# 10. Virtual Memory 1

CPU는 **현재 실행할** `Instruction` **과 해당 명령어가 접근하고자 하는 데이터만 Memory에 있으면 Program을 동작**한다.

- 이에 따르면, process의 address space에 해당하는 모든 Page가 메모리에 있을 필요는 없다.
- 따라서 동시에 서로 다른 process의 page가 memory에 올라와 있을 수 있다.

하지만 이론은 이와 다르다.

`Von Neumann architecture` 에 따르면, 실행되는 모든 program의 address space 전부가 memory에 올라와야 한다.

- 그렇다면 우리는 **필요한 Page만 Memory에 올리면서 Von neumann Arichitecture의 원칙을 유지**하는 방법을 찾아야 한다.
- 이를 위해 **OS는 Process의 address space에 해당하는 모든 Page가 이미 메모리에 올라온 것처럼 사기를 치고 이를 뒤에서 지원한다.**
- `Virtual memory` 를 사용하는 이유에 해당한다.

Virtual memory는 아래와 같은 목적을 가지고 동작한다.

`OS` 는 CPU와 Process가 process의 address space에 해당하는 **모든 page가 memory에 실제로 있다고 믿고록 cpu, process가 요청하는 데이터가 실제 DRAM에 없을 때, 이를 해결**한다.

- Physical address space를 관리한다.
- 기존에는 process가 physical address 영역을 접근할 수 있어 DRAM 영역이 확실히 구분되고 보호되었지만, **VM 사용 시, OS만이 DRAM을 관리하기 때문에 더욱 효율적으로 관리할 수 있다.**
- EX) OS는 어떤 process의 address space가 여러 process에 의해 공유될 수 있도록 한다.

`CPU / Process` 는 모든 Address space가 DRAM에 있다고 생각하고 동작한다.

- 실제 DRAM에 접근하지 않으며, Logical address space만을 관리한다.

즉, **Virtual memory는 Logical memory와 Physical memory를 분리**한다.

Virtual memory는 아래의 것을 지원한다.

- Program이 실행될 때 필요한 것만 메모리에 올린다.
- **Logical address space가 physical address보다 커도 정상적으로 동작할 수 있는 이유**이다.
- Address space가 여러 process 사이에 공유가 가능하도록 한다.
- 더욱 효율적인 process 생성이 가능하도록 한다.
  - `Fork()` 시, 기존과 다르게 parent와 child에 각각 address space를 할당하지 않고, child가 parent의 address space를 공유할 수 있도록 한다.
  - `exec()` 호출 시, 두 process의 address space가 분리된다.
    - Memory 복사 overhead가 감소한다.
- Page는 swap in / out 될 수 있다.

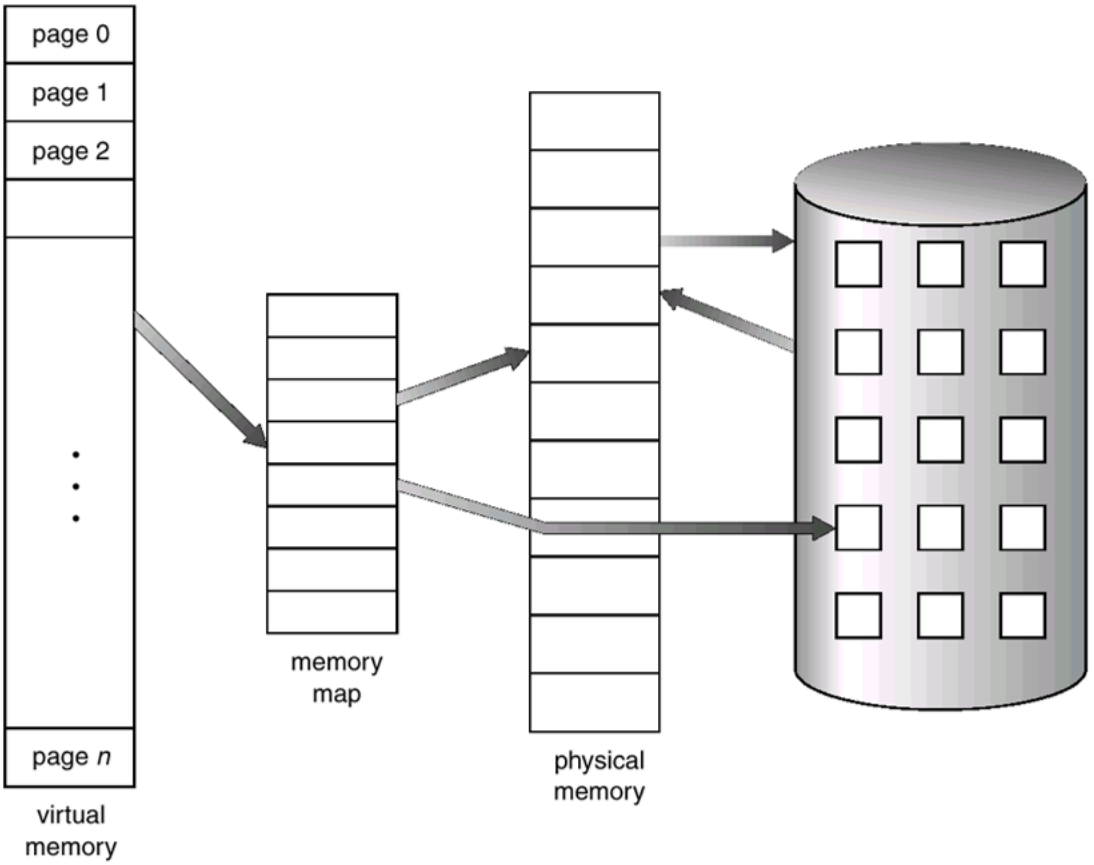
이후 모든 Virtual Memory에 관련된 내용은 **Paging 기반**이다.

