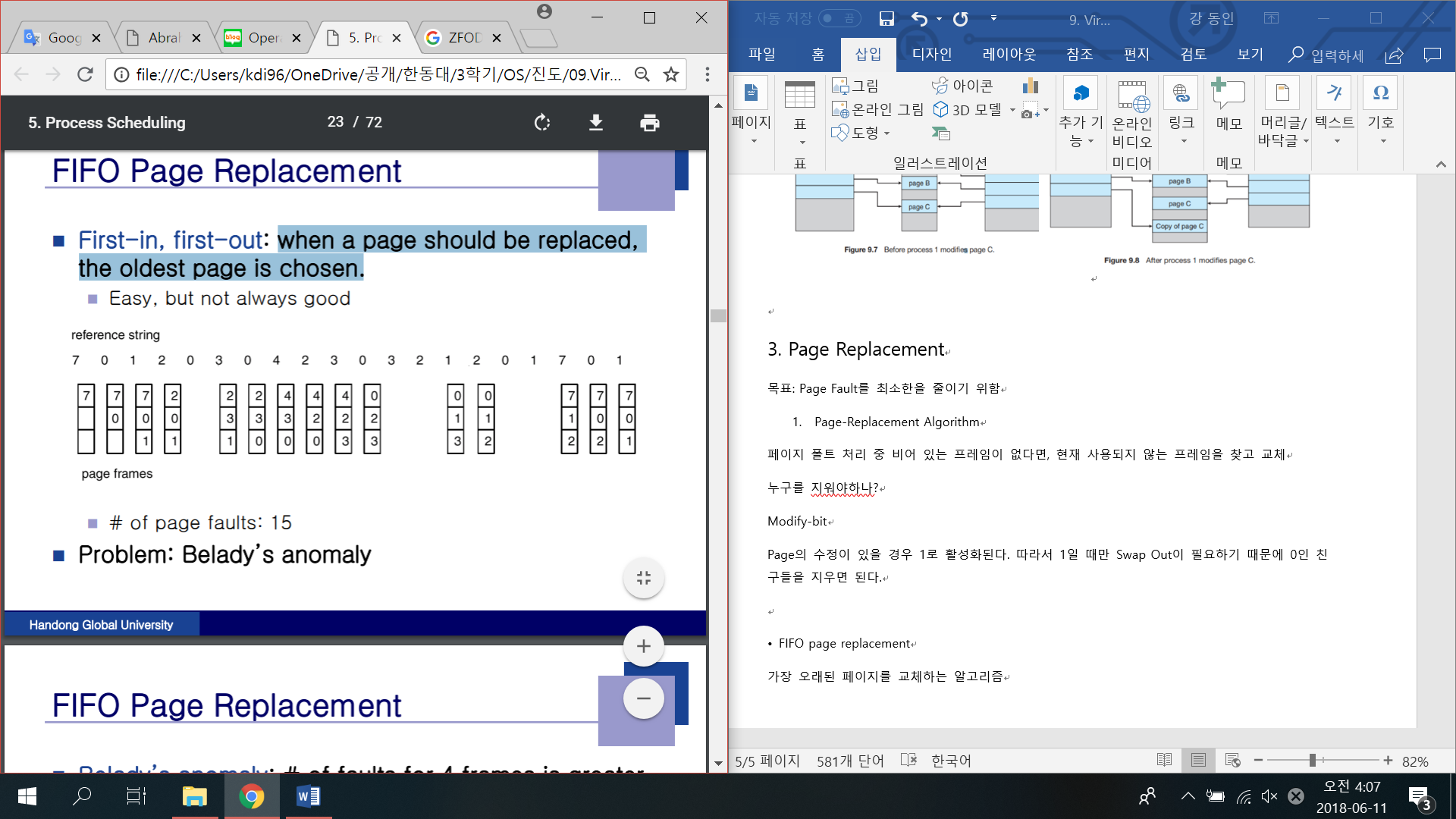
누구를 지워야하나?

