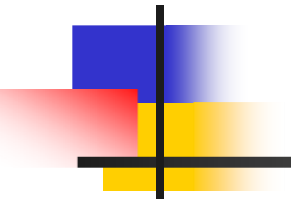
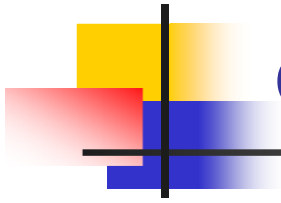


Operating Systems
(Virtual Memory)

Chapter 10

These lecture materials are modified from the lecture notes written by A. Silberschatz, P. Galvin and G. Gagne.

August, 2022



Outline

- Background
- Demand Paging
- Page Replacement
- Allocating Kernel Memory

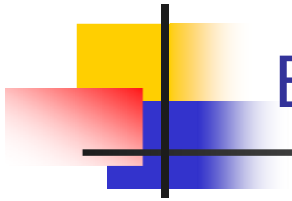
Background

Background

- 가상 메모리는 완전히 메모리에 있지 않은 프로세스의 실행을 허용하는 기술임
- 이 방식의 주요 이점은 프로그램이 물리적 메모리보다 클 수 있다는 것

- 가상 메모리는 메인 메모리를 매우 크고 균일한 스토리지 어레이로 추상화하여 프로그래머가 볼 때 논리적 메모리를 물리적 메모리와 분리함
- 프로그래머를 메모리 저장 제한 문제에서 해방시킴

- 그러나 가상 메모리는 구현하기 쉽지 않으며 부주의하게 사용하면 성능이 크게 저하될 수 있음



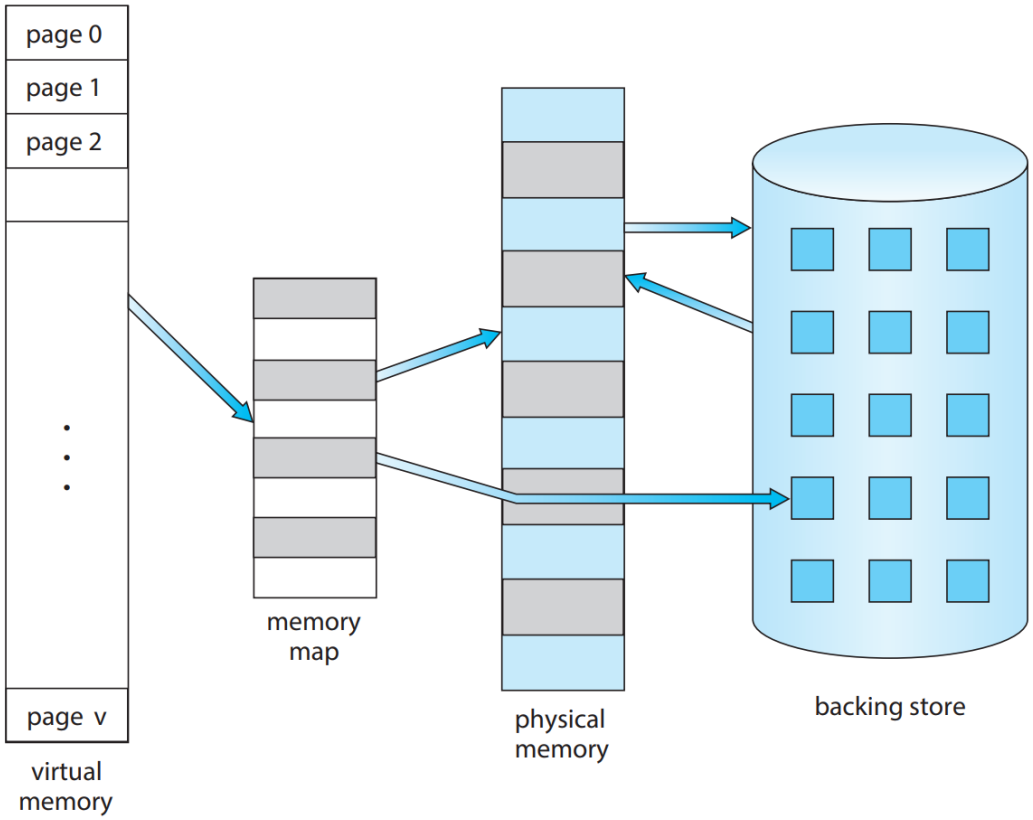
Background

- **메모리 관리 알고리즘이 필요한 이유:**
  - 실행 중인 명령어는 반드시 물리 메모리에 있어야 하기 때문.

