Background

프로그램의 일부만 메모리에 올라감

- 에러 처리 코드는 나중에 올려도 됨
- array, list, table은 실제 필요한 양보다 더 많은 메모리를 미리 확보
- 특정 기능은 사용 빈도가 낮음
- 모든 기능이 필요하더라도 동시에 필요하진 않음

일부 코드만 메모리에 올라간 프로그램의 장점

- physical memory에 제약 받지 않음
- 더 많은 프로그램을 동시에 운용 가능
- 실제로 필요한 코드만 필요할 때 메모리에 올리므로 I/O가 줄어듦

Virtual Memory Overview

Virtual Memory란?

- physical memory와 logical memory를 분리
- 필요한 부분만 memory에 올리므로 physical address space 보다 훨씬 큰 logical address space
- 다른 프로세스가 address space를 공유할 수 있게 됨
- 더 효율적인 process creation
- 더 많은 프로그램이 동시에 운용
- Less I/O

Virtual address space

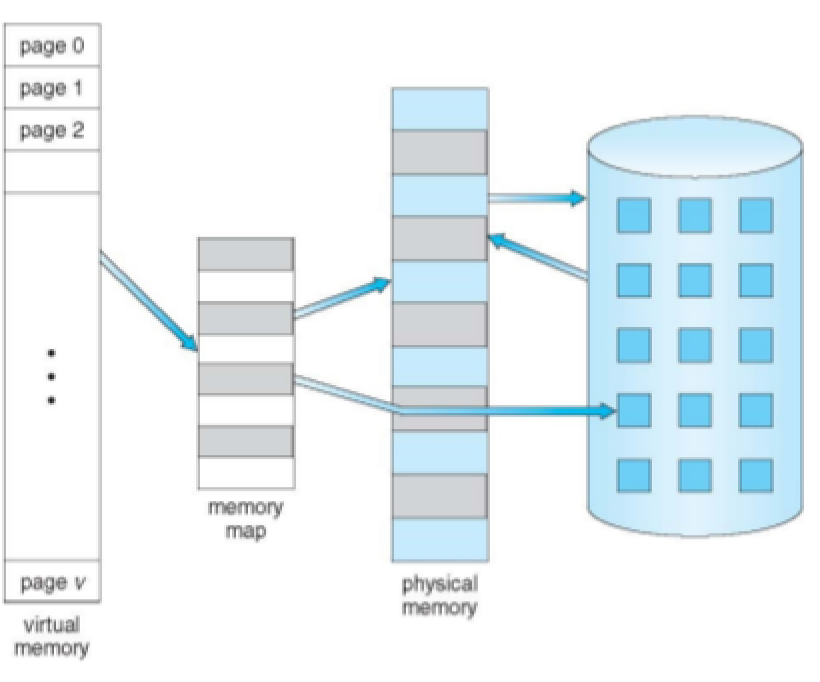
- memory에 process가 적재되는 방법에 대한 logical view
- address 0 에서 시작, space의 끝까지 contiguous
- physical memory는 page frame으로 구성
- MMU는 logical을 physical에 mapping 역할 수행

Virtual memory can be implemented via

- Demand paging
- Demand segmentation

Demand Paging

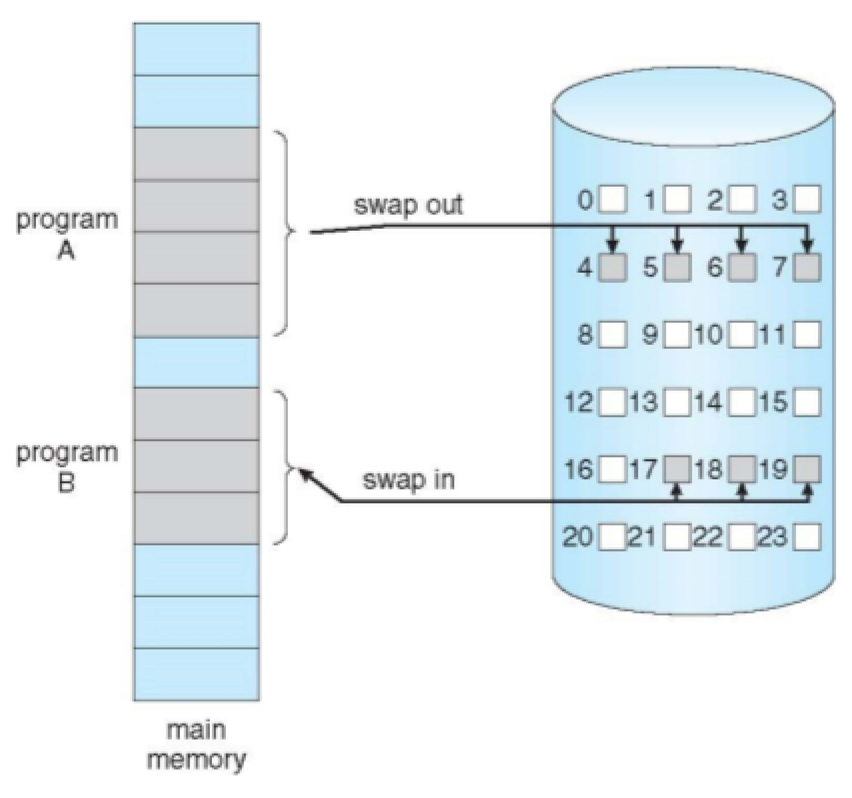
Demand Paging이란?

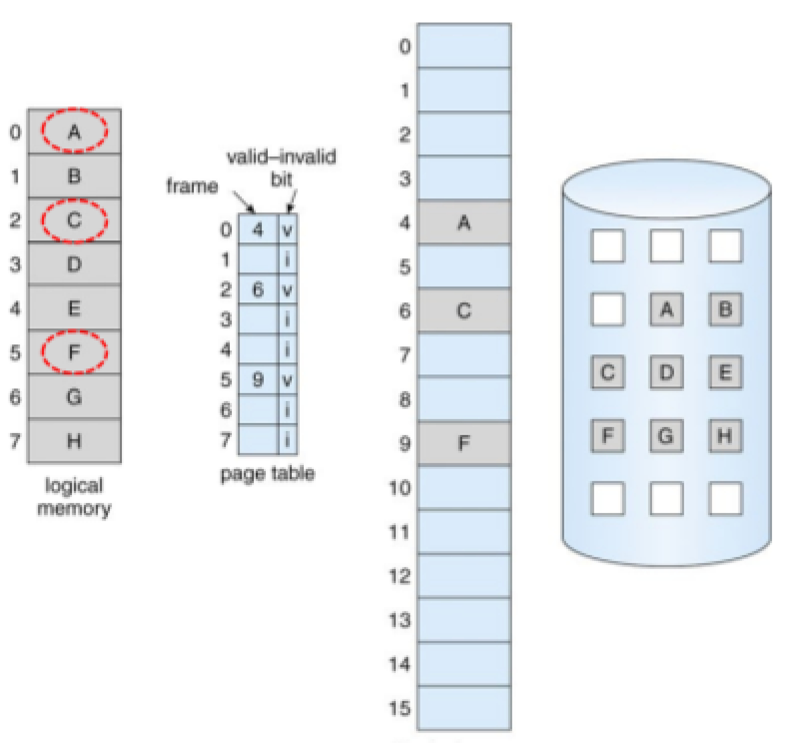


- load time 때 memory에 process를 가져옴
- 단, 그 페이지가 필요할 때만
  - Less I/O, Less memory, Faster response, more users

Demand Paging의 Basic Concepts

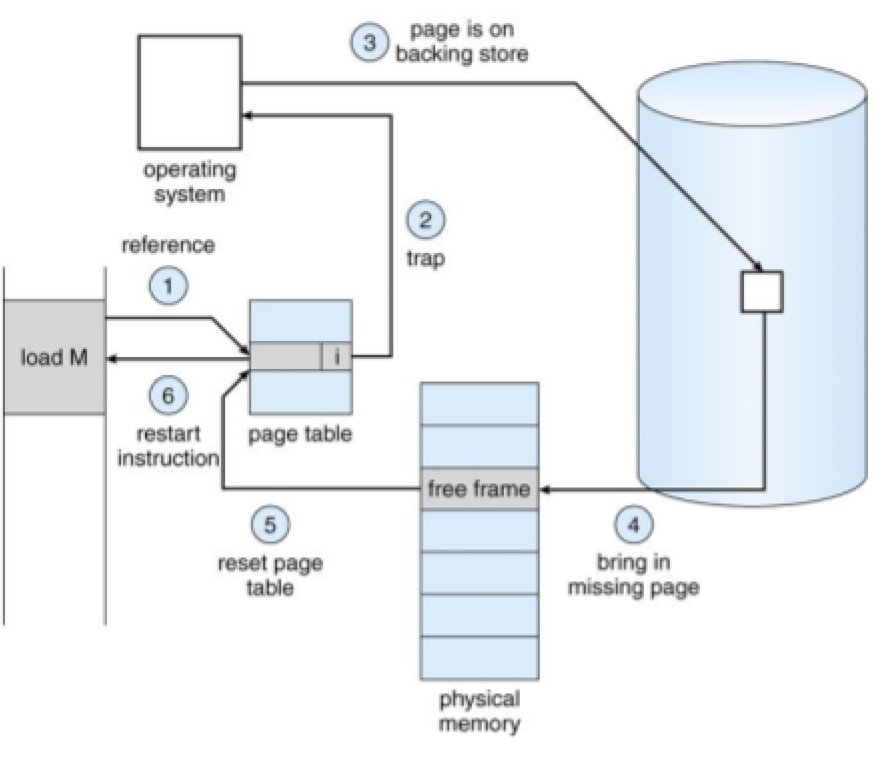
- swapper가 whole process를 가져오기 보단 those pages that will be used만 가져옴
  - (Lazy swapper, pager) : page가 필요해지기 전 까진 절대 swap하지 않음
- page가 disk에 있는지 memory에 있는지 구별하기 위해선 hardware support가 필요
  - valid-invalid bit scheme
  - valid : page가 legal하고 메모리에 있는 경우
  - invalid : page가 invalid하거나 valid하더라도 disk에 있는 경우
  - invalid reference : abort
  - not in memory : 메모리로 가져오기





page fault란?