Modify-bit

Page의 수정이 있을 경우 1로 활성화된다. 따라서 1일 때만 Swap Out이 필요하기 때문에 0인 친구들을 지우면 된다.

• FIFO page replacement

가장 오래된 페이지를 교체하는 알고리즘

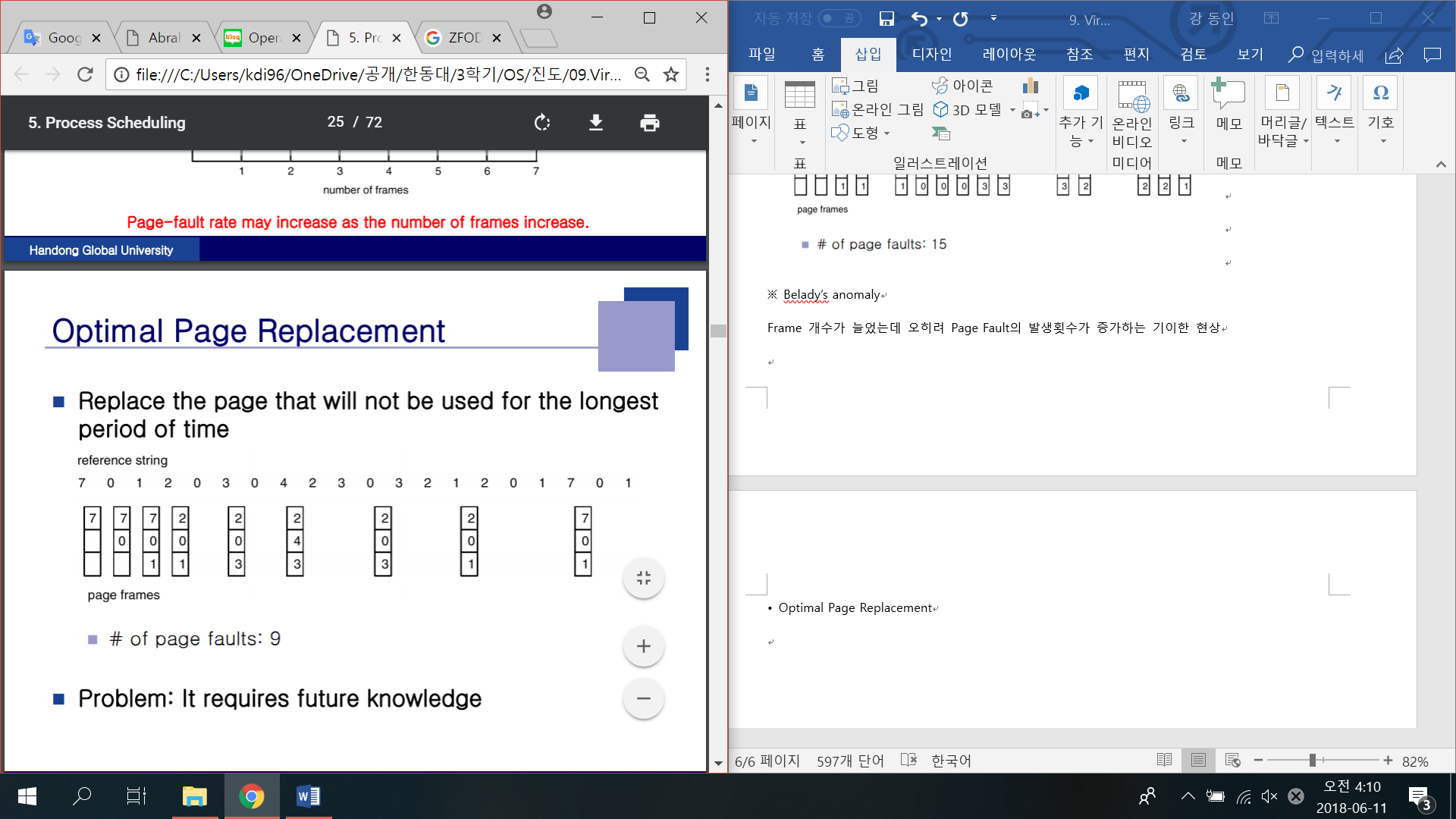