- 당연히 제일 간단한 방법은 그냥 가상 주소 공간을 전부 물리 메모리에 올려버리는 것
  - => 너무 많은 낭비가 발생

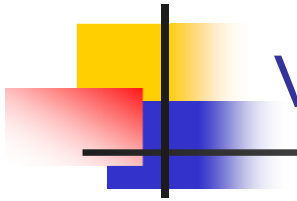
Virtual memory

- 개발자가 인식하는 논리 메모리를 실제 물리 메모리와 구분함.



메모리 맵은 특정 프로세스의 가상 메모리와 해당 공간이 물리적 메모리 주소와 어떻게 관련되는지 저장

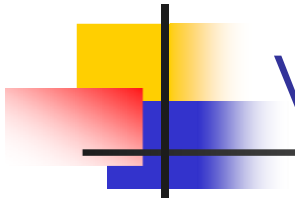
Figure 10.1 Diagram showing virtual memory that is larger than physical memory.



Virtual memory

- 물리 메모리는 페이지 프레임으로 구성되어있어 프로세스에 할당된 물리 페이지 프레임은 연속되지 않았을 수도 있음.

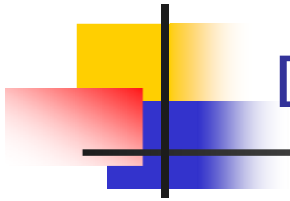
- **Memory management unit (MMU)**가 논리 페이지를 메모리 내의 물리 페이지 프레임으로 매핑



Virtual memory

- 프로그램은 더 이상 사용 가능한 물리적 메모리의 양에 의해 제약을 받지 않음.
- 각 프로그램은 물리적 메모리를 덜 차지할 수 있기 때문에 CPU 사용률과 처리량이 증가
- 프로그램의 일부를 메모리로 로드하거나 교체하는 데 더 적은 I/O가 필요하므로 각 프로그램이 더 빠르게 실행됨.

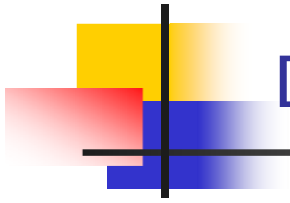
Demand Paging



Demand Paging

- 실행할 프로그램을 보조 기억 장치에서 메모리로 올릴 때 프로그램 전체를 프로그램 실행 시에 물리 메모리에 올릴 수도 있음

- **그러나 프로그램 전체가 필요하지 않을 수도 있음.**



Demand Paging

- 필요한 페이지만 메모리에 올리는 것
- 가상 메모리 시스템에서 일반적으로 사용하는 방법

- 프로그램 실행 중에 추가적인 페이지를 요구할 때만 페이지를 적재함.

- **즉, 접근하지 않은 페이지는 절대로 메모리에 올라갈 일이 없음.**

Demand Paging

- 실행 중인 부분이 메모리에 올라가있다면, 나머지 부분은 보조 기억 장치에 있다는 뜻이므로, 이 둘을 구분할 수 있게 하드웨어 단에서 지원해줘야함.
  - vaild-invaild bit가 사용됨.

- **vaild 상태:**
  연관된 페이지가 유효한 페이지(프로세스의 가상 주소 공간에 속함)이면서 메모리 위에 있음.
- **invaild 상태(비트가 설정되지 않음):**
  페이지가 유효하지 않거나(프로세스의 가상 주소 공간에 속하지 않음) 유효하지만 현재 보조 기억 장치 안에 있다는 의미