- page에 대한 참조가 있을 경우 first reference였다면 그 page는 아직 memory에 없으므로 page fault 발생
  - OS는 그 페이지가 디스크에 있는지, invalid reference(=> abort)인지 구별하기 위해 다른 table을 찾아봄
- empty frame을 찾아옴
- schedule된 disk operation으로 frame에 page swap
- 현재 memory에 있는 page를 가르키기 위해 table reset(갱신, 다시 설정)
  - valid bit를 v로 설정
- page fault를 일으킨 instruction 다시 수행



Demand Paging의 성능

demand paging의 3가지 역할

- 인터럽트 처리 : page fault시 인터럽트를 발생시킴
- disk에서 page 읽기
- process 재시작 : page fault가 일어난 명령어 부터 다시 시작

page fault rate

- 0 ~ 1까지
- 0이면 no page fault
- 1이면 every referece is a fault

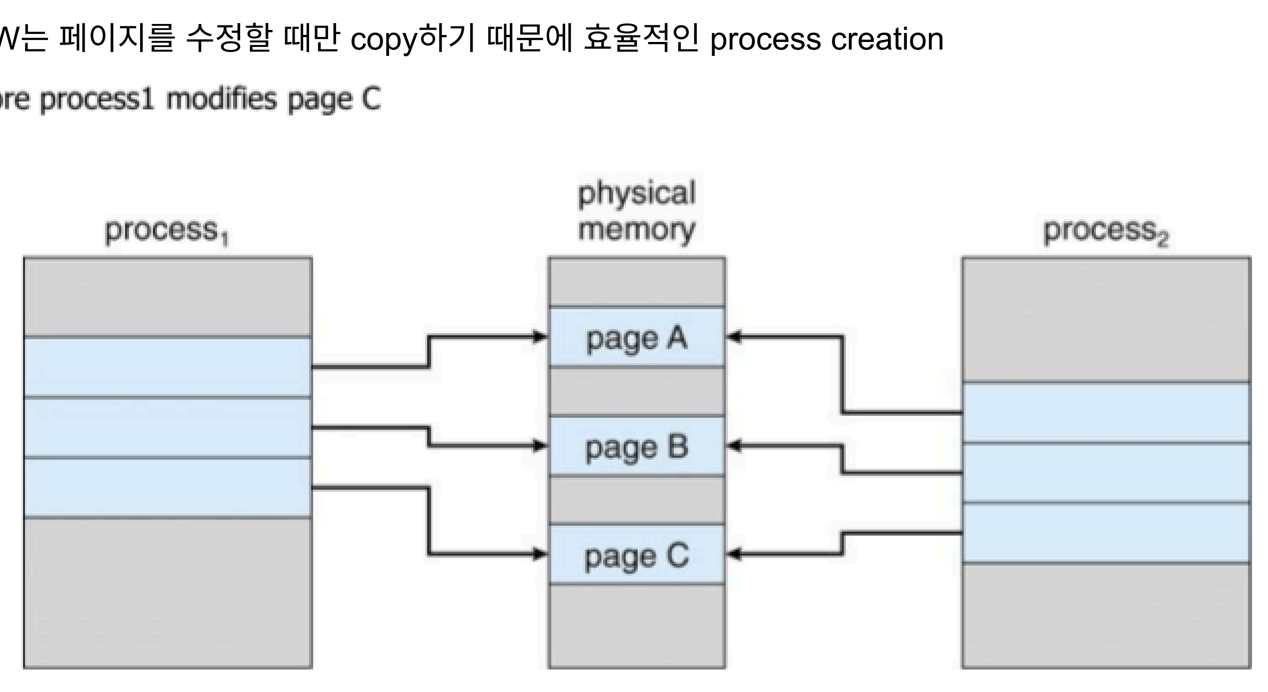
Effective Access Time (EAT)

- EAT
- = (1 - p) x Memoroy Access Time + p x page fault service time
- = (1 - p) x Memoroy Access Time + p x page fault overhead
- = (1 - p) x Memoroy Access Time + p x (swap page out + swap page in + restart overhead)
- swap page out (메모리 -> 디스크)
- swap page in (디스크 -> 메모리)

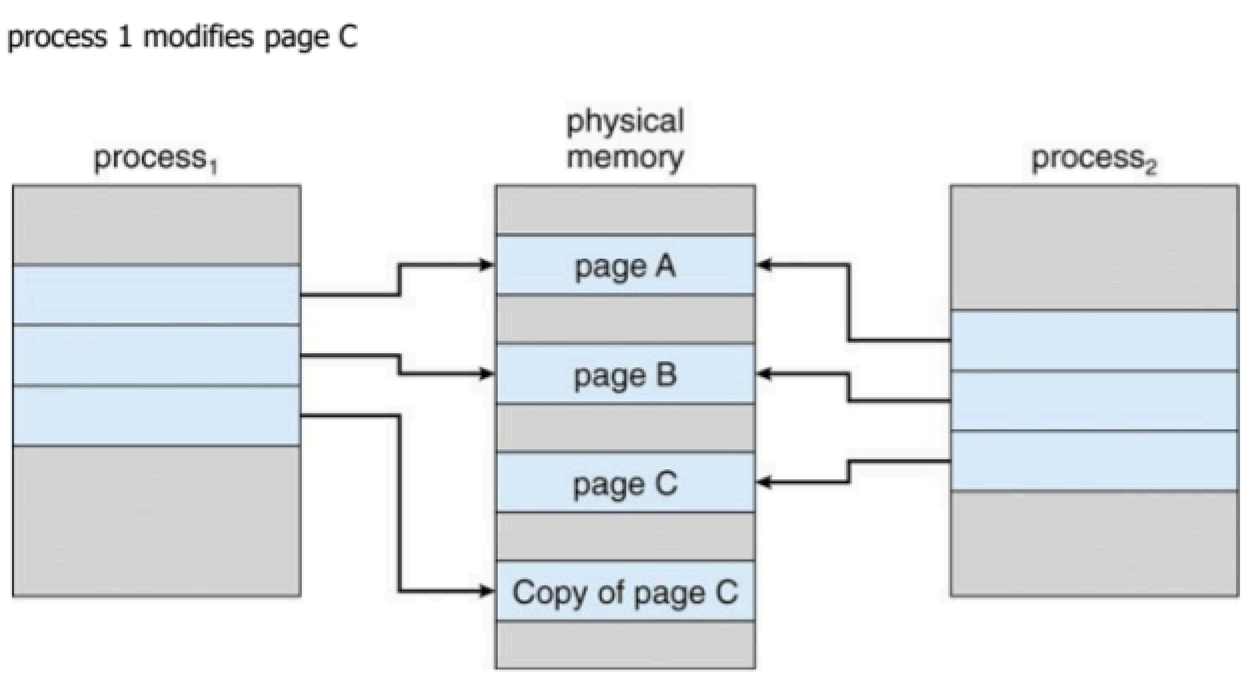
virtual memory에서 process creation (COW)

- parent와 child process는 처음엔 메모리 상 같은 페이지를 공유
- copy-on-write (COW)
  - 만약 공유되는 페이지를 parent나 child가 수정하려 한다면 그 페이지를 복사한 다음 복사한 페이지에 수정
  - COW는 페이지를 수정할 때만 copy하기 때문에 효율적인 process creation

