운영체제 10장
Virtual Memory
0. Background
Virtual Memory의 기본 개념
1. Demand Paging
proc 로딩 시, 모든 것이 메모리에 올라가지 X고, 필요할 시 Page Loading시켜 사용하는 것
2. Copy-on-Write
일단 Sharing하고 수행하다, 서로 공유할 수 없는 공간이 생겼을 때 Copy 진행한다는 것
목적 : 모든 것을 Copy하는데 걸리는 지연시간을 줄이는 것
3. Page Replacement (Demand Paging와 연관)
어떤 proc가 부담할 Page 필요할 때, 빈 공간 O시 할당하지만, 빈 공간 X시 메모리에 있는 어떤 Page들을 쫓아내고 자기가 빈 Page를 사용하게 됨
4. Allocation of Frames(Logical 관점에서 Pages를 다르게 부른 것)
proc에게 Page Frame을 할당해주는 기능
5. Thrashing
proc가 수행되기 위해서 Page가 로딩되어야 할 때, 일부 Page만 Physical Frame에 할당해 주는데, proc에게 적은 Frame을 할당해주어 Page Fault가 너무 자주 일어나는 현상
6. Memory-Mapped Files
File Access를 빨리하기 위해서, Memory에 매핑시켜 사용하는 것
7. Allocating Kernel Memory
Kernel도 Software의 일종이기 때문에, 메모리를 할당하는데 일반 proc에게 Page를 할당하는 것보다 특별한 방법
8. Other Considerations
Page에 관련된 디자인 이슈
9. Operating-System Examples

목표
-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.
-Describe how Linux, Windows 10, and Solaris manage virtual memory.
-Design a virtual memory manager simulation in the C programming language.

0. Background
(1) entire program rarely used하고 대개 partially-loaded program
prog 복잡도 및 처리량 늘어나다 보니 size가 커져 일부만 메모리에 로딩해 사용함
✓ more programs run at the same time
동시에 보다 많은 프로그램 수행 가능(CPU 사용률↑, Swapping 위한 I/O 발생률↓ - 많은 프로그램 존재해야 Swapping 늘어나서 I/O 발생 늘어나지 않나?)

(2) Virtual Memory(= Swap 공간이라고 부르기도 함, proc마다 가지고 있음)
Physical 메모리로부터 분리된 Logical 메모리 사용과 같음
(proc마다 넓은 연속적인 공간이 존재하는 것처럼 느낌 - 연속적 공간 사용)

#특징 및 장점
-수행하고자 하는 prog의 일부만 메모리에 로딩됨
(더 많은 수의 prog 메모리에 로딩되는 효과가 있음, Multiprogramming의 degree가 높음)
-각 proc마다 자원 공유를 더 쉽게 구현 가능
(Virtual memory가 physical memory로 매핑이 될 때, 공유하는 physical memory를 같은 곳을 가리키게 함으로써)
-프로세스 생성 쉬움
텍스트를 공유한다고 하면, 기존 텍스트를 그대로 사용하면 되므로 속도 및 효용성 향상
-더 많은 프로그램 동시에 수행 가능
-Swapping이 자주 일어나지 않아, 더 적은 I/O operation 발생

(3) Virtual-address space
proc마다 연속적인 공간이 있다고 생각(logical, physical memory개념 그대로 이용)
-VM 공간 -> Physical Memory 공간으로 mapping되는데, VM>므로 proc의 일부만 Load
✓ 이로 인해 다른 부분으로 넘어가 page fault가 났을 때, 필요한 부분을 Swap공간으로부터 가져와 Physical memory로 loading함
Demand paging: 이를 paging 기법으로 수행
Demand Segmentation :이를 segment 기법으로 수행

-Shared Library Using Virtual Memory
proc간 공유가능한 Library 존재(memory mapping통해 같은 physical 공간 가르킴로써)

1. Demand Paging
(Virtual 공간 > physical 공간) proc마다 일부만 physical memory에 올라가다 보니 proc 진행 중 없는 부분이 나올 때, 해당 page를 physical memory로 올릴 것을 요구하는 것