Demand Paging

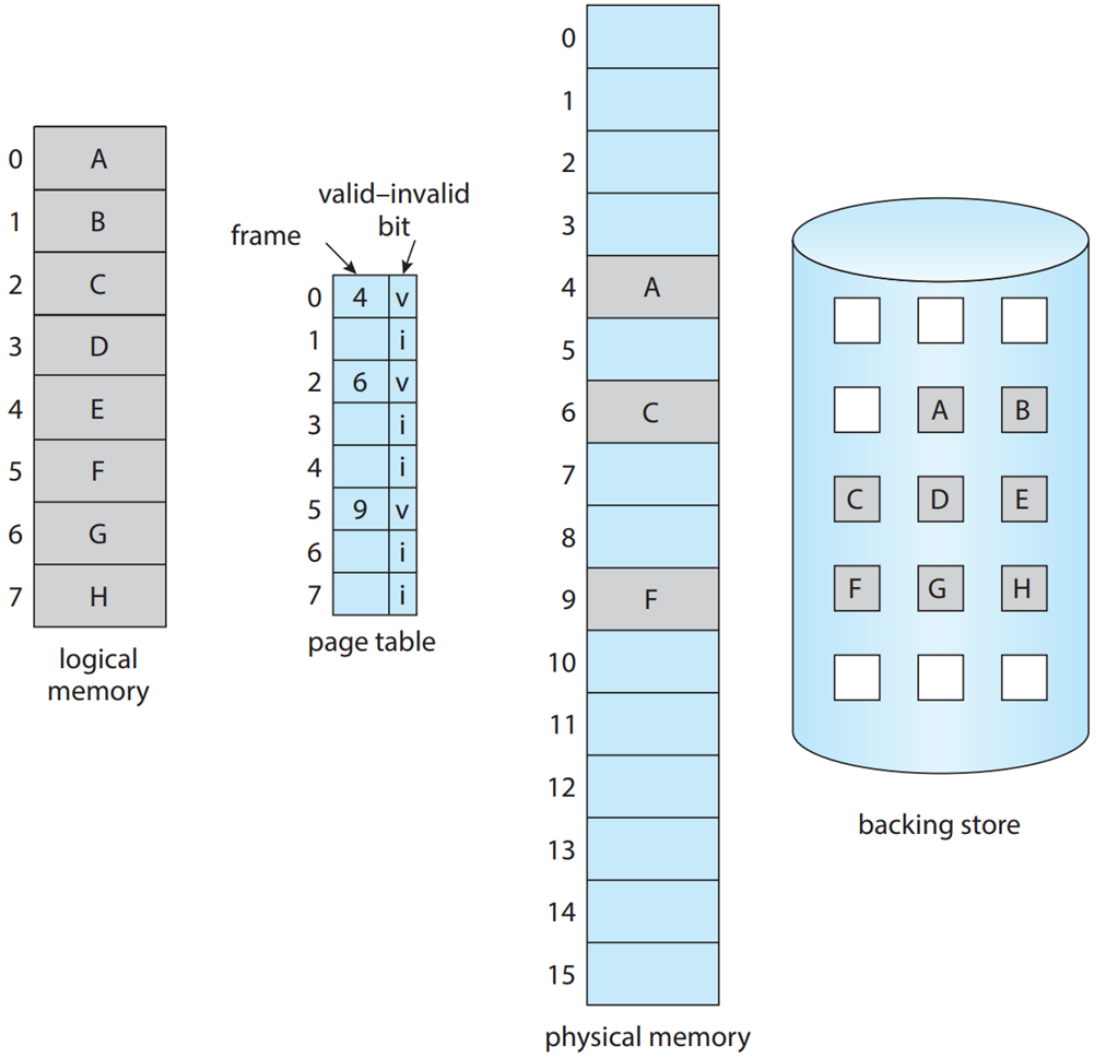


Figure 10.4 Page table when some pages are not in main memory.

Page fault

- 프로세스가 메모리에 없는 페이지에 접근하려고 하면 페이지 부재 page fault가 발생하게 됨.
- **page fault의 처리 방법:**