Before process1 modifies page C



After process 1 modifies page C



- page C만 수정했기 때문에 C만 복사 후 p1만 갖게 됨

Page Replacement

If there is no free frame

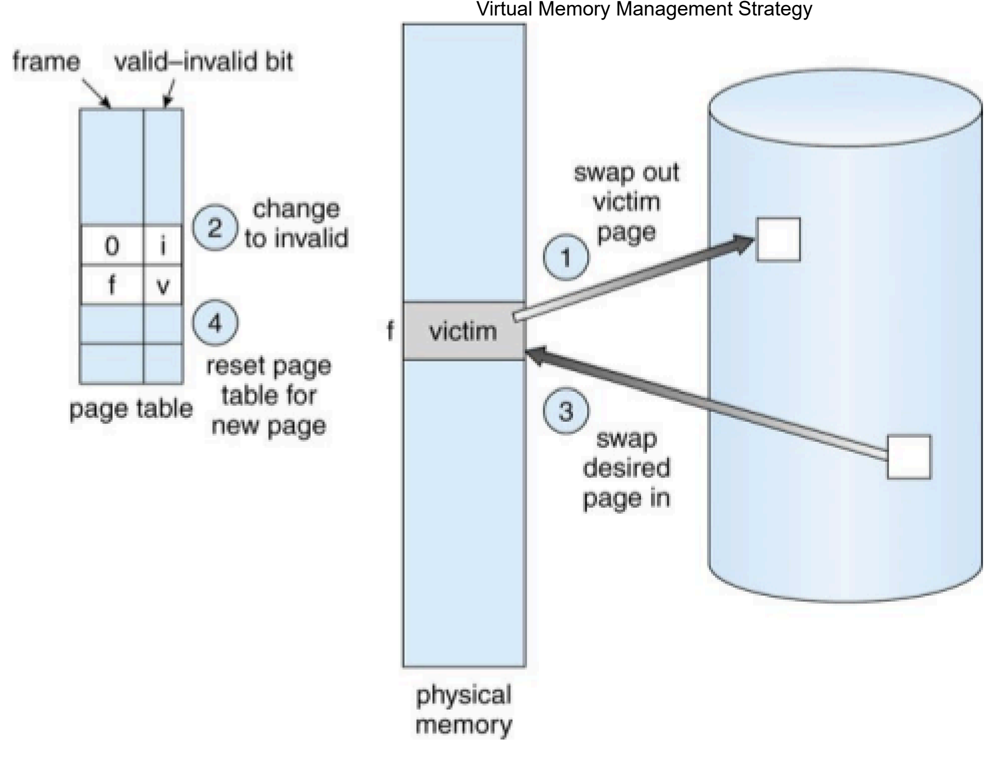
- page replacement : 메모리 속 사용하지 않는 페이지와 swap, 기존 페이지는 disk로 swap out
- 최소한의 page faults를 발생시키는 page replacement algorithm이 필요
- modify bit (dirty bit)을 사용해서 page transfer의 overhead를 reduce
  - 기존 페이지를 disk로 swap out 할 때 사실 기존 페이지도 disk에서 가져온 것
  - modify bit (dirty bit)으로 메모리에서 한 번이라도 수정된 페이지만 swap out 함
  - why? 한 번이라도 수정되지 않은 페이지는 어차피 disk의 데이터와 같을 것이므로 가만히 냅두면 됨
- page replacement 덕분에 logical memory와 physical memory를 분리시키고 Large virtual memory, smaller physical memory를 구현할 수 있음

Basic page replacement

1. disk에서 desired page의 location 찾기
2. free frame 찾기 (있으면 쓰고, 없으면 page replacement algorithm)
3. desired page를 the newly free frame으로 가져오고 해당 page와 frame table을 update

4. trap을 발생시킨 instruction을 다시 시작

Virtual Memory Management Strategy



frame 수와 page fault의 관계

Frame-allocation algorithm

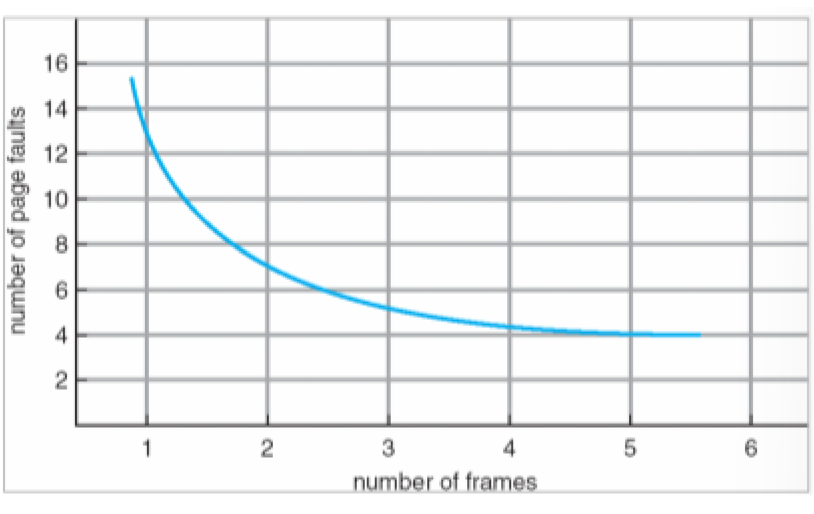
- 각 프로세스에 frame을 얼마나 할당할지 결정
- 어떤 frame을 replace할지 결정

page-replacement algorithm

- first access든 re-access든 가장 적은 page-fault 수를 원함

algorithm evaluation

- page number가 담긴 reference string을 순차적으로 처리하며 page fault 수 count
- page fault는 frame의 수에 영향을 받는다는 결론이 나옴

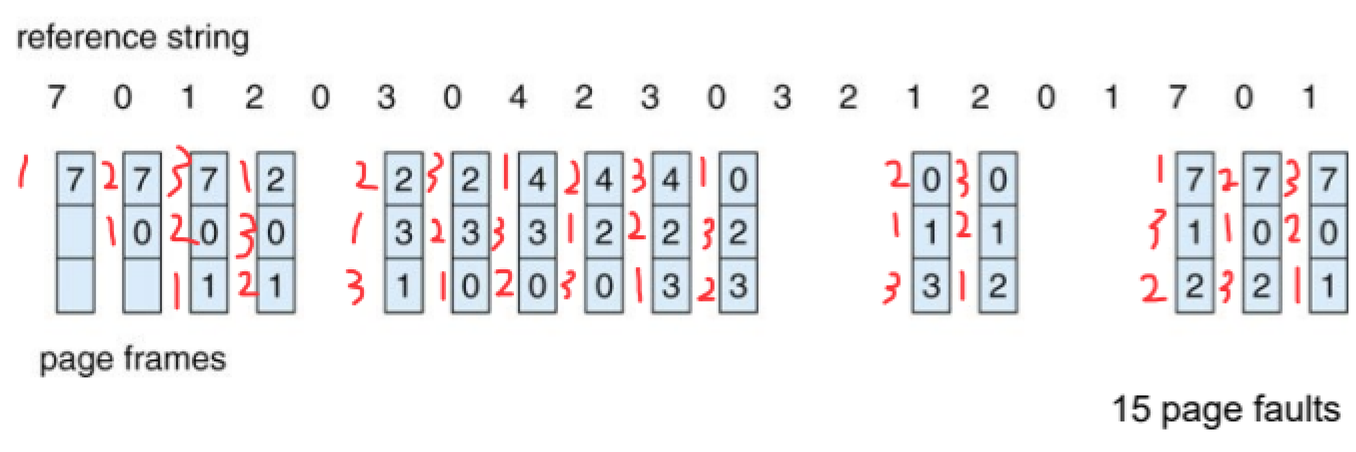


page replacement algorithm

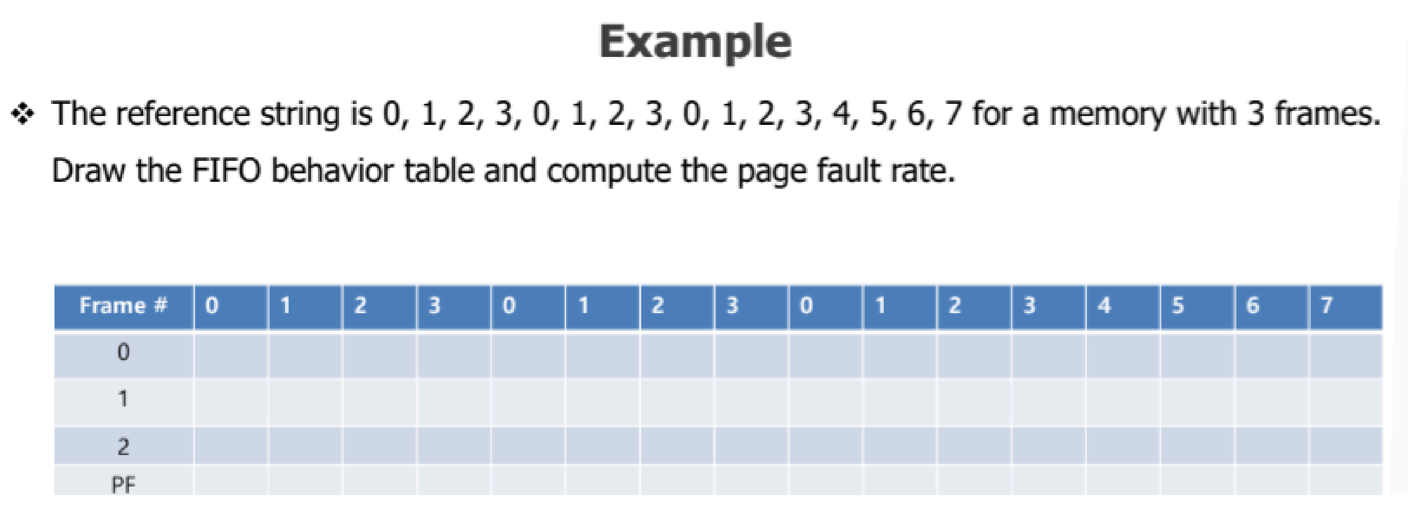
- 예제는 시험 하루 전에 직접 해보는 게 좋을 듯 (시간이 있다면)