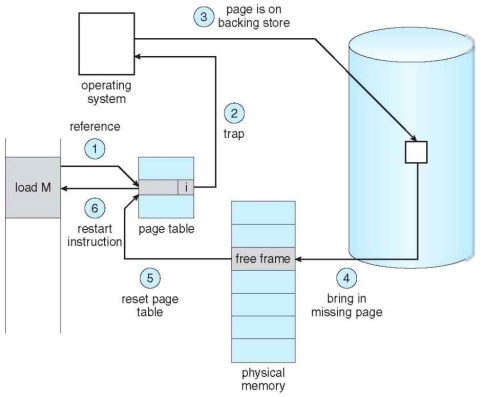
-장점
Swap으로 인한 I/O 연산 감소, 적은 physical memory 필요. 빠른 응답시간, 더 많은 이용자 수용 가능

*해당 내용 없어 Page 요구 시(있으면 그대로 참조), 2 가지 경우 존재
-본인proc 내 다른 부분 경우 : 그것을 참조해서 옴
-커널이나 다른 proc영역에 있어 참조가 불가능한 경우 : 해당 proc 종료(abort)

-Lazy swapper: page를 필요할 때마다 swap하는 것(메모리 공간이 부족하므로)
▶ swapper 기법 상에서의 표현(paging 기법에서는 demand paging)

(1) Basic Concepts
-MMU: virtual 공간을 physical 공간으로 mapping해주는 것
-(not)memory resident: page가 메모리에 있을 수도 있고 없을 수도 있음
-Valid-invalid Bit
v-memory resident하여 mapping 가능하다는 것
i-memory에 없어서 mapping불가 (자기 영역이 아니어 발생하는 Invalid와 다름)

(2) Step in Handling Page Fault
a. 자신이 사용하고자 하는 부분이 invalid인 경우: Page fault 발생
b. OS 다른 table 확인해 없는 2가지 경우 판단
-Invalid reference(자기 영역 아닌 것에 access) -> abort
-Just not in memory
c. free frame(빈 공간) 찾음
d. Swap 공간에 있는 page를 disk 연산 통해 memory 갖다 놓기
e. 해당하는 page table frame을 설정 및 수정함
(validation bit 설정 => v)
f. page fault가 발생한 부분으로 돌아가 다시 수행 시작



(3) Aspects of Demand Paging
-multiple page faults: instruction이 page 사이 걸쳐있는 경우가 존재하는데, 이 영역으로 인해 두 개의 page가 동시에 page fault 발생 가능

-**locality** of reference: prog 수행 시, 사용하는 부분을 집중적으로 사용하는 경향 있음

-Instruction Restart: 어떤 instruction이 블록에 있는 data를 다른 블록으로 옮기려 할 때, 여러 page에 걸쳐 존재하는 경우, 필요한 page 모두 메모리에 loading해야 함

-Free-Frame List: page fault 발생 시, OS는 physical frame을 page fault난 곳에 할당해줘야 하다 보니, OS가 일정한 free-frame을 List 형태로 관리하고 있음
(빈 공간 없으면, 존재하는 proc 쫓아내고<disk I/O발생>, 필요한 부분을 HDD에서 memory로 가져옴<disk I/O발생> - disk 공간 없을 시, maximum 2번 발생 가능하여 미리 관리함)

Stages in Demand Paging - Worse case[ppt 12개 과정]
트랩 발생(S/W INT) -> 수행 부분 저장 -> INT 구별 -> Disk에 있는 page를 memory에 가져옴 -> Disk I/O 요청 (device 다른 일하고 있으면 대기-차례오면 disk access-해당 블록 읽고 전달) -> 돌고 있던 CPU waiting 동안 재할당 -> I/O INT완료 후 신호 받음 -> 다른 사용자의 상태 저장 -> INT 구별(I/O종료되었음) -> page를 HDD에서 가져와 page table 조정 -> (proc 실행위해) CPU 할당 기다림 -> 아까 Save했던 내용을 restore해 proc 다시 수행

(4) Performance of Demand Paging
-Page Fault Rate p: 0<= p <=1
-Effective Access Time (EAT) 계산
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