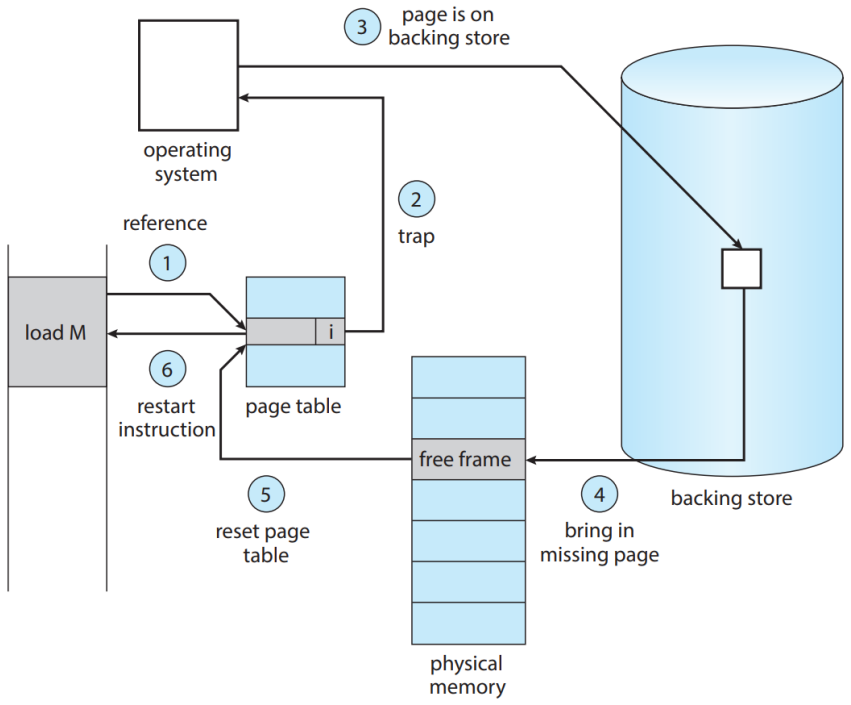
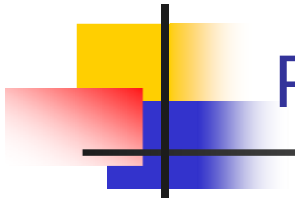
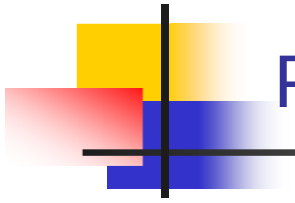


Figure 10.5 Steps in handling a page fault.



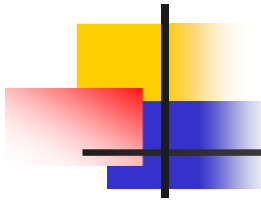
Page fault

- 1. 참조가 유효한 메모리 액세스인지 또는 유효하지 않은 메모리 액세스인지 결정하기 위해 이 프로세스에 대한 내부 테이블을 확인.
- 2. 참조가 유효하지 않은 경우 프로세스를 종료. 유효했지만 아직 해당 페이지를 가져오지 않은 경우 이제 페이지를 올림.
- 3. free 상태의 프레임을 찾음
- 4. 올려야할 페이지를 새롭게 할당한 프레임에 읽어올 보조 기억 장치 연산을 스케줄링해줌.
- 5. 저장소에서 다 읽어오면 내부 테이블이랑 페이지 테이블을 수정. 페이지가 메모리에 올라갔다는 걸 알려주기 위함.
- 6. 트랩에 의해 중단된 명령을 다시 시작. 프로세스는 이제 항상 메모리에 있었던 것처럼 페이지에 액세스할 수 있음.



Pure demand paging

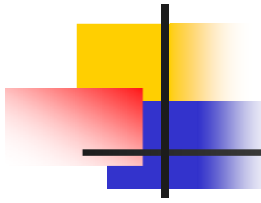
- 극단적인 경우에는 애초에 메모리에 페이지 단 한 개도 없이 프로세스를 실행할 수 있음.
- 바로 page fault가 발생하고, 필요한 페이지들을 계속 메모리에 올려주면서 실행하게 됨.



Performance of demand paging

- 요구 페이징은 컴퓨터 시스템에 성능에 상당한 영향을 줄 수 있음.
- 요구 페이징된 메모리의 effective access time을 계산해야 함.

- effective access time = (1 − p) × ma + p × page fault time
  - p : page fault 확률(0 ≤ p ≤ 1)
  - ma : memory access time



Page fault time

- Page fault time에 제일 큰 부분을 차지하는 세 부분
  - 1. 페이지 부재 인터럽트 서비스
  - 2. 페이지 읽기
  - 3. 프로세스 재시작하기

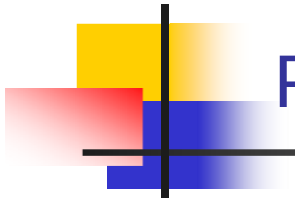
- 이 중 1, 3은 코딩을 잘 하면 몇백 명령어로 줄일 수 있음.