FIFO algorithm

- 가장 오래된 page가 선택됨
- 큐에 3개의 frame만 들어갈 수 있다고 가정하자



- 예제

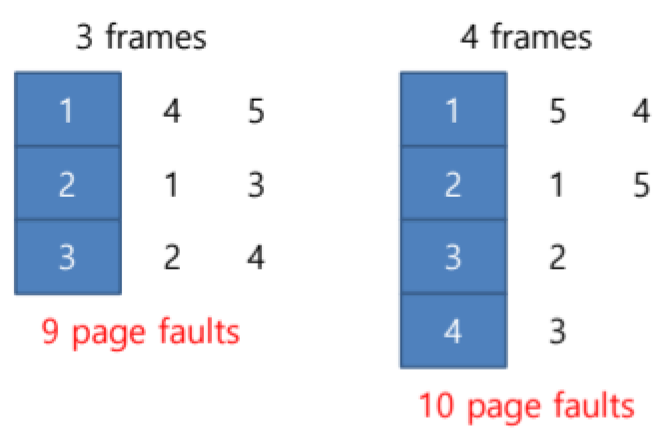


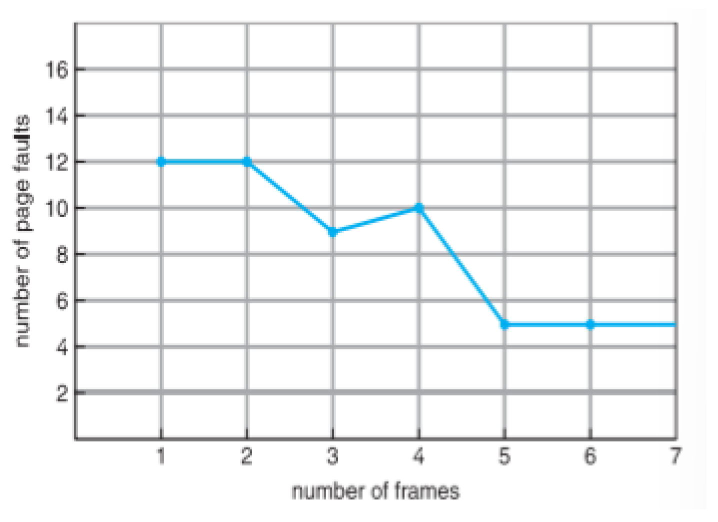
Example

❖ The reference string is 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7 for a memory with 3 frames. Draw the FIFO behavior table and compute the page fault rate.

Frame #	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0																
1																
2																
PF																

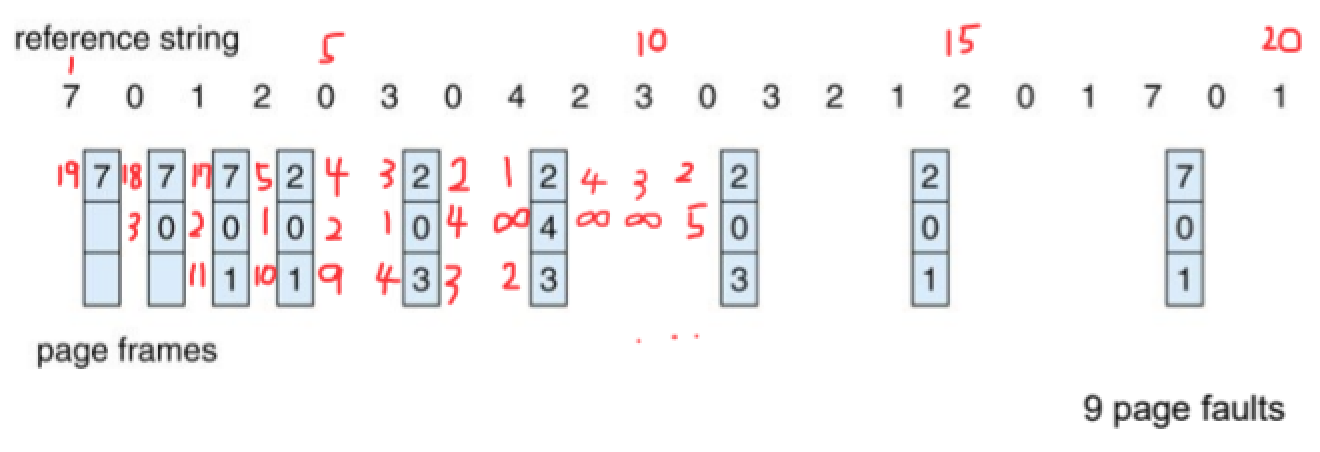
- Belady's abnomaly : frame 수를 늘렸는데 오히려 page fault가 더 많아지는 이상 현상

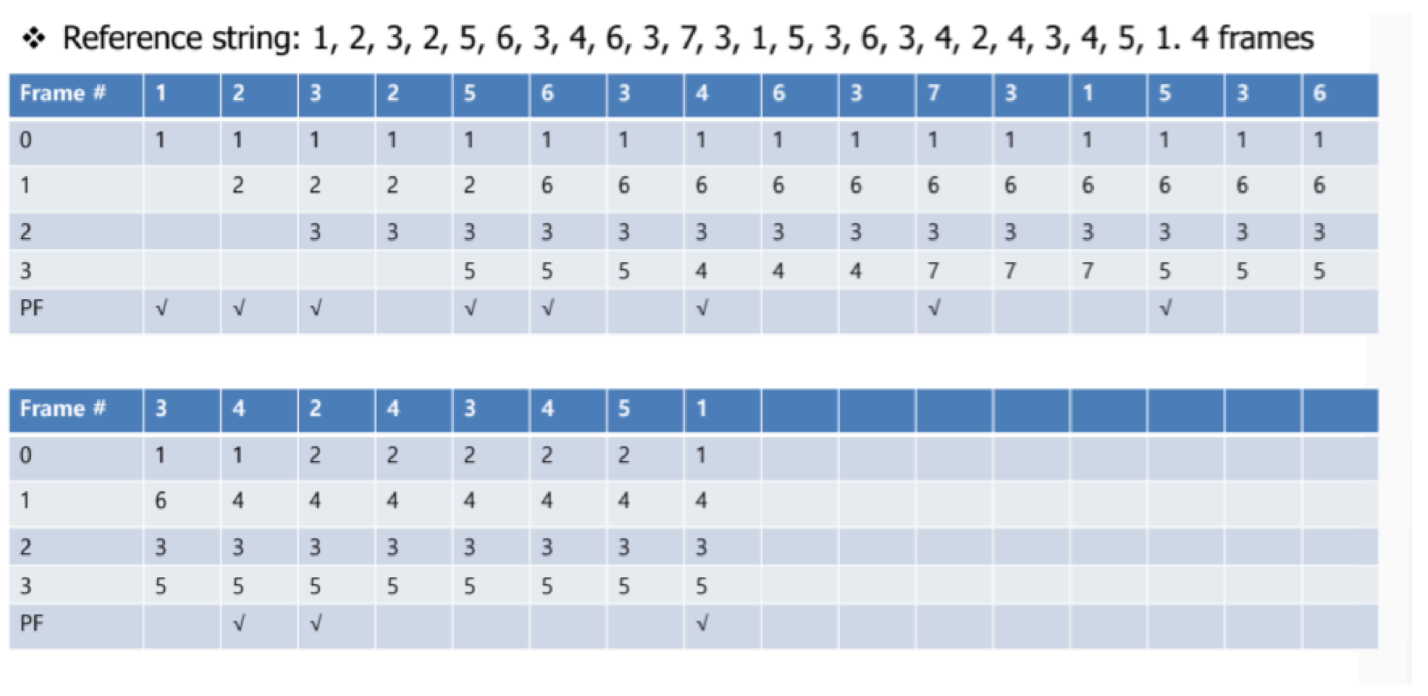




Optimal algorithm

- 가장 오랫동안 안 쓸 page를 replace
- 가장 이상적인 방식
- 가장 적은 수의 page fault 발생
- 어떤 page가 얼마나 오래 안 쓸지 어떻게 알 수 있는가?
- 실제 OS에서 쓰이진 않지만 다른 algorithm의 performance를 평가할 때 비교 기준으로 사용