※ Belady’s anomaly

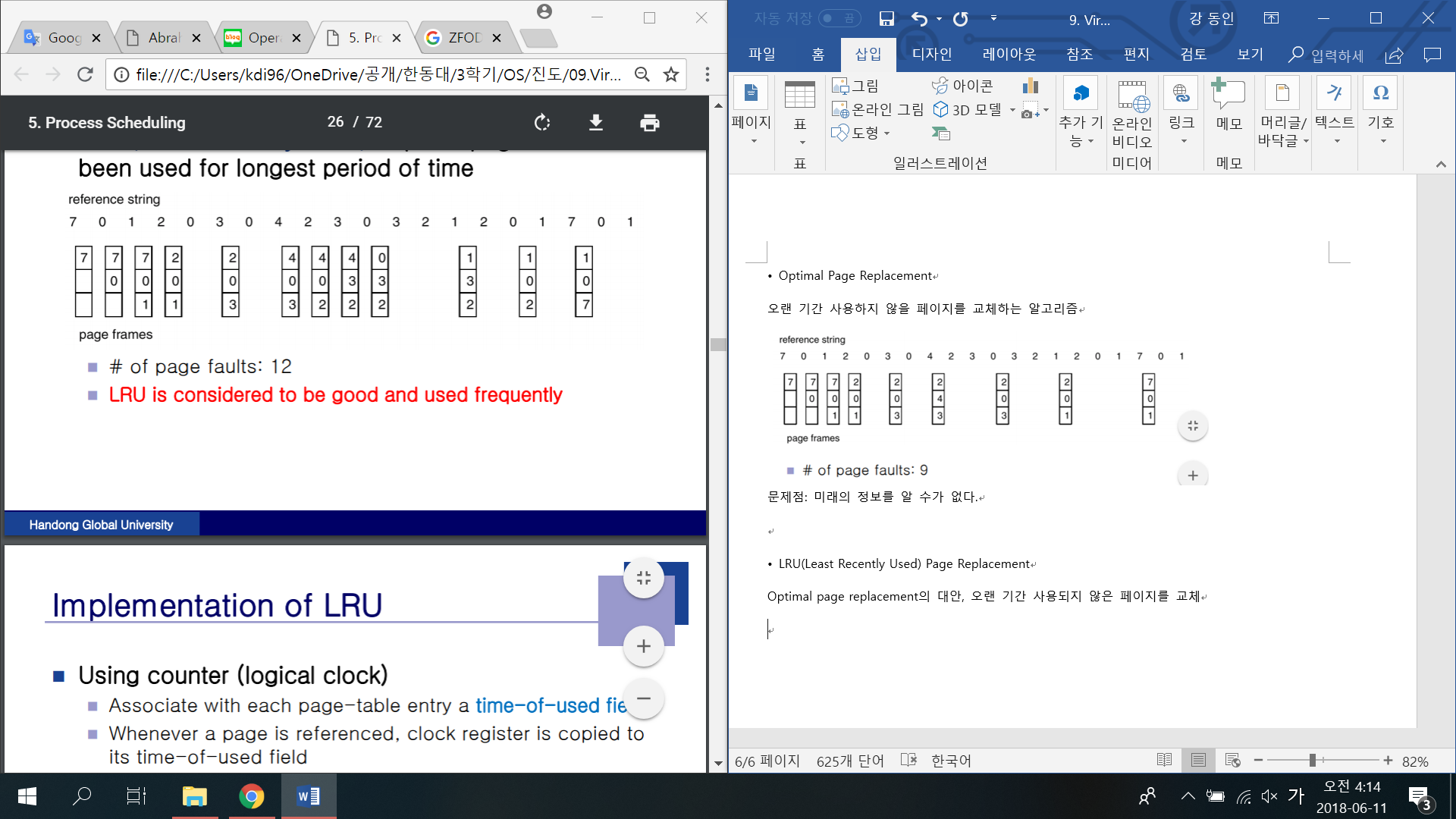
Frame 개수가 늘었는데 오히려 Page Fault의 발생횟수가 증가하는 기이한 현상

• Optimal Page Replacement

오랜 기간 사용하지 않을 페이지를 교체하는 알고리즘

문제점: 미래의 정보를 알 수가 없다.