Performance of demand paging

- 평균적으로, page fault time = 8 ms
  memory access time = 200 ns

- effective access time = (1 - p) × 200 + p × 8,000,000
- = 200 + 7,999,800 × p

- effective access time은 page-fault rate에 직접적으로 비례함.
- **Demand paging에서는 page-fault rate을 최대한 낮추는 게 제일 중요함.**



Page fault time

- Page fault time에 제일 큰 부분을 차지하는 세 부분
  - 1. 페이지 부재 인터럽트 서비스
  - 2. 페이지 읽기
  - 3. 프로세스 재시작하기

- 이 중 1, 3은 코딩을 잘 하면 몇백 명령어로 줄일 수 있음.

Page Replacement

Page Replacement

- 프로세스 실행 중에 page fault가 발생하면, OS는 해당 페이지를 보조 기억 장치에서 찾음.
- 하지만 사용 가능한 프레임이 없어 페이지를 올릴 수가 없음.

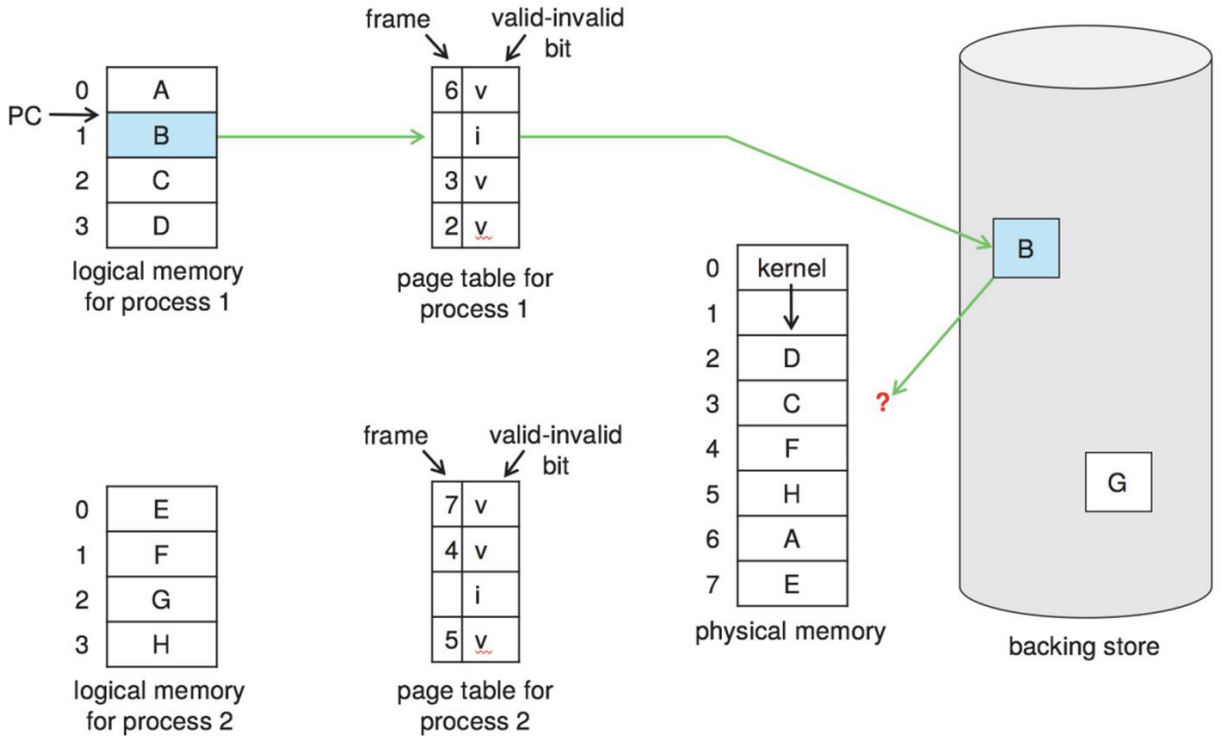
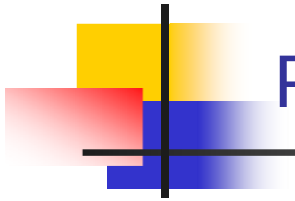


Figure 10.9 Need for page replacement.