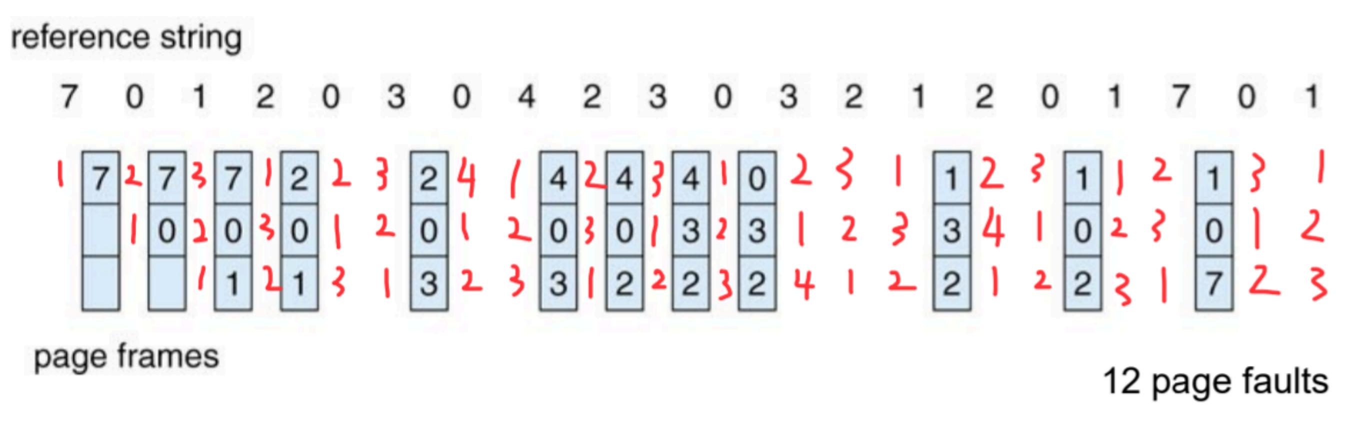
❖ Reference string: 1, 2, 3, 2, 5, 6, 3, 4, 6, 3, 7, 3, 1, 5, 3, 6, 3, 4, 2, 4, 3, 4, 5, 1. 4 frames

Frame #	1	2	3	2	5	6	3	4	6	3	7	3	1	5	3	6
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1		2	2	2	2	6	6	6	6	6	6	6	6	6	6	6
2			3	3	3	3	3	3	3	3	3	3	3	3	3	3
3					5	5	5	4	4	4	7	7	7	5	5	5
PF	√	√	√		√	√		√			√			√		

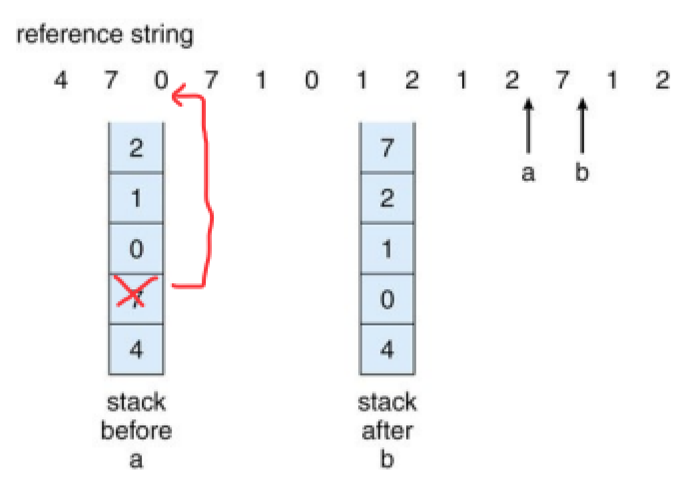
Frame #	3	4	2	4	3	4	5	1								
0	1	1	2	2	2	2	2	1								
1	6	4	4	4	4	4	4	4								
2	3	3	3	3	3	3	3	3								
3	5	5	5	5	5	5	5	5								
PF		√	√					√								

LRU (Least Recently Used) algorithm

- 가장 오랫동안 access 되지 않은 page를 replace
  - FIFO는 가장 먼저 할당된 page를 swap out
- locality가 있는 상황에서는 LRU가 optimal에 가깝게 작동함
- stack algorithm이라 Belady's anamaly가 발생하지 않음
- 각 page들이 언제 access 됐는지 track 해야하므로 구현하기 어려움
- page access의 frequency를 고려하지 않음
  - 자주 쓰이는 page라도 가장 오래 전에 access 됐으면 가차없이 victim
- 모든 종류의 workload(작업 패턴)에 잘 맞지는 않음



- counter와 stack으로 구현
  - counter
    - 모든 page table entry가 counter를 가짐
    - page가 이 entry를 통해 reference 될 때 마다 clock이 counter에 복사됨
  - stack
    - 여기서 말하는 stack은 자료구조의 stack과 달리 top이 아닌 원소도 제거 가능
    - page가 reference 될 때 마다 stack에서 제거 후 top에 put
  - hw support가 필요함

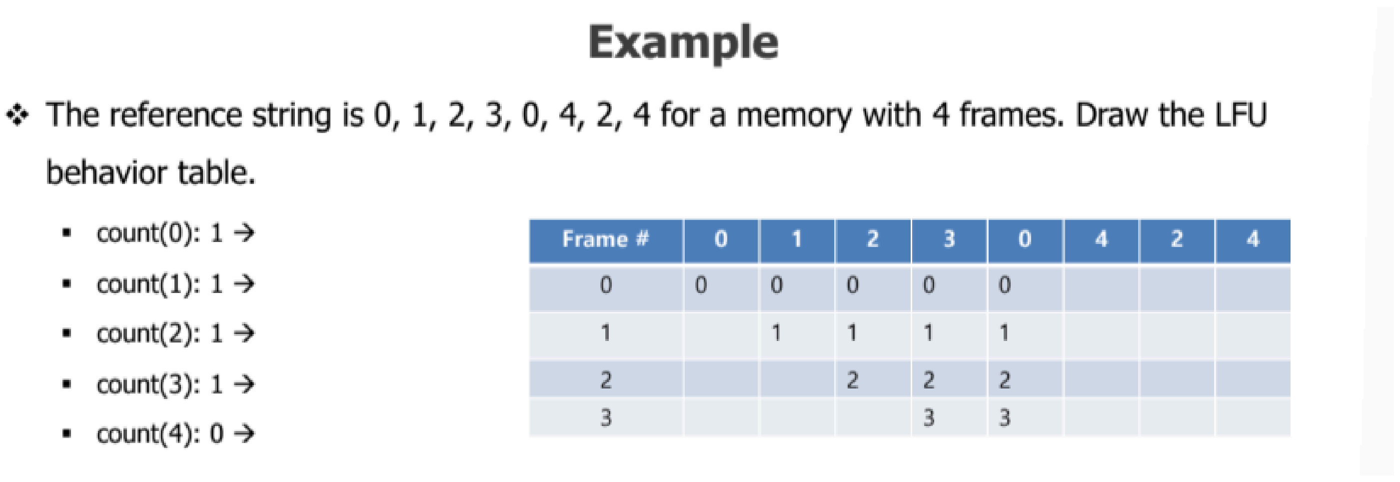


LRU-Approximation algorithm

- LRU 알고리즘은 여전히 느리고 hw support를 지원하는 computer system이 적음
- use bit을 사용한 LRU-approximation algorithm
  - 각 페이지에 1비트짜리 플래그 use bit을 설정
  - 하드웨어가 페이지 참조 시 자동으로 1로 설정
- Additional-Reference-Bits Algorithm
  - use bit는 한 번 참조되면 오래 되었더라도 최근에 참조된 것과 동일한 지위를 가짐
    -> 정보 부족
  - reference bit 사용
    - 각 페이지마다 1비트짜리 reference bit가 존재
    - 페이지가 참조되면 하드웨어가 1로 설정
  - 8비트 기록 필드 추가
    - 페이지마다 8비트짜리 히스토리 저장공간이 생김
    - high <-----> low
    - 현재 reference bit를 most left bit에 저장
    - regular interval 마다 OS가 다음 작업 수행
      - right shift 1 bit
      - discard most right(lowest) bit
  - 가장 값이 작은 페이지를 victim
- second-chance algorithm
  - 기본 구조는 FIFO 큐
  - 큐의 앞에서부터 페이지 검사
  - use bit이 1이면 give second chance (큐 뒤로 보내면서 use bit = 0)
  - use bit이 0이면 victim
- enchanced second-chance algorithm
  - 기존 second-chance는 reference bit만 고려
  - reference bit와 modify bit(dirty bit) 사용
    - rencently referenced만 고려하지 않고 write cost까지 고려
  - four possible cases (priority)
    1. R(0) M(0) : 최우선으로 교체
    2. R(0) M(1) : 사용 안했으므로 교체 후보, but 쓰기 비용이 존재하므로 보류
    3. R(1) M(0) : 최근에 사용됐으므로 다시 쓸 확률이 높음
    4. R(1) M(1) : 교체될 일이 거의 없음
  - circular queue를 돌면서 가장 적합한 후보 탐색
  - 최악의 경우 큐를 여러번 돌아야할 수 있음

Counting-Based Algorithm

- 페이지마다 reference된 횟수 counter로 tracking
- LFU (Least Frequently Used) algorithm
  - 가장 적은 count page를 replace
  - active하게 사용된 page일 수록 reference count가 많을 것이라 가정
- MFU (Most Frequently Used) algorithm
  - 가장 많은 count page를 replace
  - 가장 적은 count를 가진 page가 곧 사용될 것이라 가정



Example

❖ The reference string is 0, 1, 2, 3, 0, 4, 2, 4 for a memory with 4 frames. Draw the LFU behavior table.