• LRU(Least Recently Used) Page Replacement

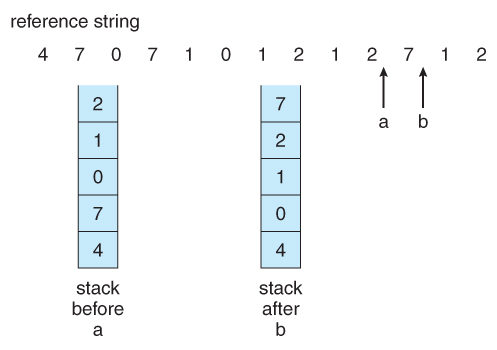
Optimal page replacement의 대안, 오랜 기간 사용되지 않은 페이지를 교체

구현

a. Counter (logical clock)

각 페이지 테이블 엔트리와 Time-of-used field를 연결하고 페이지가 참조 될 때마다 clock register가 time-of-used field에 복사된다.

b. Stack

Frame # 만큼의 크기의 Stack을 만들고 Page를 아래부터 채우다가 Page hit이면, 스택의 꼭대기로그 페이지를 올려준다. 이후 교체해야 할 상황이 오면 Stack의 맨 아래부분을 쫓아낸다.

※ Stack Algorithm: 프레임 개수가 n+1개인 모든 경우의 수가 프레임 개수가 n개인 모든 경우의 수에 포함되는 경우 –> Belady’s anomaly 발생X

LRU Algorithm은 Stack Algorithm이다.

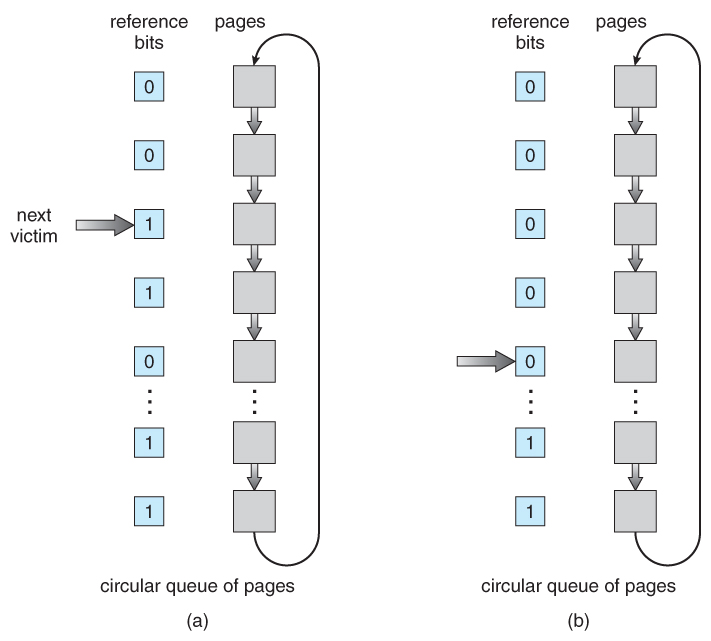
• LRU-Approximation Page Replacement

1. Additional-Reference-Bits Algorithm

Stack대신 Additional-Reference-Bits(1 이상)를 사용해 교체될 page를 정한다.

Memory Access가 발생할 때 마다, Reference bit을 올려주고 교체가 필요할 때 Reference bit이 가장 작은 page를 교체한다.