- Swapping도 address space 기준이 아니라, Page 기준으로 동작하면 된다.

**Virtual Memory를 구현하기 위해선 아래 두 가지가 필요하다.**

1. `Demand Paging`
   - CPU가 접근하고자 하는 Page가 메모리에 존재하지 않는다면, 메모리로 올리는 행위를 빠르게 처리
   - Demand Segmentation도 존재한다.
2. `Swapping`
   - 사용하지 않는 page를 memory에서 쫓아낸다.

**Virtual Memory의 효과는 다음과 같다.**




- **Process의 Logical address space가 physical address space(DRAM)의 크기보다 큰 경우에도 동작할 수 있도록 한다.**
- 만약 VM이 없다면 아래와 같은 상황이 발생한다.
  - 처음에는 page table에 Invalid bit 인 entry가 많다.
  - 실행하면서 동적 메모리 할당, function call 등으로 address space의 크기가 점점 커진다.
  - 어느 순간부터 DRAM의 크기보다 커져, Heap allocation  실패하는 등 process가 동작하지 못 하게 된다.

# Demand Paging

**CPU가 해당 page에 접근할 때, 그 Page를 메모리에 올리는 방법**

**아래 네 가지의 장점이 있다.**

- `Less I/O 발생`
  - 필요한 것만 Disk → Memory
- `Less Memory`가 필요
- `빠른 응답성`
  - Program 시작 시, 전체 address space가 메모리에 올라가는 것을 기다릴 필요가 없기 때문이다.
- `더 많은 User`가 동시에 실행 가능

**Page가 필요해졌을 때, 해당 Page를 reference 해야 한다.**

1. `Invalid reference` → abort
2. `Not in memory` → Memory로 가져옴
   - Page table의 Valid / Invalid bit을 활용해 확인한다.

**Lazy swapper라고도 한다.**

- 필요하지 않을 때까지는 메모리에 올려놓지 않다가, 필요해졌을 때 올리기 때문이다.
- Page를 다루는 swapper를 `Pager`라고도 한다.
- Process가 suspend에서 해제된 경우에도 지금 당장 필요로 하는 부분만 Memory에 올린다.

**Valid / Invalid bit**

- Page table의 각 Entry가 가지고 있다.
- `Invalid`는 여러 가지를 의미한다.

**Invalid의 의미**

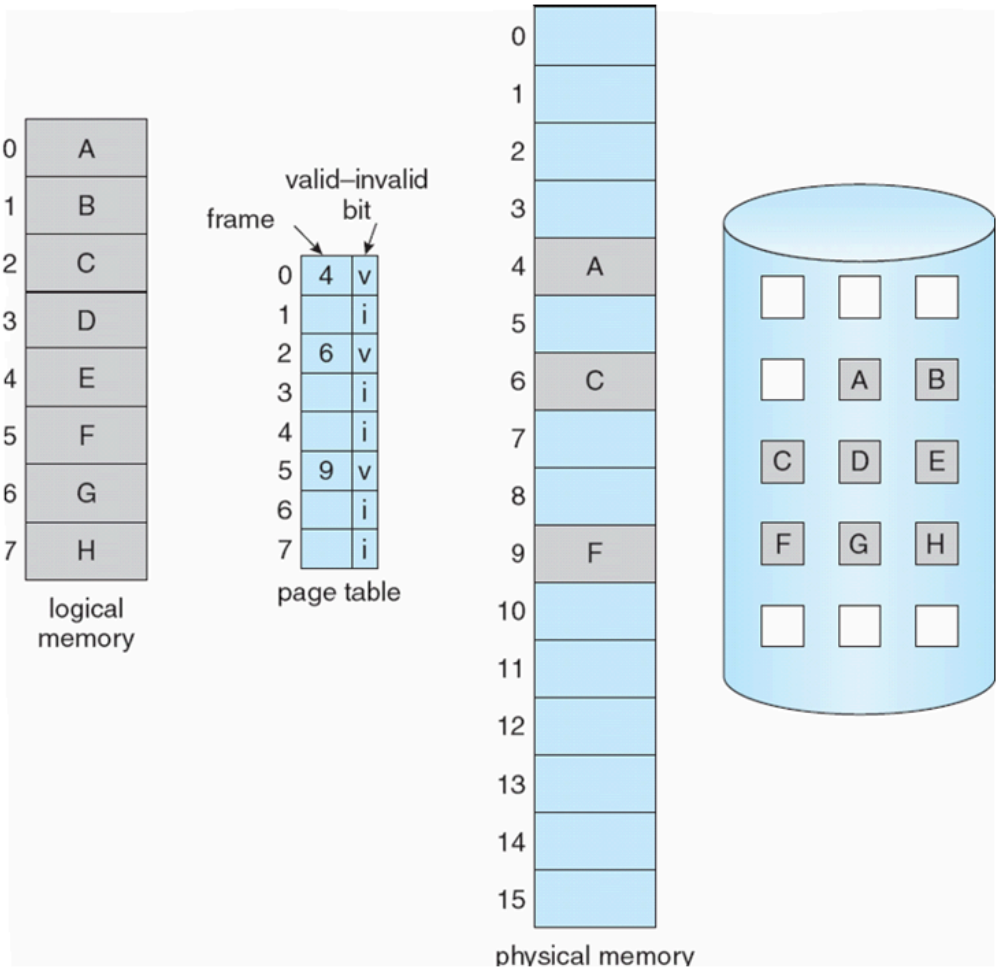
1. `illegal` : Logical address의 범위를 벗어나는 등의 명백한 잘못된 접근