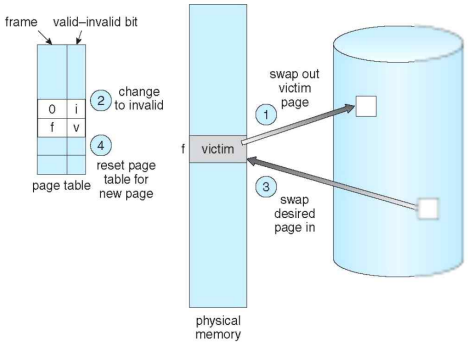
-Demand Paging Optimizations 방법
*Larger chunks : Swap 공간이 흩어져 있으면 관리하기 힘드니, 모여있는 넓은 공간인 larger chunks 이용 가능
*Demand page의 경우 prog binary(=instruction)라, read-only임(disk에서 바로 가져옴)
> read-only니 swap out 시 그냥 버려도 됨
(text(=instruction)에 대해 swap 공간을 따로 가지고 있지 않고 excutable file 이용)
*mobile 시스템 경우, SSD를 사용하여 swapping을 지원하지 않음(write 횟수 제한)

2. Copy-on-Write
일단 Sharing하고 수행하다, 서로 공유할 수 없는 공간이 생겼을 때 Copy 진행한다는 것
(parent-child proc에서 쉽게 찾아볼 수 있음)
목적 : 모든 것을 Copy하는데 걸리는 지연시간을 줄이는 것
-vfork() : fork()이후, exec()하게 되면 기껏 복사해놓은 것이 낭비되므로, vfork()를 통해 애시당초 공유하게 하여 그 낭비를 줄이는 것
> proc가 write하려고 한다면, 그 순간 필요한 부분에 대해 복사본(서로 공유할 수 없는 공간이 생김) 만들어 write 진행

3. Page Replacement 알고리즘
page replacement: Virtual memory demand page에서 발생, page fault때 Free-frame 할당해주는데 빈 공간이 X시, 양보를 통해 공간 확보 후 사용하려는 것 page in 진행
(Free-frame이 없을 때 진행되며, page 단위로 교체 진행)

#주요 개념
-over-allocation: 한 proc에게 너무 많은 page 주는 것 좋지 않음(자원 공평성)
-modify(dirty) bit: page out되었을 시, 변동 유무 판단하는 것
ex. HDD에서 가져온 A가 A'로 변동되었을 때, 이를 보호해 온전히 page out진행

#page replacement 단계
a. 그 page가 disk의 어디에 있는지 찾음
b. free-frame을 찾음
- free frame없는 경우, page replacement 사용 (victim frame 선정)
- victim frame에 쓰기 진행(dirty bit 확인해서 아니라면)
c. 원하는 page가 메모리에 로딩되면 update the page(invalid-> valid)and frame tables
d. page replacement 알고리즘 종료 후 proc 재 수행



4. Allocation of Frames
(1) Page and Frame Replacement Algorithms 소개
a. Frame-allocation algorithm
physical frame을 proc에게 할당하는 것
-얼마나 많은 frame을 각 proc에게 부여할 것인지
(한 곳에 많은 frame을 주면 좋겠지만, 다른 proc에게 피해가니)
-어떤 frame을 교체할 것인지

b. Page-replacement algorithm
page fault가 났을 때, 빈 공간 없을 경우에 어느 page가 양보를 할 것인지
-여러 가지 알고리즘이 존재하며, 그 성능 비교를 위해 particular string of memory references (reference string)를 수행하거나 reference string에서의 page faults 수를 계산함
※ page faults 수 $\propto$ 1/proc당 할당한 frame 수

(2) Page Replacement Algorithms
a. FIFO(First-in-First-Out)
 먼저 frame에 들어온 것이 양보하도록 하는 알고리즘
장점: 상대적으로 구현하기 쉬운 알고리즘
단점: 성능(page fault 줄이는 것)이 좋지 않음, Belady's Anomaly 문제
#Belady's Anomaly
 일반적으로 proc 할당 frame이 늘어나면 page fault가 줄어드나, FIFO에서 proc 할당 frame이 늘어나도 page fault가 줄어들지 않는 특정 경우 발생

b. Optimal
 page fault를 최소화할 수 있는 알고리즘
-앞으로 가장 오랫동안 사용되지 않을 page를 찾아 교체하는 방법
-전제: proc가 앞으로 사용할 페이지를 미리 알아야 함
(실질적으로 이를 알 방법은 없어 제대로 구현 불가능)
-특징: 모든 page replacement 알고리즘 중 가장 페이지 교체 수가 적음