1. Second Chance Algorithm



Reference bit가 1인 bit를 clock hand 포인터가 0으로 바꾸면서 이동하다 0을 만나면 그 페이지를 교체한다.

• Page-Buffering Algorithms2

2. Allocation of Frames

여러 프로세스간 사용 가능한 메모리를 어떻게 할당할 것인가?

= 각 프로세스에게 몇 개의 프레임을 할당할 것인가?

• Equal allocation

전체 메모리를 frame 크기만큼 나눈 m을 n개의 프로세스에 동일하게 할당하는 방법

m/n frames for each process

• Proportional allocation

process 에 할당된 프레임 개수

: process의 크기

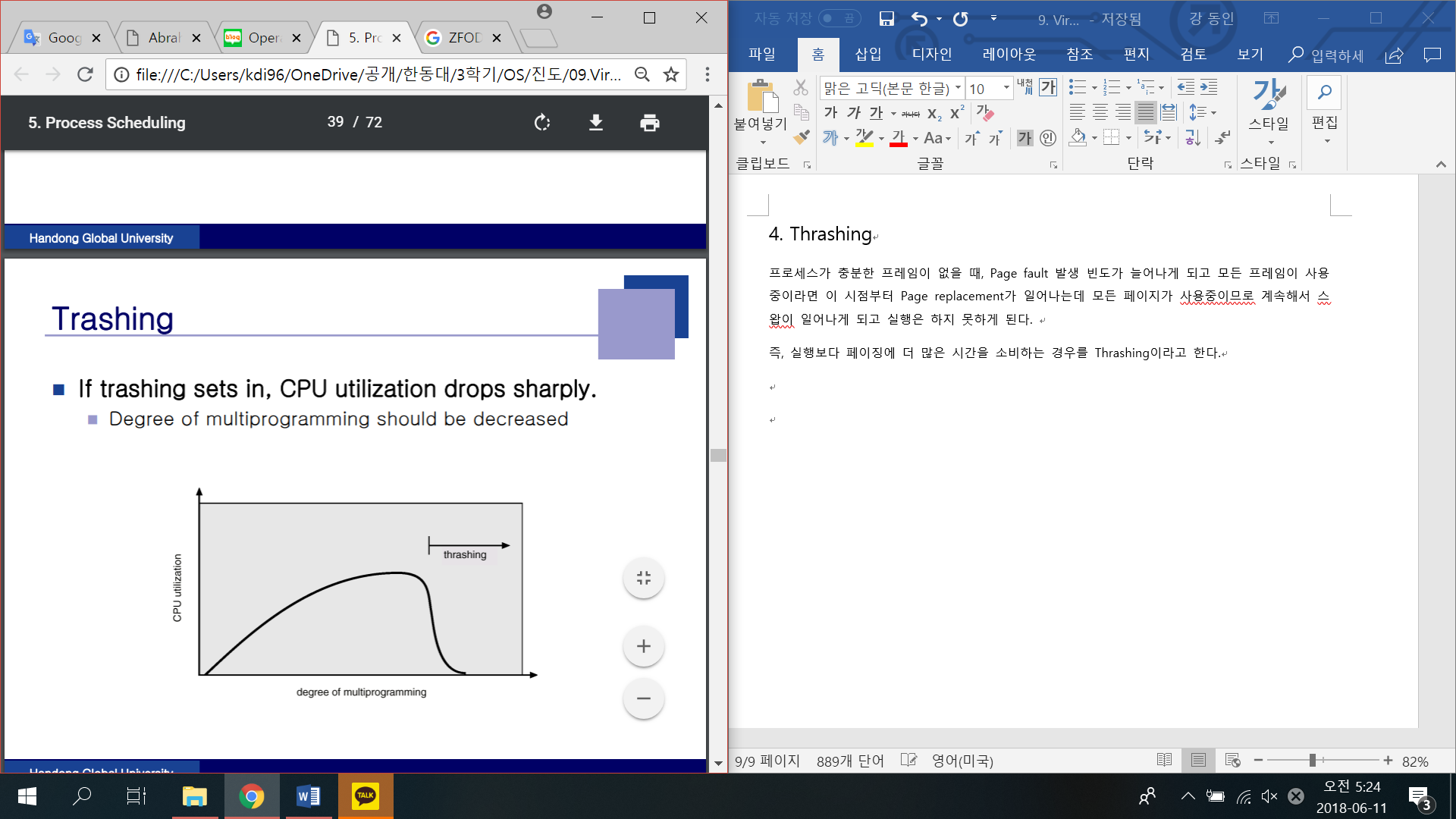
: 의 총합

Global replacement: 프로세스는 다른 프로세스에 할당 된 프레임을 포함하여 모든 프레임 집합에서 교체할 프로세스를 선택할 수 있다.

Local replacement: 프로세스의 프레임 수 변경 불가

4. Thrashing

프로세스가 충분한 프레임이 없을 때, Page fault 발생 빈도가 늘어나게 되고 모든 프레임이 사용 중이라면 이 시점부터 Page replacement가 일어나는데 모든 페이지가 사용중이므로 계속해서 스왑이 일어나게 되고 실행은 하지 못하게 된다.