Page Replacement

- 이 경우, 기존 페이지 중 안 쓰는 페이지 하나를 백업 저장소 (보조 기억 장치)에 작성해주고, 그 자리에 해당 페이지를 올림.
- 즉 대부분의 OS는 swapping with Paging을 사용

Page Replacement

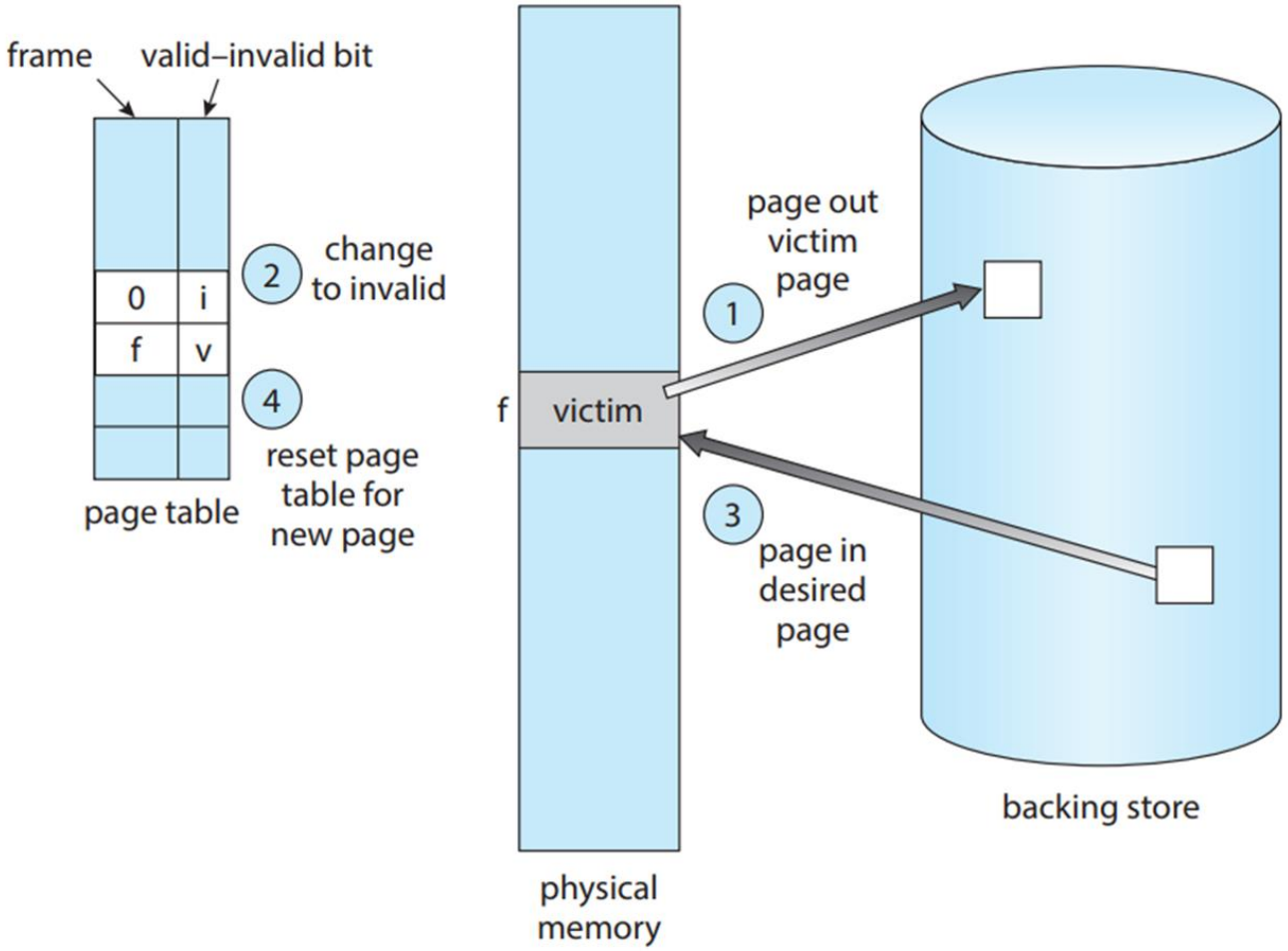
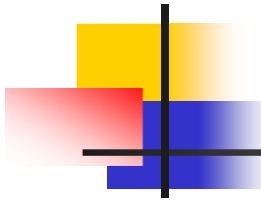
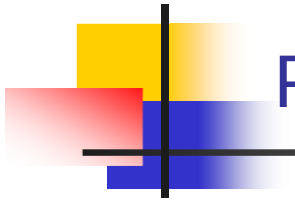