2. `Not in Memory` : Memory에 현재 존재하지 않는 경우
3. `Obsolute` : Disk로부터 Memory에 올라와 있는 데이터의 Copy가 Disk에 저장되어 있는 원본 데이터 값이 변경되어서 참조하지 말아야 하는 경우

- 별개로, **모든 page table enrty는 Invalid로 초기화된다.**

**Address translation 과정에서, 만약 Page table entry가 Invalid bit이라면, Page fault를 발생한다.**

- Address space → CPU가 Page 요청 → Logical address → MMU → Page table → Invalid
- Invalid → Page fault → Page fault handler 호출



- Memory에 올라온 데이터는 복사본이고, Disk에 원본이 존재한다.
- **Valid-Invalid bit가 Memory에 올라와 있는 곳만 Valid인 것을 확인할 수 있다.**

**Page fault 처리로 들어가기 전에 먼저 Demand paging의 과정을 살펴보자**

1. 먼저 잘못된 접근을 하고 있는 지부터 판단한다.
2. 잘못된 접근이 아니라면 Empty page frame을 찾는다.

3-1. Empty page frame이 없다면 하나를 Swap-out 해야 한다.

3-2. Empty page frame이 있다면 다음 단계로 넘어간다.

4. Disk→Page frame 의 I/O 작업이 필요하다. P
    - 이 과정에서 I/O로 인해 process는 waiting 해야 한다.
    - process 입장에서는 억울한 부분이 있을 수 있다.
5. I/O가 종료되면, 해당 Page를 Page table에 등록한다.
    - valid-invalid bit을 v로 설정
    - Page table에 실제 Page frame의 시작 주소를 적는다.
6. Waiting → Ready로 바꾸고, Ready queue에 넣는다.
7. 언젠가 Scheduler에 의해 Running 상태가 되면 Page fault를 발생한 명령어부터 재시작한다.

# Page Fault

Invalid page에 접근하는 것은 **HW Trap (MMU)를** 발생한다.

- Page fault trap이 발생한다.

OS가 **Trap handler를 통해 Page fault handler를 호출**한다.

Page fault를 처리하는 과정은 다음과 같다.

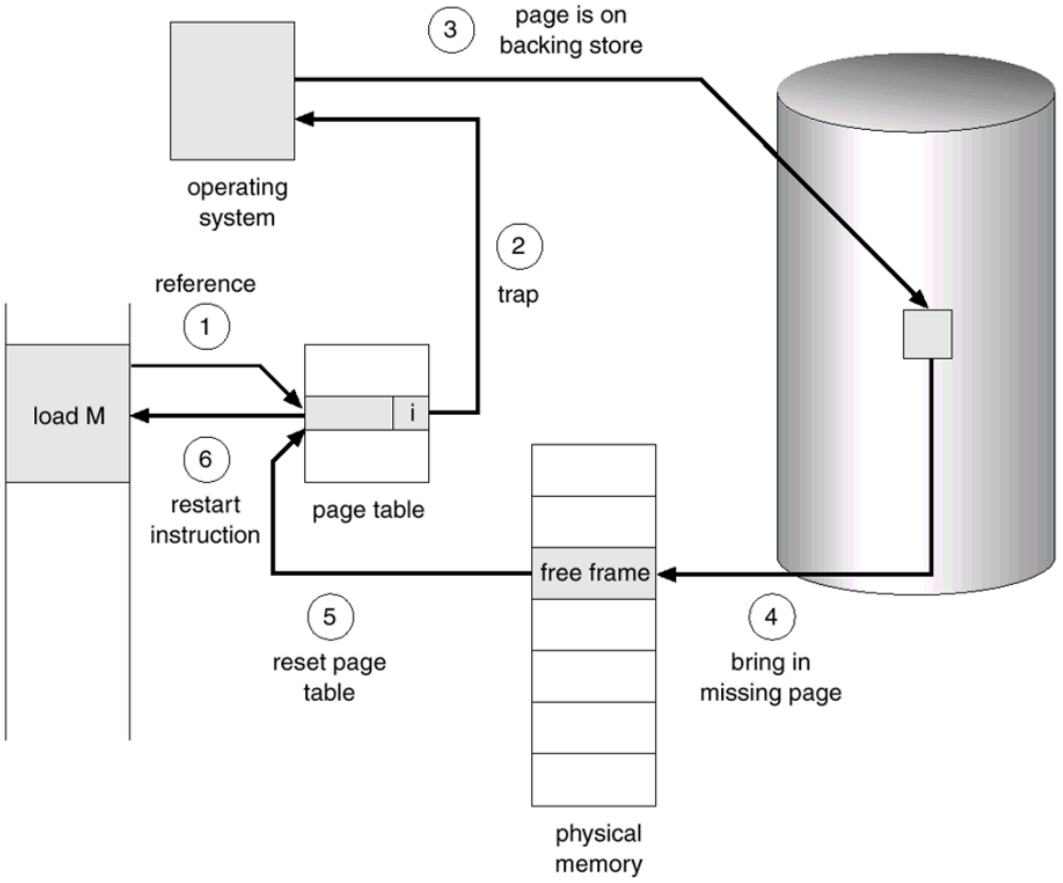
1. 먼저 **잘못된 접근을 하고 있는 지부터 판단**한다.
    - 잘못된 접근인 경우에 abort