즉, 실행보다 페이징에 더 많은 시간을 소비하는 경우를 Thrashing이라고 한다.

Thrashing이 발생하면 CPU 이용률이 감소하기 때문에 OS는 멀티 프로그래밍 수를 늘려 더 많은 Thrashing이 발생하게 된다.

어떻게 충분한 Frame 수를 알 수 있을까?

Locality: 현재 특정 시간에 집중되는 Page의 집합

프로세스는 하나의 Locality로부터 다른 Locality로 이동한다.

- Working Set model

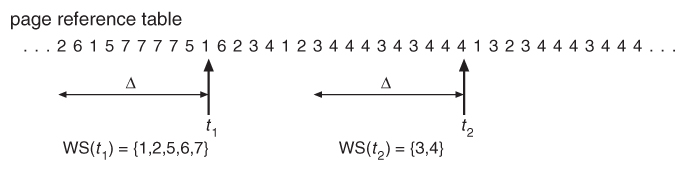
하나의 프로세스를 구성하는 모든 Page의 합을 구하기 어렵기 때문에 실제 working하고 있는 page의 합을 구하는 아이디어

• 세모: Working-set window, 고정된 Page reference의 숫자

• WS: 세모 내에서 Page들의 집합

• WSS: 프로세스에서 WS들의 합

• D: 모든 프로세스들의 WSS의 총합, 프로세스 전체의 요구량

만약, 세모가 너무 작다면 Locality를 포함하지 못할 수 있다.

만약, 세모가 너무 크다면 여러 Locality를 하나의 Locality로 판단하게 되어 접근하는 Page수가 증가하게 되고, Thrashing의 발생이 증가