Figure 10.10 Page replacement.



Page Replacement

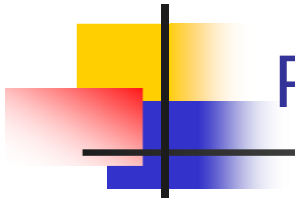
- 1. Page fault 발생.
- 2. 보조 기억 장치에서 필요한 페이지의 위치를 찾음
- 3. 사용 가능한 프레임을 찾음
  - 사용 가능한 프레임이 있으면 사용
  - 없으면 Page Replacement 알고리즘으로 victim frame을 정함.
    victim frame을 보조 기억 장치에 작성
    - 페이지와 프레임 테이블을 갱신 (페이지는 수정되었을 경우만)
- 4. 필요한 페이지를 새롭게 사용 가능해진 프레임에 읽어온 후, 페이지와 프레임 테이블 갱신
- 5. page fault가 발생했던 프로세스를 다시 재개



Page Replacement

- 사용 가능한 프레임이 없을 경우
  페이지 이동이 두 번(페이징 아웃, 페이징 인 각각 한 번씩) 발생
  - page fault time이 두 배가 되어 effective access time이 증가

- **modify bit(혹은 dirty bit)로 오버헤드를 줄일 수 있음.**
  - 적절한 victim frame을 찾게 도와줌.



Page Replacement

- 페이지 교체를 하려고 할 경우 우선 페이지의 modify bit를 확인.
- 비트가 설정된 경우 페이지가 수정되어, 페이지를 기억장치에 업데이트 해주어야 함.

- modify bit가 0인 frame을 victim frame으로 선택하면, 페이지를 보조 기억 장치에 업데이트 해주지 않아도 됨.
  - page fault time이 modify bit가 1인 frame의 절반이므로, **effective access time을 낭비하지 않음.**

Page Replacement Algorithm

- **목표:**
  page fault rate를 최소화 하는 것.

- **평가:**
  특정 reference string에서 알고리즘을 실행하고
  해당 문자열의 페이지 오류 수를 계산하여 평가함.

- **모든 예의 reference string:**
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

FIFO Page Replacement

- 가장 간단한 페이지 교체 알고리즘. 선입선출 알고리즘.
- 각 페이지가 메모리에 올라간 시간 순으로 FIFO 교체됨.

- **장점:** 이해와 구현이 쉬움
- **단점:** 성능 보장 불가.