2. 잘못된 접근이 아니라면 **Empty page frame을 찾는다.**

3-1. Empty page frame이 없다면 하나를 **Swap-out(Replacement)** 해야 한다.

3-2. Empty page frame이 있다면 다음 단계로 넘어간다.

4. **Disk→Page frame 의 I/O 작업**이 필요하다.
    - 이 과정에서 I/O로 인해 process는 waiting 해야 한다.
    - process 입장에서는 억울한 부분이 있을 수 있다.
5. I/O가 종료되면, **해당 Page를 Page table에 등록**한다.
    - valid-invalid bit을 v로 설정
    - Page table에 실제 Page frame의 시작 주소를 적는다.
6. **Waiting → Ready로 바꾸고, Ready queue**에 넣는다.
7. 언젠가 Scheduler에 의해 Running 상태가 되면 **Page fault를 발생한 명령어부터 재시작**한다.



1. Load 명령어가 reference를 한다.
2. Page table에서 Invalid인 것을 확인한다.
3. Trap (Page fault) 발생
4. OS가 Page가 Backing store에 존재하는 것을 확인하고 Memory로 올림
5. Page table을 세팅
6. Load 명령어 재실행

# VM을 위한 Hardware design의 어려움

Page fault가 명령어 실행 중간에 발생할 수도 있기 때문에 어렵다.

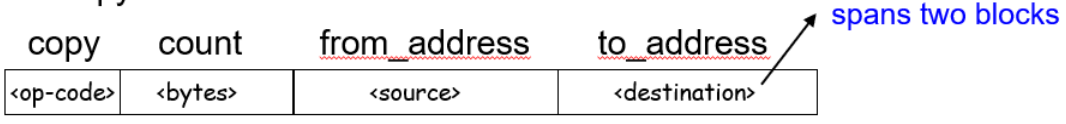
1. Instruction Fetch 과정에서 발생
    - Memory에 올린 후 다시 fetch

2. Operand (피연산자) fetch 과정에서 발생

- 명령어 fetch
- decode
- operand fetch
- 위 세 가지 과정을 반복해야한다.

3. Worst case : Block copy 명령어

Ex: Block copy instruction:




- From addr → To_addr로 count byte만큼 copy하는 명령어
- 위 명령어를 수행하는 과정에서 두 개의 block을 copy 해야 한다고 가정하자.

- 만약 첫 번째 Block의 copy는 정상적으로 수행한 이후에, 두 번째 block을 copy 하는 과정에서 Page fault가 발생한다면??
- 그냥 Inst Fetch 부터 재실행하면 되지만, 이 과정까지 **첫 번째 Block에 대한 데이터 무결성에 문제가 발생한다.**
- 이런 상황이 발생하면, **명령어 시행 이전 상태로 Undo** 해야한다.
    - 이를 위해선 Temporary addr와 value를 저장할 **추가적인 하드웨어**가 필요하다.

## Demand Paging 성능 평가

지금까진 배운 방법으로는 **Inst를 실행시마다 Memory에 page가 없어서 page fault + I/O가 발생할 가능성**이 있다.

- 이는 심각한 문제로, 성능에 대한 평가가 필요하다.

Page Fault Rate $0 \le p \le 1.0$

- 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 if needed] + swap page in + restart overhead)

- `EAT`에서 swap-in / out 모두 I/O 작업이다.
- I/O overhead는 굉장히 크기 때문에 조심해야 한다.
- restart overhead는 inst fetch부터  다시 실행하는데 걸리는 Overhead이다.

예시를 확인해보자

- Memory access time = 200 nanoseconds (*ns*)
- Average page-fault service time = 8 miliseconds (*ms*)
- EAT = (1 – p) x 200 + p x 8 ms = (1 – p) x 200 + p x 8,000,000 = 200 + p x 7,999,800 *ns*
- If one access out of 1,000 causes a page fault, EAT = 8.2 *μs*
- This is a slowdown by a factor of 40!!

- 이 경우, p가 1/1000 이더라도, EAT는 8.2 us로 **단순히 memory access 하는 경우보다 40배** 느려졌다.
- 그렇다면 Virtual memory를 사용해도 괜찮을까?

지금까지 배운 방식은 Pure demand paging 이다.

- 즉, cpu가 요청할 때까지 절대 memory에 등록되지 않는다.
- 이 경우, 새 program이 시작되면 계속 page fault가 발생하게 된다.

다른 방법들로 보완할 수 있다.