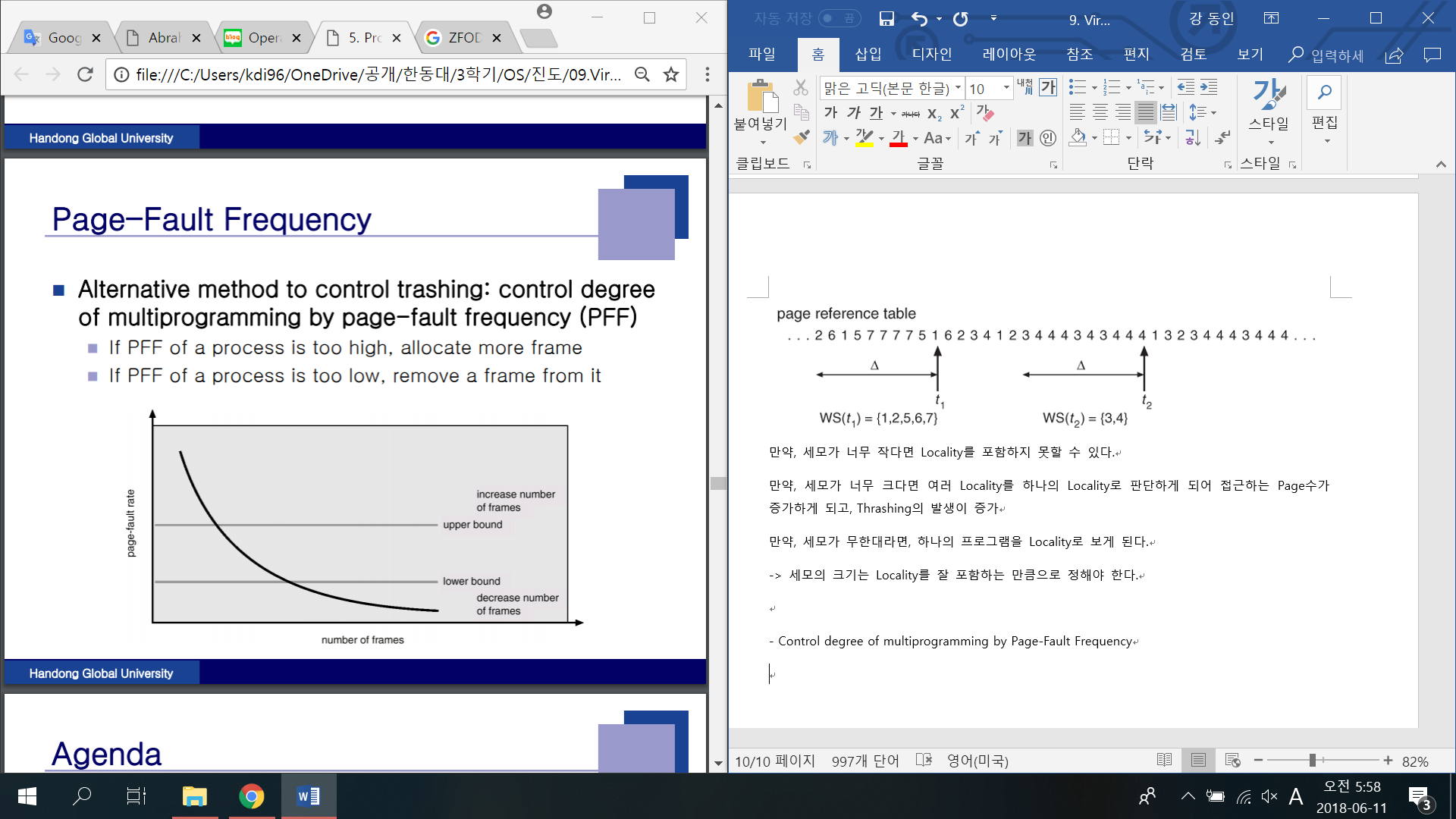
만약, 세모가 무한대라면, 하나의 프로그램을 Locality로 보게 된다.

-> 세모의 크기는 Locality를 잘 포함하는 만큼으로 정해야 한다.

D > m(시스템이 보유한 총 메모리 크기)

Thrashing 발생 -> 프로세스 하나 선택해서 조짐

- Page-Fault Frequency

PFF가 너무 높으면 frame을 더 할당

PFF가 너무 낮으면 frame을 제거

5. Memory-Mapped Files

6. Allocating Kernel Memory

Buddy System

무조건 2의 제곱수로 페이지를 할당한다.

Slab Allocation

커널은 다양한 데이터 구조를 저장하기 위해 다양한 크기의 페이지를 요구한다.