- count(0): 1 →
- count(1): 1 →
- count(2): 1 →
- count(3): 1 →
- count(4): 0 →

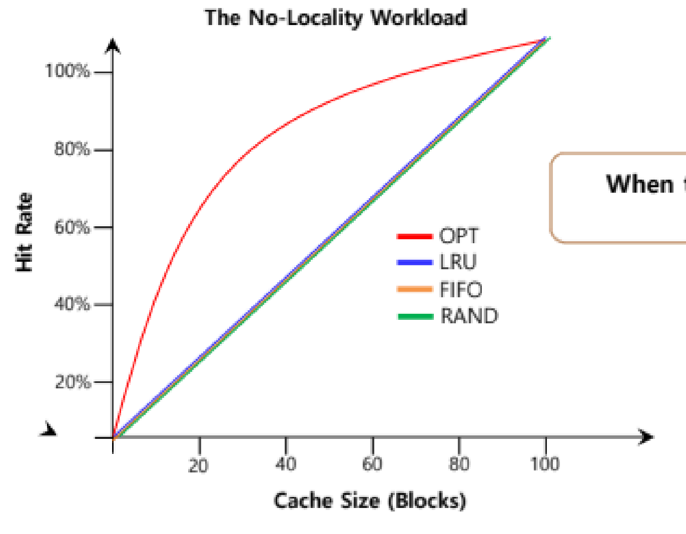
Frame #	0	1	2	3	0	4	2	4
0	0	0	0	0	0			
1		1	1	1	1			
2			2	2	2			
3				3	3			

Wokrload Example

No-Locality Workload

- 무작위로 페이지에 접근하는 시뮬레이션

- 무작위 접근이므로 no locality

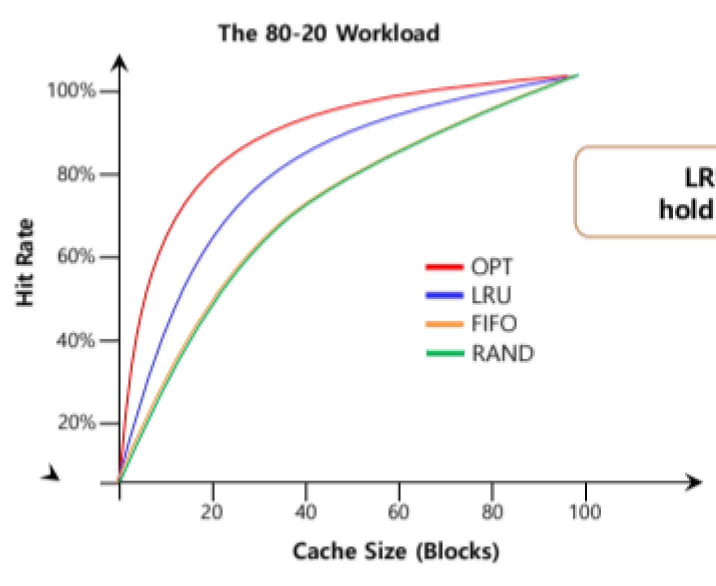


When the cache is large enough to fit the entire workload, it also **doesn't matter** which policy you use.

- optimal은 미래를 예측하는 것과 같으므로 항상 최적의 hit rate (평가 기준으로 사용)
- cache는 메모리를 의미
- 접근하려는 디스크 페이지가 전부 memory에 담겨있으면 당연히 hit rate 100%
- locality가 없다면 모든 알고리즘이 무의미하다는 것을 보여줌

The 80-20 Workload

- 80%의 reference는 20%의 페이지로부터 만들어지고 나머지 20%의 reference는 나머지 80%의 페이지로부터 만들어짐

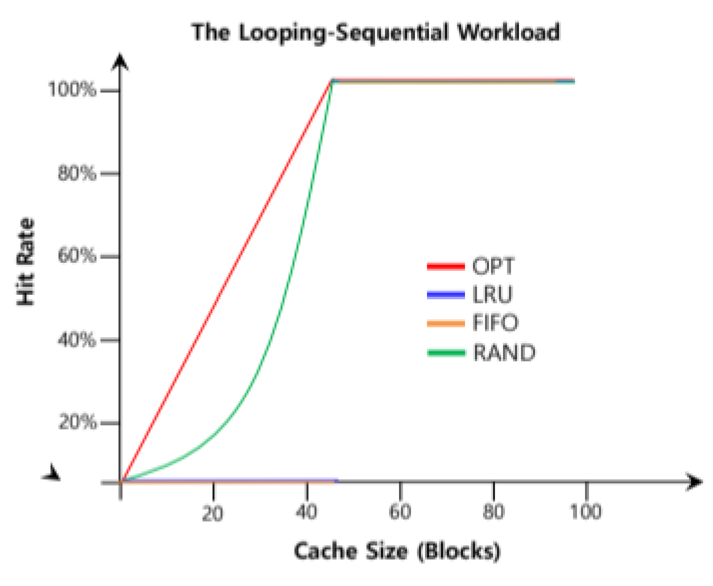


LRU is more likely to hold onto the **hot pages.**

- LRU가 가장 OPT에 근접한 hit rate를 보여줌줌
- FIFO는 거의 random access랑 hit rate가 유사하다는 것을 볼 수 있음

The Looping Sequential

- 페이지 50개를 순서대로 계속 1만번 looping



- 페이지가 50개이므로 cache (memory) 크기가 50만 넘으면 모든 페이지가 memory에 담길 수 있음
- LRU는 가장 오래된 페이지를 버리는데 page access가 sequential 하므로 이 상황에선 FIFO와 같은 역할을 하게 됨
- page access가 sequential할 경우 LRU와 FIFO는 cache가 50 미만이면 절대 hit가 일어날 수 없음
- 오히려 random replace의 hit rate가 더 높은 상황

Allocation of Frames

- 각 프로세스는 최소한의 프레임 수만 필요
- 최소 프레임 수는 컴퓨터 아키텍쳐가 정함
  - IBM 370에선 MOVE instruction을 다루기 위해 6페이지만 사용
  - instruction은 6바이트 밖에 안 하지만 두 페이지에 걸쳐있을 수 있으므로 2페이지 필요
  - indirect addressing 방식이면 from과 to를 처리해야하므로 각각 2페이지 필요

Fixed allocation

- equal allocation
  - 만약 100프레임과 5 프로세스가 있을 경우 각 프로세스에 20 프레임 할당
- proportional allocation
  - 프로세스의 크기에 따라 할당
  - 프로세스A와 B가 각각 10, 127의 크기를 갖고 총 프레임이 64개일 경우
  - 프로세스 A에 할당되는 프레임 수 x, 프로세스 B에 할당되는 프레임 수 y
  - 10 : 127 = x : y
  - x + y = 64

priority allocation

- 사이즈가 아닌 prioirties로 proportional allocation
- 프로세스가 page fault를 발생시키면 lower priority의 프로세스에서 replacement할 frame을 선택 (빼앗기)

Global replacement

- 모든 frame의 집합으로부터 frame을 replacement해서 사용
- 다른 process의 frame을 뺏는 것이므로 process의 실행시간이 제각각임
  - 특정 process만 page fault가 자주 발생할 수 있음
- 하지만 일반적인 경우 더 높은 throughput을 기대할 수 있음
  - 넓게 보면 더 적은 page fault와 더 빠른 메모리 접근을 하기 때문

Local replacement

- 각 프로세스가 오직 자신에게 할당된 프레임 내에서만 페이지 교체
  - 다른 프로세스의 프레임을 뺏을 수 없음
- 어떤 프로세스든 자기가 가진 메모리만 사용하므로 실행시간이 일정하고 안정적
- 자기 프레임 안에서만 페이지를 교체하므로 다른 프로세스와 간섭이 없음
- 대신 어떤 프로세스는 프레임이 남아도는데 다른 프로세스는 프레임이 부족할 수 있음

Thrashing

Thrashing이란?

- 몸부림치다, 허우적대다, 마구 휘젓다라는 뜻
- 프로세스가 계속해서 페이지폴트만 발생시키느라 해야할 일은 하지 못 하고 페이지 교체에만 시간을 다 쓰는 현상

왜 Thrashing이 발생하는가?