1. prepaging
   - 시작 전에 뻔해 보이는 것은 미리 올려놓자!
2. Read ahead
   - 한 page를 읽을 때, 뒤에 연속된 다른 page도 그냥 같이 올려버리자

**Pure demand paging 이 비효율적이므로 우리는 Locality를 이용하자**

**Locality of reference**

- 어떤 workload던지, 짧은 시간 내에 굉장히 많이 참조되는 Page가 존재한다.
- **Loop 같이, Locality가 높은 page를 memory에 올라간 후 Swap - out 되지 않게 지속적으로 잘 관리하면 한 번의 Page fault로 많은 것을 커버할 수 있다.**
- Swap-out 되지 않도록 유지하기 위해서 **System이 Locality를 알고, 사용할 수 있도록 해야 한다.**

# Page replacement algorithm

- Over allocation을 방지한다.
- `Dirty (modify) bit` 를 활용해 Dirty bit = 1인 경우에만 Disk에 write 한다.
  - 해당 Page가 memory에 올라온 이후, 한 번이라도 수정되었다면 Dirty bit = 1
  - **Dirty bit가 1인 경우에만 SWAP OUT**
  - 0이라면 해당 page를 그냥 덮어쓴다.
- Logical memory와 physical memory의 완전한 분리가 가능하도록 만들어 준다.

**Goal: 어차피 올릴 대상은 정해져 있다! 우리는 Victim (내쫓길 대상)만 찾으면 된다.**

- **Page fault를 최소화하고 program behavior의 locality를 최대한 보장**하는 방식으로 동작해야한다.
- `정성적 목표`: Locality를 최대한 보장
- `정량적 목표` : 미래에 발생한 page fault를 최소화
  - 이를 위해선 특정 page가 곧 사용될 만한 page인지를 알아야 한다.

# Basic page replacement

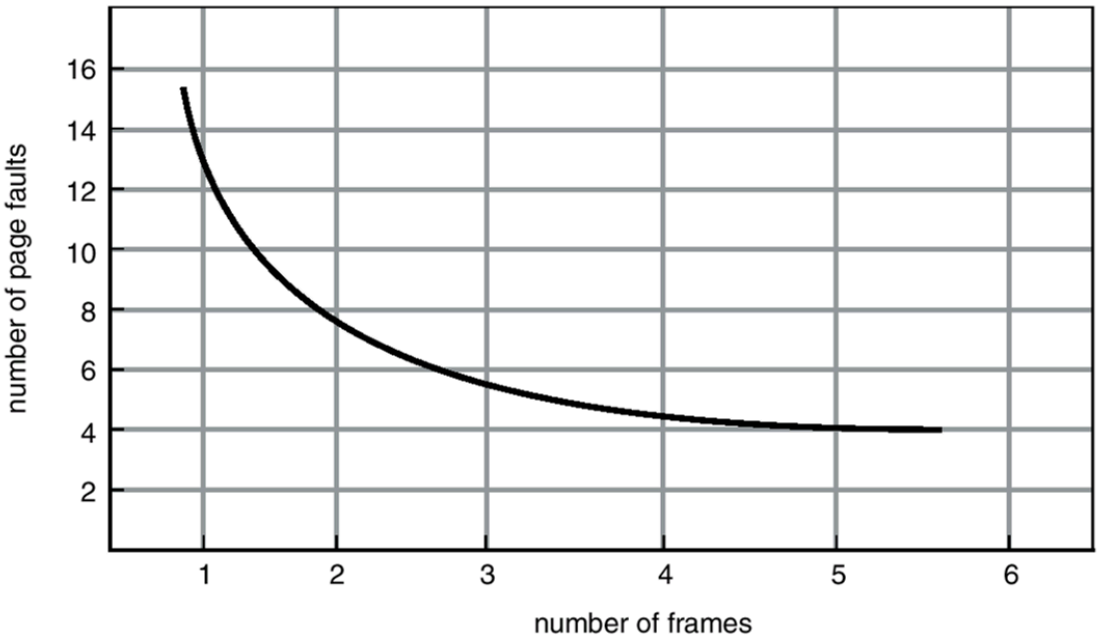
1. Disk에서 DRAM에 올리고자 하는 page를 찾는다.
2. DRAM에서 Free frame을 찾는다.
   - 만약, free frame이 없다면 Replacement algorithm을 사용해야 한다.
3. 원하는 페이지를 메모리에 올리고, page table update
4. Process 재시작

### Page-replacement algorithm

- Lowest page fault rate을 목표
- `reference string` : 알고리즘을 평가하기 위해 시뮬레이션 진행할 때, 시뮬레이션을 위해 page 접근 순서를 사전에 정해 놓은 것

- **Reference string에 따라 memory에 접근할 때, Page fault가 최대한 적게 발생** 해야 한다.