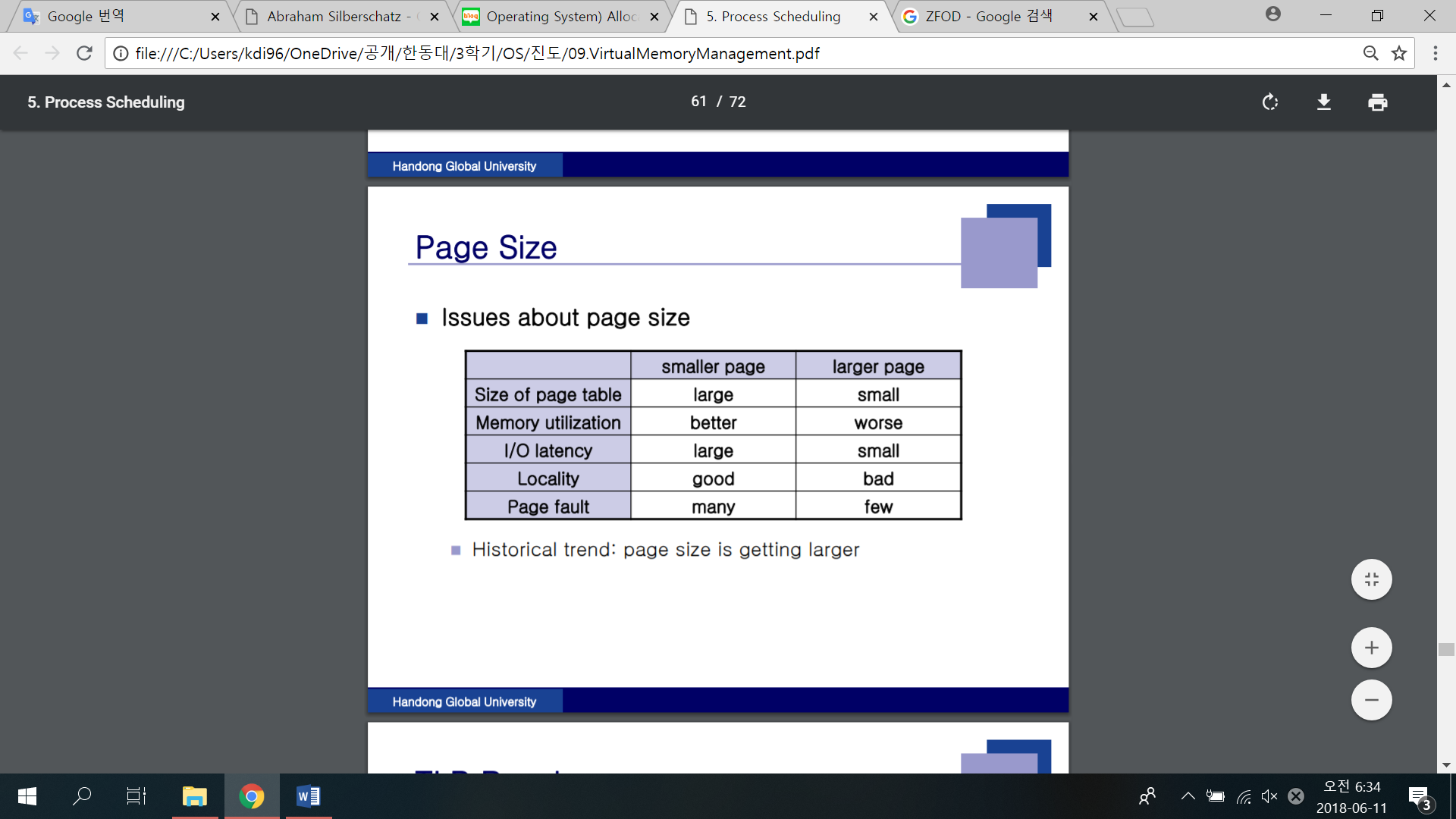
Slab: 하나 또는 그 이상의 물리적인 연속적 Page

Cache: 하나 또는 그 이상의 Slab

7. Other Considerations

- Prepaging <-> pure demand paging: work-set단위로 paging한다.

- Page Size



- TLB Reach: TLB에 있는 entry 개수 X 페이지 크기

- Inverted Page Tables

- Program Structure

- I/O Interlock