c. LRU(least recently used page)
-가장 오랫동안 사용되지 않은 page를 교체하는 방법(과거 data 기반)
-Optimal 알고리즘이 구현 불가능하여, 비슷한 효과낼 수 있는 방법 사용한 것
-특징: 최적 알고리즘보다 페이지 교체 횟수 높지만, FIFO 알고리즘보다 효율적
-구현
 Counter(사용 시마다 증가)나 Stack(새로 ref된 것 계속 쌓고 아래 확인하며, 새로 access 한 것은 다시 위로 쌓기 아래 확인)이용

d. LRU-approximation
LRU 구현이 어려우므로(Stack overhead 큼)
-reference bit 이용 LRU 구현 방법(초기 ref bit를 0으로 설정하고, refer 되었을 경우 1로 설정해 단 하나의 bit로 ref 되었는지 아닌지 확인용)
-second-chance(두 번째 기회) algorithm 이용 LRU 구현 방법<=Clock algorithm>
Clock이라는 화살표가 돌면서 0->1(해당 위치에 새로운 page 오도록)로, 1->0(해당 page는 메모리에 남겨두지만)으로 바꿈

#Enhanced Second-Chance Algorithm
 LRU-approximation을 더 개선한 알고리즘(modify bit도 함께 고려)
(reference, modify<dirty, physical memory에 올라온 뒤에 변화 유무>)
- 경우 4가지: (0,0), (0,1), (1,0), (1,1)
modify가 되지 않았으면, replace가 될 때 page-out 할 필요가 없음(원본 HDD에 존재)
-> 그냥 덮어씀(page-in이라는 하나의 I/O operation으로 가능)

(2) Counting Algorithms
 ref한 횟수를 가지고 따지는 것

a. LFU(Lease Frequently Used) Algorithm
여태까지 사용빈도가 제일 떨어지는 것 교체

b. Most Frequently Used (MFU) Algorithm
여태까지 사용빈도 높았다면, 앞으로는 사용되지 않을 가능성이 높다고 판단하여 그것 교체

(3) Page-Buffering Algorithms
-OS는 free frame 어느 정도로 유지하는 것 중요함(page replacement 잘 일어나지 않도록)
-Modified pages(=dirty): 당장 replace에 사용하지 않아도 빈 공간 확보 차원에서 free frame pool에 넣어놓음.
-> 해당 공간 replace해서 reuse하기 전 다시 참조해서 사용 가능(Disk Loading 없이)

(4) Applications and Page Replacement
App과 page replace 연동되는 경우
OS에서 page replacement 관리하여, 모든 application에서 공통적으로 Page Replacement 알고리즘을 적용(일반적으로 적용하는 방식)
>> But, DB나 어떤 application은 일반적으로 적용되는 방식 대신, application의 성격에 최적화한 memory page frame을 사용하는 알고리즘 이용(prog speed 향상 가능)
#Raw disk mode
[Application이 계층 따라 M/W, OS에 의존 X고, H/W에 바로 접근하여 최적화 방식 정의]

(5) Allocation of Frames
proc에게 physical frame을 얼마나 할당할 것인지
-각 proc는 기본적으로 동작 위한 minimum frame 필요함

a. Fixed Allocation
proc에게 고정된 frame 할당하는 방법

*Equal allocation
모든 proc에게 동일한 수의 frame을 할당해주는 것

*Proportional allocation
proc 크기에 비례해서 frame을 할당해주는 것

b. Global vs. Local Allocation
page replacement가 일어날 때, replace 대상을 찾는 방식
Global: 서로 어느 page가 양보할 것인가를, 모든 proc에서 따지는 것
Local: 서로 어느 page가 양보할 것인가를, 자기 page 내 다른 proc에서 따지는 것

(6) Reclaiming Pages
page replace할 때, 어떤 threshold 값이 존재한다는 것

-자기 proc가 필요로 하는 일정(적절)한 수준의 frame을 할당하는 것
(maximum threshold, minimum threshold 2개 값 사이 존재하도록 그 범위 값을 벗어나면 다시 증가, 감소하도록 하여 조정함)