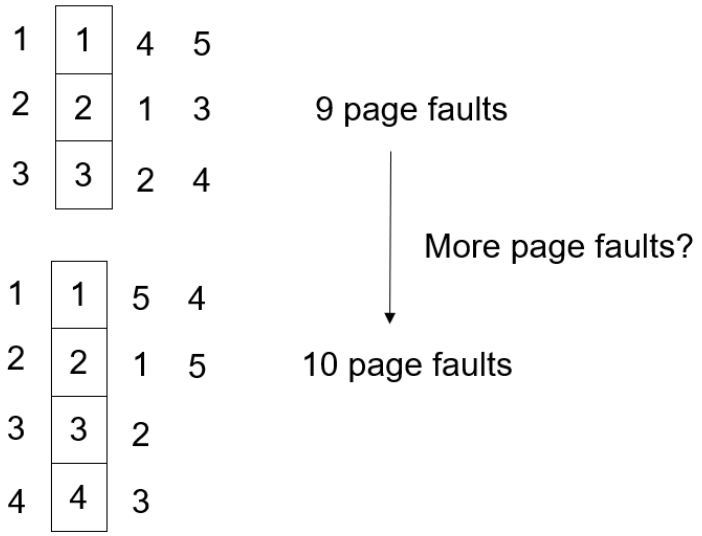
알고리즘이 정상적으로 동작하는 경우라면, 아래처럼 Memory 공간이 커질 때, Page fault 발생 횟수도 줄어야 한다.




# Algorithms

1. FIFO : Scheduling 문제에서는 필수적인 알고리즘이다.

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **3** frames (3 pages can be in memory at a time per process)




- **4** frames

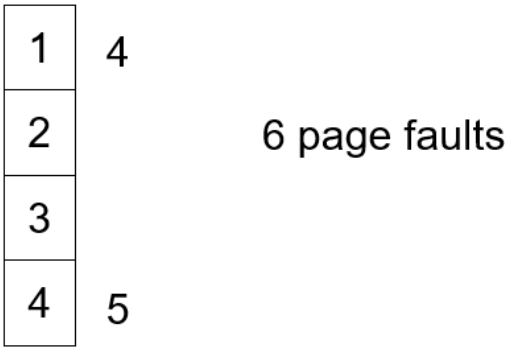
- Belady라는 사람이 Page frame의 개수가 증가했는데, Page fault의 개수도 증가하는 이상 현상을 발견했다.
- 이 때문에 **FIFO는 Replacement algorithm으로 사용하지 못한다**.

FIFO는 이외에도, 직전에 쫓겨 난 Page가 바로 다음에 요청되면 Page fault가 계속 발생한다.

2. `Optimal` : Replacement algorithm의 상한선을 제시
   - **미래에 대한 모든 정보를 안다고 가정하고, 최적의 알고리즘**을 만들어보자
   - 다른 알고리즘은 Optimal과 최대한 유사하게 동작해야 한다.
   - 즉, **Optimal algorithm은 다른 알고리즘의 동작을 측정하는 용도**이다.
   - **제일 나중에 접근 또는 다신 접근되지 않을 것 같은 Page를 먼저 Replace**

4 frames example

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



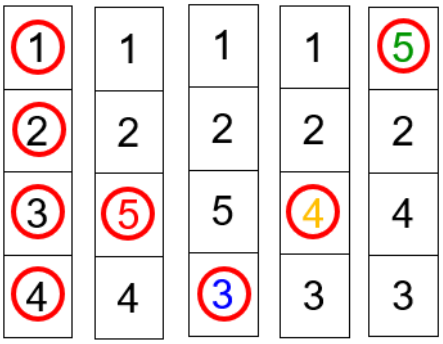

- FIFO에 비해 성능이 40% 증가하였다.

Optimal algorithm이 성능이 좋음은 확인했다.

**그렇다면 우리는 어떻게 미래에 대한 정보를 알 수 있을까?**

3. Least Recently Used (LRU)
   - Optimal을 모방한 알고리즘
   - **과거 정보와 Locality를 이용하여, 미래를 예측**
   - **최근에 사용되지 않은 것은 미래에도 사용되지 않을 것이라는 아이디어**

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 이 경우에 Optimal 만큼 좋진 않지만, 꽤나 괜찮은 성능을 보이는 것을 확인할 수 있다.

문제) LRU를 어떻게 구현할까?

- 각 Page마다 page table에 `Timestamp` 라는 변수를 관리해야 한다.
- Replace가 필요한 경우마다, 전체 Timestamp를 확인하며 최솟값을 찾아야 한다.
- Page 접근 시마다 전체 Timestamp를 Update도 해야 한다.
  - 이는 **Timer device I/O를 유발**한다.
  - 마지막으로 접근된 시간을 관리한다고 생각
- **따라서 공간 (추가적인 변수) / 시간 (Update 및 Min 찾기) 적인 측면에서 Overhead가 너무 크다.**

우리는 LRU를 그대로 구현하지 않고, **LRU와 비슷하게 동작하고, Overhead는 줄이는 방식**의 **LRU Approximation algorithm**을 사용할 것이다.