- process에 할당된 frame(memory)가 너무 적을 경우
- process는 필요한 page를 disk에서 가져와야 함(page fault)
- 다른 page를 frame에서 내보냄
- 방금 내보낸 frame에 들어있던 page가 다시 필요해질 경우 (page fault)

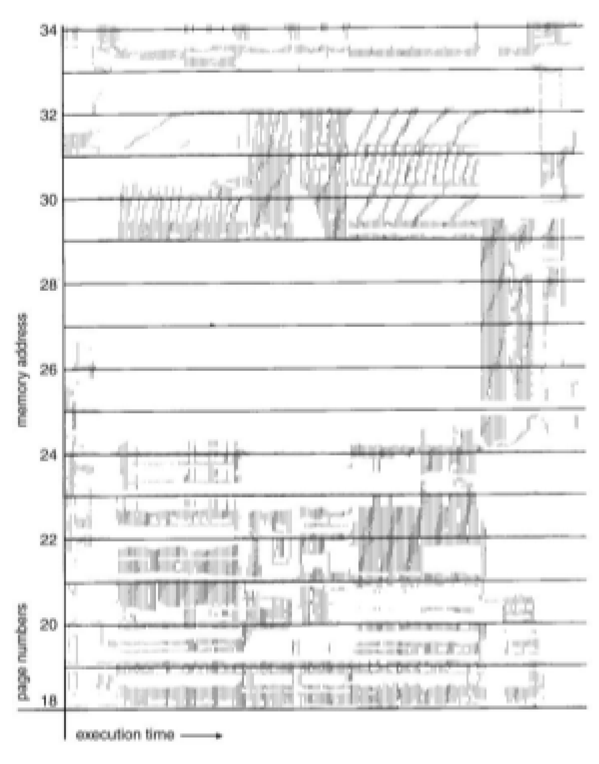
Trashing의 결과

- CPU가 I/O를 기다리는 것에 비해 process의 work를 처리하는 시간이 적으므로 low CPU utilization
- OS는 low CPU utilization을 보고 CPU를 더 써먹을려고 함 (increase multiprogramming)
- OS가 다른 프로세스도 system에 넣어버림
- 상황 악화

Thrashing 방지하는 방법

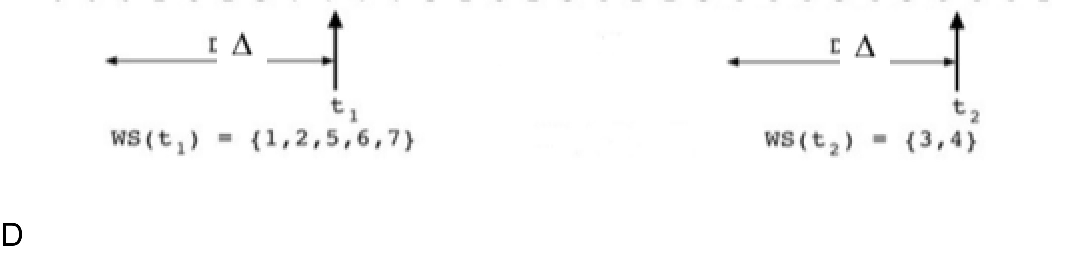
- 필요한 만큼 frame을 최대한 많이 제공 받아야 함
- 그 필요한 만큼이 어느정도인가?
  - Locality model로 파악 가능
- Locality model of process execution
  - 프로세스는 실행 중 특정 페이지들을 집중적으로 사용한다는 가정
  - ex) function call
    - function call 시 그 function의 code, stack, data 등이 메모리에 같이 필요
    - 이들이 하나의 locality를 형성
  - process는 하나의 locality에서 다른 locality로 이동하면서 실행됨
  - locality들은 서로 overlap 되기도 함
- 이렇게 보면 할당된 frame의 크기보다 현재 locality의 크기가 더 크기 때문에 thrashing이 발생했다고 볼 수도 있음
- Thrashing은 앞서 배운 local page replacement와 prioirty page replacement로 Thrashing현상을 제한시킬 수 있음

실제 locality 시각화 (memory reference pattern)



Working-Set model

- locality에 근거한 모델
- the working-set window
  - 삼각형(delta)으로 표현
  - 최근 delta번의 참조된 page 집합
- working-set of process Pi
  - WSi 로 표현
  - 최근 delta번의 참조된 page 중에서 Pi가 사용한 page들의 집합
- the size of WSi
  - WSSi로 표현
  - 최근 delta번의 참조된 page 중에서 Pi가 사용하고 있는 page의 수
  - 만약 delta가 너무 작으면 어떤 locality도 포함시키지 못할 것
  - delta가 너무 크면 몇몇 locality들을 포함
  - delta가 무한대로 크면 모든 프로그램을 포함할 것임



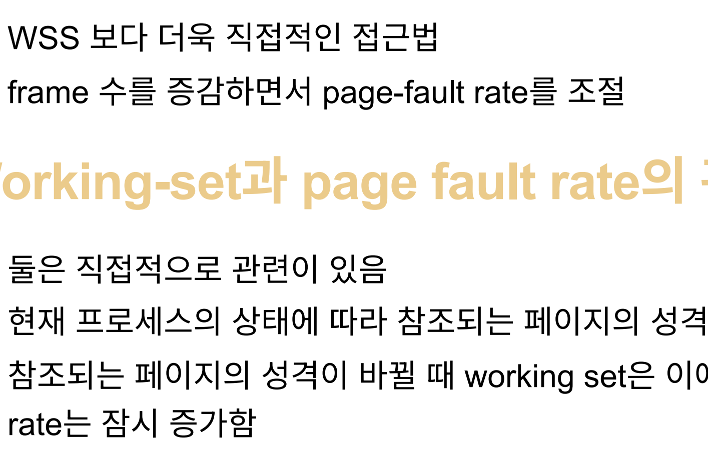
- D
  - the total demand for frames
  - WSSi의 총합과 같음
- D > m(total number of frames)일 경우 Thrashing 발생
  - 이 경우 프로세스 중 하나를 중단 시켜야 함

working-set tracking 하기

- 최근에 참조한 페이지들의 집합(working-set)을 유지하는 건 어렵고 비용이 많이 소모
- interval timer와 reference bit를 사용한 approximate 방법
- delta(최근 참조된 page들의 집합)가 1만개라 가정하자
- 메모리에 각 페이지마다 2비트의 referece bit를 사용
- 5천 references 마다 timer interrupt 발생
- page가 refence 되면 R(0) = 1
- timer interrupt 시 2비트의 reference bit를 left shift
- 2비트 중 1개만이라도 1이면 해당 page는 working set에 있다고 판단
- working set이 1만개의 페이지고 timer interrupt는 5천 페이지마다 발생하므로 timer interrupt가 두 번 발생했을 때 1이 없으면 working set에 없다는 것을 알 수 있음

Acceptable PFF

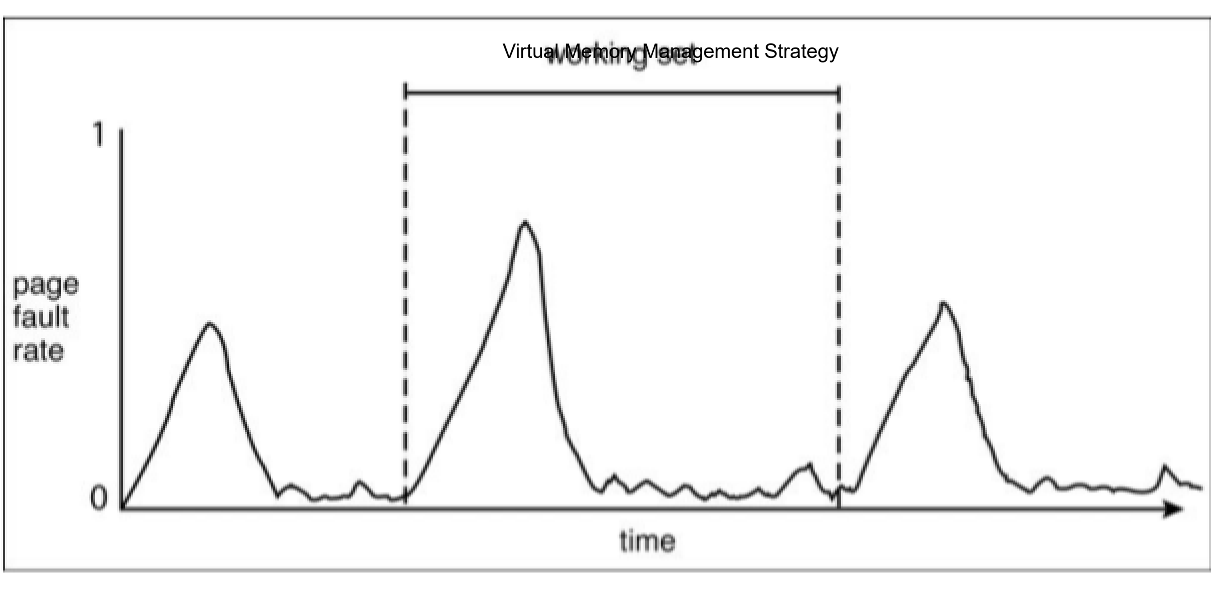
- establish acceptable page fault frequency