(7) NUMA(Non-Uniform Memory Access)
컴퓨터 아키텍쳐 - VM page 기법 연관지어 생각해본 것
Memory 사용할 때, 같은 값이면 local에 있는 memory 수행하는 것이 좋음
>> proc 관리할 때 또한, CPU가 가까운 memory에 loading 된 proc를 수행하는 것이 좋음
만약 여러 core가 같이 사용하는 memory 존재한다면, core나 CPU끼리 lgroups(인접한 그룹)을 지어 memory 사용하는 것이 좋음

5. Thrashing
page-fault rate가 높음 -> prog 진행 시간보다 이것 처리에 시간을 너무 많이 쓰게 됨
(proc에게 너무 적은 수의 frame을 할당하여 page-fault가 너무 자주 발생하는 것)
-단점: 낮은 CPU 활용성으로 이어짐

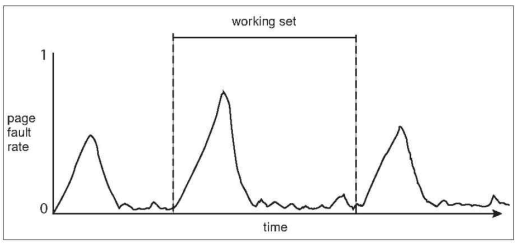
(1) Demand Paging and Thrashing
Locality model : 앞서 언급했던 적절한 수준
proc가 커다란 공간을 다 사용하지 않고 일부만 memory에 올려놓음
-> 프로세스가 실행될 때 항상 어떤 특정 지역에서 메모리를 집중적으로 참조함
이 locality는 overlap될 수 있음
#Locality 관점에서 Thrashing
$\sum$(locality의 크기) > 총 memory 크기

(2) Working-Set Model
최근(Δ)에 사용하는 frame의 집합
WSS(i): Proc P(i)의 working-set<현재 사용하는 frame(=page)들의 집합>
D = $\sum$WSS(i) $\equiv$ total demand frames (시스템이 가져야될 frame 개수)
#Working-Set 관점에서 Thrashing
D > m (sys이 가지고 있는 physical memory)
✓ memory 용량 늘리거나 memory에 올라와 있는 proc 개수 줄임(multiprogramming ↓)
✓ 결론 : working-set이 memory에 올라가 있어야 page-fault rate ↓ 가능(Thrashing X)

#Working-Set을 Tracking 하는 방법
page들의 ref를 기록해, 최근 일정기간 동안 어떤 page들이 ref했는지 추적 가능

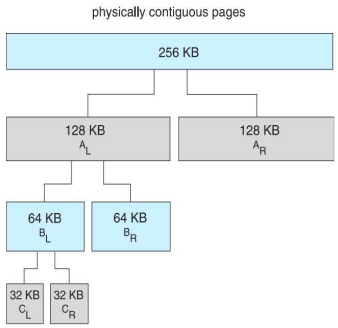
(3) Page-Fault Frequency(PFF)
-upper bound : 최소한 할당해주어야 하는 frame 수 기준
-lower bound : frame을 더 할당해주어도 page-fault 거의 감소하지 않은 기준
✓ page-fault와 할당 frame 수는 이 bound 사이 수준으로 유지하는 것이 적절함

*Working-set과 page-fault rate 관계
Working-set마다 수행 처음에는 아무것도 loading 안되어 있으므로 page-fault 많이 발생하다가 이후 점차 줄어드는 형태가 반복

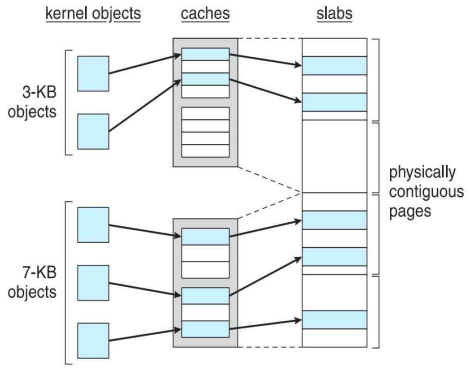


//6. Memory-Mapped Files
7. Allocating Kernel Memory
커널에게 어떻게 메모리를 할당해주면 좋은지
-커널은 특성상 우선순위가 높고, 관리하는 data size에 큰 것들이 많이 존재하여 작은 frame을 할당해주는 것보다 연속적인 frame을 커다란 page 하나로 관리하는 것이 효율적임