LRU Approximation에 대해 살펴보기 전에, LRU Implementation algorithm을 살펴보자

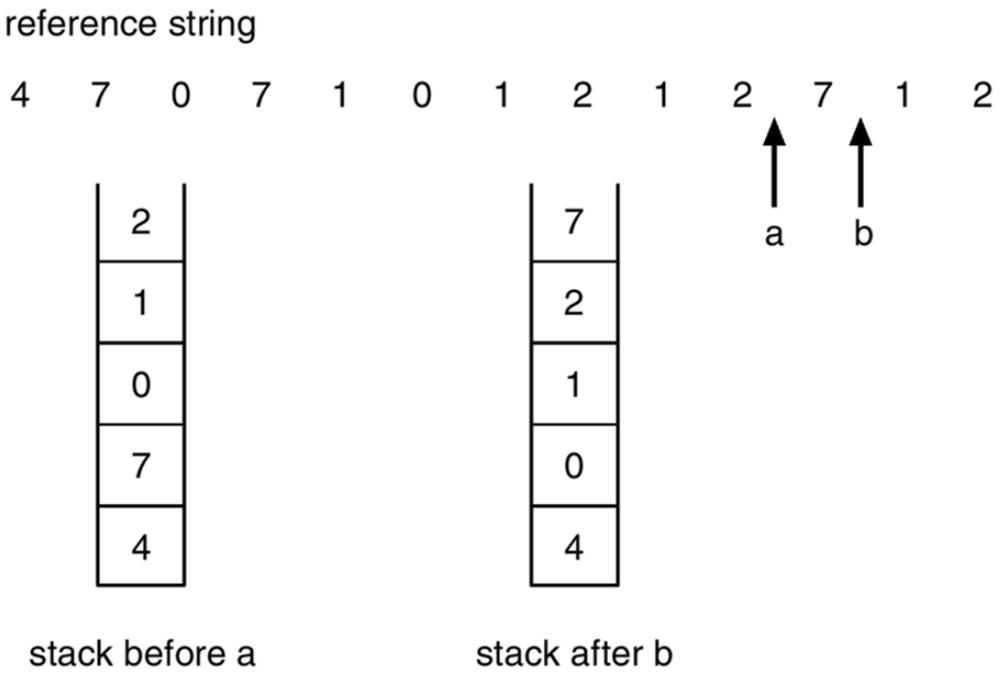
# LRU Implementation Algorithm

- **LRU와 완벽히 유사하게 동작하지만, 구현 방식을 달리하여 Overhead를 줄인 방식**

1. `Counter implementation`
   - 각 page는 counter를 갖는다.
   - `CPU Counter` **라는 큰 하나의 정수형 변수가 존재하고, 이 변수는 모든 memory 접근 시마다 증가한다.**
   - 특정 Page에 접근하는 경우 해당 page의 counter에  CPU counter 값을 복사한다.
   - Replacement의 경우, minimum counter 값을 찾기 위해 page table을 확인한다.

- 각 Page에 Timer device가 I/O 하는 대신에, CPU counter만 1씩 늘려주면 된다는 점에서 Overhead는 비교적 작다.
- 하지만 **Search, Space 등의 Overhead는 여전히 남아 있다**.

2. Stack implementation
   - 각 page가 **doubly linked list로 관리**되도록 구현한다.
   - **Top에 가장 최근에 접근한 page가 오도록 한다.**
     - 이때, pointer update를 해야 한다.




- 여기서 Replacement 대상은 가장 아래 위치한 page이다.
- Search에 O(1)이 소요된다는 큰 장점이 있다.
- 그러나, Top으로 옮기는 과정에서 top과 pointer를 바꾸는 과정이 필요하다.
  - Pointer 변경도 Memory 접근이다.
  - Overhead 증가
- 위 Doubly linked list를 관리하는 것도 Overhead다

**지금 확인한 것처럼 결국 (LRU와 완벽하게 동일하게 동작하는 것)과 Overhead는 어쩔 수 없는 Trade-off 이다.**

→ 결국 Overhead를 줄이기 위해서는 LRU의 성능을 어느 정도 포기해야 한다.

→ LRU Approximation algorithm

# LRU Approximatin algorithm

1. Reference bit
    - 각 page가 갖는 bit이고 0으로 초기화
    - 각 Page가 referenced 되면, 1로 세팅
    - 주기적으로 0으로 초기화 한다.
    - **Reference bit = 0인 어떤 Page를 랜덤하게 선택하여 victim으로 선택한다.**

위 방법은 너무 간추렸다.

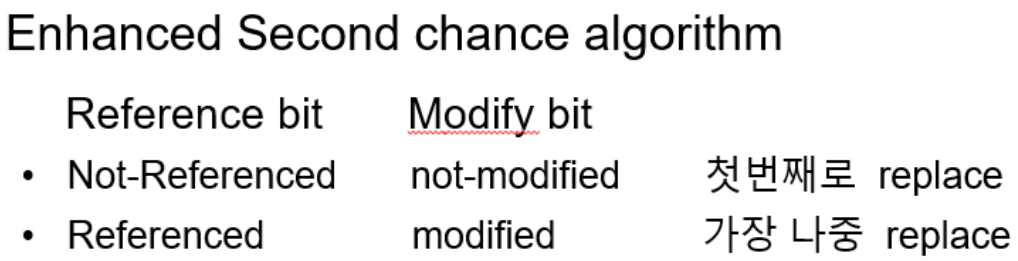
Victim을 선정할 때, Reference bit = 0인 page 중 어느 것이 최근에 사용되었는지 알 수가 없다.

따라서 성능이 좋지 않다.