- WSS 보다 더욱 직접적인 접근법
- frame 수를 증감하면서 page-fault rate를 조절

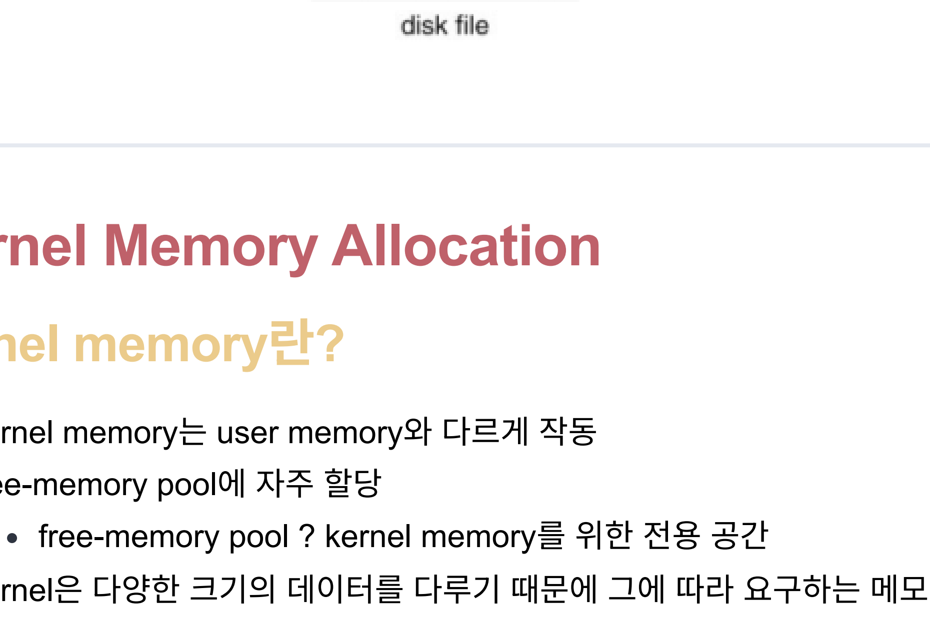
Working-set과 page fault rate의 관계

- 둘은 직접적으로 관련이 있음
- 현재 프로세스의 상태에 따라 참조되는 페이지의 성격도 달라지는데 working set도 그에 따라 계속 바뀜
- 참조되는 페이지의 성격이 바뀔 때 working set은 이에 적응하는 시간이 있는데 그 시간 동안 page fault rate는 잠시 증가함
- OS는 이에 따라 메모리를 잠시 많이 할당해주고 page fault rate가 안정되면 다시 메모리 할당을 줄이는 식으로 유동적으로 관리해야함



Memory-Mapped Files

- disk block을 memory에 있는 page에 mapping 함으로써 routine memory access 하듯이 file I/O를 다룰 수 있는 방법
- file은 처음에 demand paging 기법을 사용해서 필요할 때만 read됨
- file system이 file에서 page 크기만큼 read
- 그 후의 read/write는 마치 평범한 memory access(배열 접근)처럼 다뤄짐
  - 실제로 cpp에서도 file을 read하고나면 배열 처럼 다룰 수 있음
  - 그리고 read 하면 모든 데이터가 불러와지는 게 아닌 필요한 데이터만 불러와짐
- 여러 프로세스가 같은 파일을 같은 메모리 주소에 mapping 할 수 있음
  - 이 파일이 공유 메모리에 있는 것 처럼 사용 가능
  - cpp에서 같은 파일을 한 프로세스는 read, 다른 파일은 write 하는 모습
  - 이때 한 프로세스가 write 하면 다른 프로세스는 수정된 데이터를 볼 수 있음



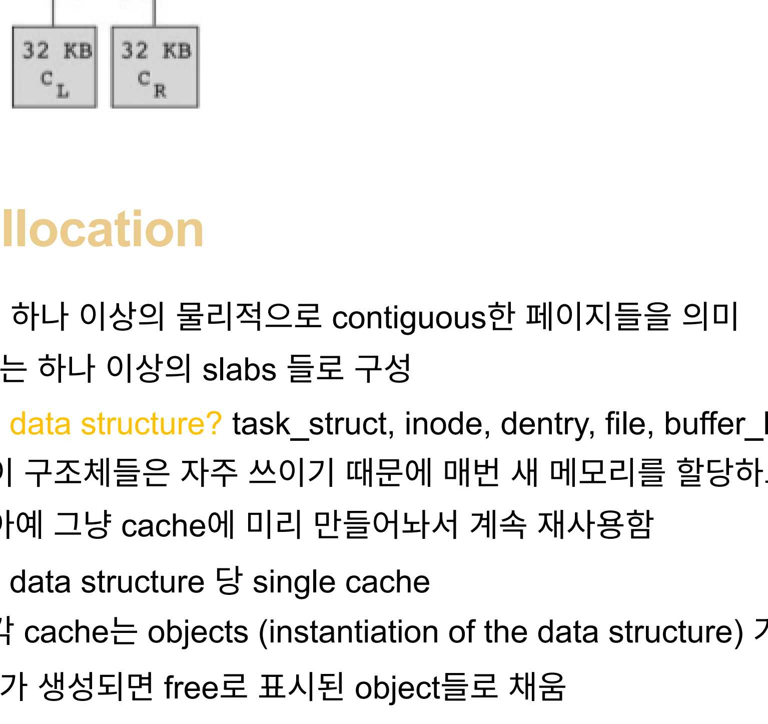
Kernel Memory Allocation

kernel memory란?

- kernel memory는 user memory와 다르게 작동
- free-memory pool에 자주 할당
  - free-memory pool ? kernel memory를 위한 전용 공간
- kernel은 다양한 크기의 데이터를 다루기 때문에 그에 따라 요구하는 메모리 크기도 다양함
- 어떤 kernel emmory는 contiguous까지 해야함

buddy system

- 물리적으로 contiguous한 페이지로 구성된 고정 크기의 segment에 memory를 할당하는 것
- power-of-2 allocator를 사용한 memory 할당
  - memory 크기가 power of 2
  - 요청된 memory 크기와 가장 가까운 power of 2 크기로 할당
  - 할당된 memory가 너무 많으면 이전 2의 배수로 split 됨



slab allocation

- slab은 하나 이상의 물리적으로 contiguous한 페이지들을 의미
- cache는 하나 이상의 slabs 들로 구성
- kernel data structure? task_struct, inode, dentry, file, buffer_head, sk_buff, ...
  - 이 구조체들은 자주 쓰이기 때문에 매번 새 메모리를 할당하고 초기화하면 비효율적
  - 아예 그냥 cache에 미리 만들어놔서 계속 재사용함
- kernel data structure 당 single cache
  - 각 cache는 objects (instantiation of the data structure) 가 자리를 차지함
- cache가 생성되면 free로 표시된 object들로 채움
- structure가 store되면 object를 used로 표시
- slab가 used object로 가득 차면 empty slab에서 다음 obejct가 할당됨
- 만약 empty slab가 없으면 new slab allocated
- 장점 : no fragmentation, fast memory request satisfaction

Other Issues

Prepaging

- process를 startup할 때 생기는 대량의 page fault를 줄이기 위한 기법
- process가 reference 하기 전에 page의 일부 혹은 전부를 prepage
- S 만큼의 page를 prepage 했는데 이 중 A 만큼만 사용됐다고 가정하자
  - S x A 만큼의 page fault 감소 vs S x (1 - A) 만큼의 불필요한 page 증가
  - A가 0에 가까울 수록 prepaging이 오히려 독이 됨

Page Size를 어느 정도로 할 것인가?

- internal fragmentation을 막기 위해선 smaller page
- small page table을 위해선 bigger page
- small latency를 위해선 smaller page
- locality를 위해선 smaller page (better resolution)

TLB Reach

- TLB Reach는 TLB를 통해 access 할 수 있는 memory의 크기
- TLB Reach = TLB size x page size
- 만약 process의 working-set 전부가 TLB에 저장된다면 가장 이상적
- TLB Reach를 increase 하기 위해선
  - TLB의 entries 수 늘리기 (expensive)
  - page size 늘리기 (fragmentation 우려)
  - multiple한 page size 지원 (OS가 TLB을 관리해줘야 함)

Inverted page table

- inverted page table?
  - 일반 page table은 프로세스 별로 virtual address space 전체를 mapping한 정보를 가짐
  - 그래서 virtual address space 크기에 비례해 매우 커짐
  - page table은 프로세스 별로 존재
  - 반대로 inverted pabe table은 physical memory의 각 frame 마다 어떤 virtual page가 올라와 있는지 저장
  - inverted page table은 시스템 전체에 단 하나
- 하지만 inverted page table은 프로세스의 logical address space에 대한 모든 정보를 갖지 않음
  - 각 virtual page가 어디에 있는지 알려줄 external page table가 필요

I/O Interlock

- 어떤 프로그램이 I/O 작업 중일 때 그 메모리 영역이 swap out 되지 않도록 lock
- lock bit 사용
- paging이 더 복잡해짐

program structure

```
Program 1
Arrary[100][100];

for(i=0; i<100; i++)
          for(j=0; j<100; j++)
                    A[i][j]=i*j;

→ Page faults: 100회
```

```
Program 2
Arrary[100][100];

for(j=0; j<100; j++)
          for(i=0; i<100; i++)
                    A[i][j]=i*j;

→ Page faults: 10000회
```

평소대로 짜면 page fault가 적음