*그 예시들
(1) Buddy System
page의 크기를 커다랗게 잡고 필요에 따라 2의 n제곱 유지하면서 조정하는 것



(2) Slab Allocator
커널이 사용하는 Object에게 연속적인 공간 할당해주는 것
(object에게 Cache 할당, Cache에는 Slab 할당 등 계층적으로 할당 관리)
Cache: 하나 또는 여러 개의 연속적인 Slab
Slab: 여러 개의 연속적인 Page



*Slab Allocator in Linux
Slab에 3가지 State 존재
[Full – all used, Empty – all free, Partial – mix of free and used]
Partial 먼저 사용 -> Empty 순으로 할당해주는 기법
#여러 가지 변형 존재
-SLOB: limited memory를 가진(복잡한 자료구조 유지 힘듦) system 위한 것4
Simple List of Blocks(small, medium, large objects로 나눠 간편하게 관리)
-SLUB: page 구조에서 per-CPU queues, metadata 제거한 performance-optimized Slab

8. Other Considerations
page 기법에서 또 다른 고려요소들
(1) Prepaging(<=>Demand Page: 자원 필요 시 사용, page-fault 발생)
미리 필요할 것이 예상되는 page를 미리 memory에 올리는 것
장점: page-fault 방지 가능
단점: 예측의 정확도가 문제됨
-연산
s: page의 개수, α: page 사용 비율(1-α: 미리 사용될 page 잘못 예측한 것)
*Prepaging의 성능(둘 중 무엇이 큰지에 따라)
s * α의 비용 만큼 page-fault 감축 <-> 필요하지 않은 page에 대한 s * (1- α)만큼 손실
>> α가 0에 가까울수록 prepaging 효과 떨어짐

(2) Page Size
page 크기가 작으면, internal fragmentation을 줄일 수 있으나, page 수 늘어나고 page tab의 entry가 증가해 page 관리 위한 overhead가 늘어남
-Size: 보통 1K(2^10)~4K(2^12)로 설정하고, 특별히 4M(2^22, OS설치)로 크게 설정 가능
-Page Size 고려 요소
Fragmentation, Page table size, Resolution, I/O overhead, Number of page faults, Locality, TLB size and effectiveness

(3) TLB Reach
TLB: Page table의 Cache
-TLB Size: TLB라는 Cache에 들어갈 수 있는 page의 entry 수
-TLB Reach = (TLB Size) * (Page Size) <- 고정 or 변동
✓ Working-set이 TLB Reach 내 들어갈 수 있다면 보다 효율적임
+ Multiple Page Size를 제공하는 것이 더 효율적(복잡도↑)

(4) Program Structure
Prog은 각 행마다 한 page 씩 저장되는 구조로 형성되어
i가 행, j가 열로 설정하여 탐색했을 때는 행마다 page가 memory에 올라가, 128 page-fault가 발생하지만, i를 열, j를 행으로 설정하여 탐색했을 때는 매번 memory에 올라가는 page가 달라지므로 128*128의 page-fault가 발생함.

(5) I/O interlock and page locking

-I/O interlock

Device driver나 network같은 I/O를 prog에서 사용할 때, CPU와의 속도 차이로 인해 buffer 공간을 메모리에 두어 느린 I/O가 buffer에 가져오면, CPU가 그것을 가져감

✓ buffer가 사용되는 page가 replace 당하면, prog 진행 중 문제 생길 수 있어, I/O interlock을 걸고 이 공간에 대해서는 replacement 발생하지 않도록 설정

9. Operating-System Examples

(1) Windows

기본: 클러스터링 사용하는 Demand paging(page-fault 발생 시 근처 page 같이 가져옴)

-working-set 개념 활용

-automatic working-set trimming 이용(working-set minimum, maximum)

시스템이 free memory가 충분이 있다고 판단하면, working-set 늘려주고 등

(2) Solaris

기본: Demand paging

-page-fault 발생 시, 두 번 I/O access하는 것을 방지하기 위해 free-memory를 일정한 수준으로 유지

-Scanrate: page-replacement algorithm에서 clock이 얼마만큼 빨리 움직일지 (slowscan, fastscan 사이 조정)