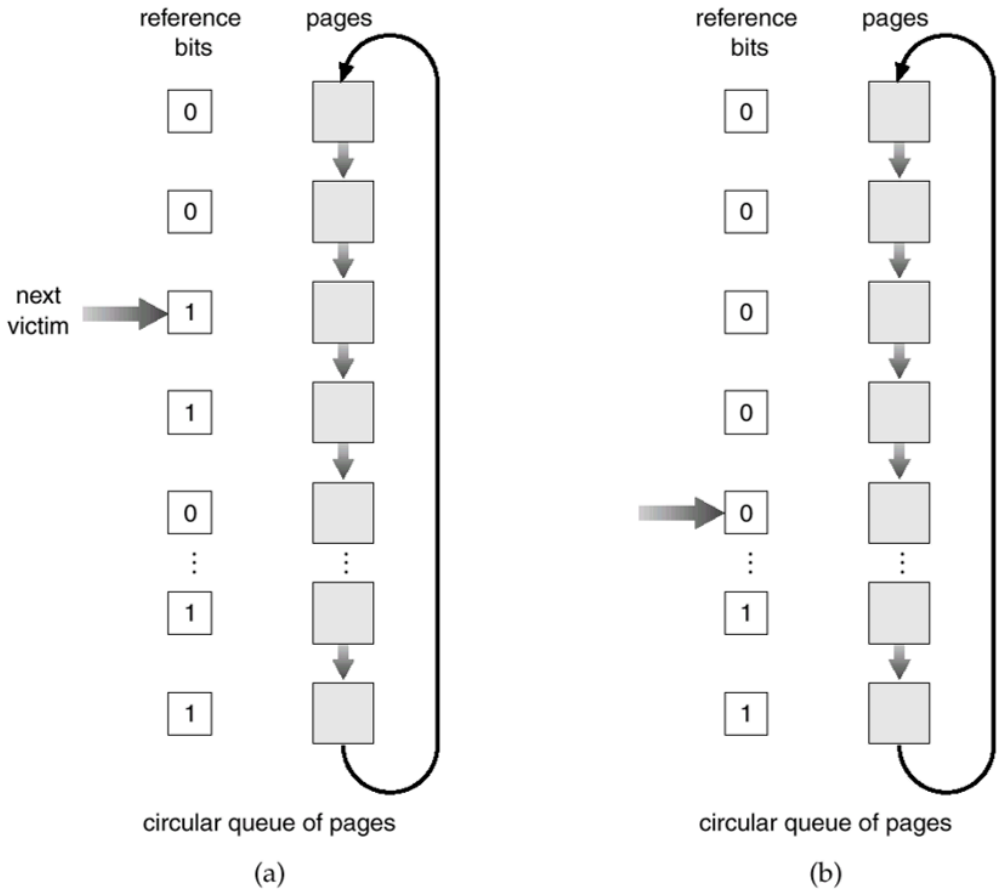
2. Additional - Reference bits
    - 기존 Reference bit에 추가적으로 8 bit reference bit 사용
    - **Additional Reference bit는 주기적으로 오른쪽으로 shift** 한다.
    - 기존 1-bit reference bit는 (1)과 같이 동작한다.
    - 1-bit reference bit이 0이라면 **History (Additional 8 bits)를 보고 어떤 것을 victim**으로 선정할 지 확인한다.
        - MSB 부터 확인해서, 0인 것이 먼저 나오는 것을 쫓아낸다.

그러나 이 방법은 각 page마다 8개의 추가적인 공간이 필요하고, Additional reference bits를 비교하여 최솟값을 구하는 알고리즘을 구현해야 한다.

3. Second chance (clock) algorithm
    - Reference bit를 하나 사용한다.
        - (1)과 동일하게 동작한다.
    - **Page를 Circular queue 형식**으로 만든다.
    - **Advance pointer를 갖는다.**
        - 만약 Pointer가 지금 가리키고 있는 Page의 Reference = 0 이라면 제거한다.
        - 만약 Pointer가 지금 가리키고 있는 Page의 Reference = 1 이라면 Reference count를 0으로 바꾸고 한 바퀴 돈다.
            - 만약 한 바퀴 돌게된 후에도 Reference count가 0이라면 pointer가 한 바퀴 도는 동안 한 번도 참조되지 않았다는 의미이다.
            - 이 경우, 실제로 자주 사용되지 않는 Page일 가능성이 크기에 제거한다.
    - **Circular queue로 구현되어져 있기 때문에, Reference bit와 다르게 page 접근 순서에 대한 보장이 생긴다.**
        - 즉, **현재 보고 있는 Reference bit = 0인 page는 다음 reference bot = 0인 page보다 더 과거에 접근되었을 것이라는 것이 보장된다**
        - **결국 자연스럽게 더 오랫동안 사용되지 않았던 Page를 먼저 Replace하게 된다.**
    - Worst case : 모든 Page의 Reference count = 1이라면 FIFO와 똑같이 동작하게 된다.
    - 아래와 같은 방법으로 성능을 향상시킬 수도 있다.




Example)




장점)

- **Space overhead가 작다**
  - Reference count는 각 page마다 1개씩, pointer는 전체에서 1개만 존재한다.
- **Victim을 찾는 알고리즘이 따로 필요하지는 않다.**

# Counting algorithm

접근 시점도 중요하지만, 접근 빈도도 중요하지 않나?

1. LFU (Least Frequently Used algorithm
   - 최근에 가장 적게 사용된 Page를 버림

2. Most Frequently Used
    - 가장 많이 사용된 Page를 버림
    - **“최근에 가장 많이 사용된 것은 이미 충분히 사용할만큼 사용되고 가장 적게 사용된 것은 최근에 Memory에 올라온 것 아닌가?” 라는 아이디어에서 나왔다.**

**일반적으로는 LRU가 성능이 더 좋다. 그러나 특수한 목적에 따라 다른 알고리즘도 사용 가능하다!**