FIFO Page Replacement

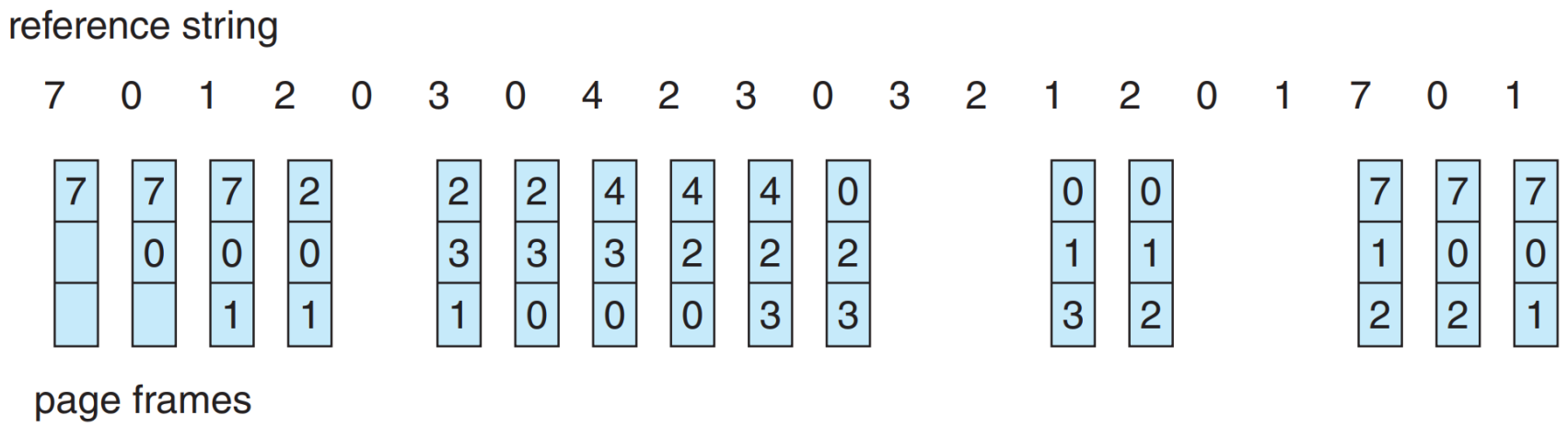


Figure 10.12 FIFO page-replacement algorithm.

Belady's anomaly(벨라디의 변이)

- 할당 가능한 프레임 수가 증가할 때 page fault rate가 오히려 증가할 수 있다는 것
- 보통 메모리를 증가시키면 성능이 좋아질 거라고 하지만, **언제나 참은 아님.**

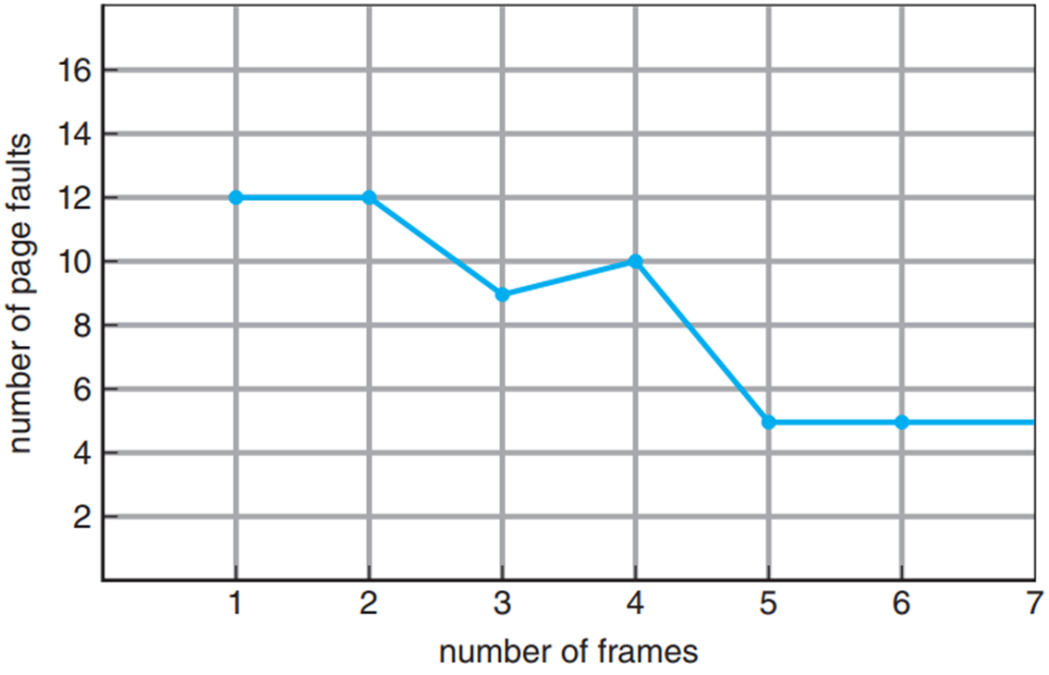


Figure 10.13 Page-fault curve for FIFO replacement on a reference string.

Optimal page-replacement algorithm

- 가장 오랜 기간 동안 사용하지 않을 예정인 페이지를 교체
- page fault rate이 제일 낮은 알고리즘. Belady's anomaly가 발생하지 않음.
- 고정된 프레임 수에서 가장 낮은 페이지 부재율을 보장

Optimal page-replacement algorithm

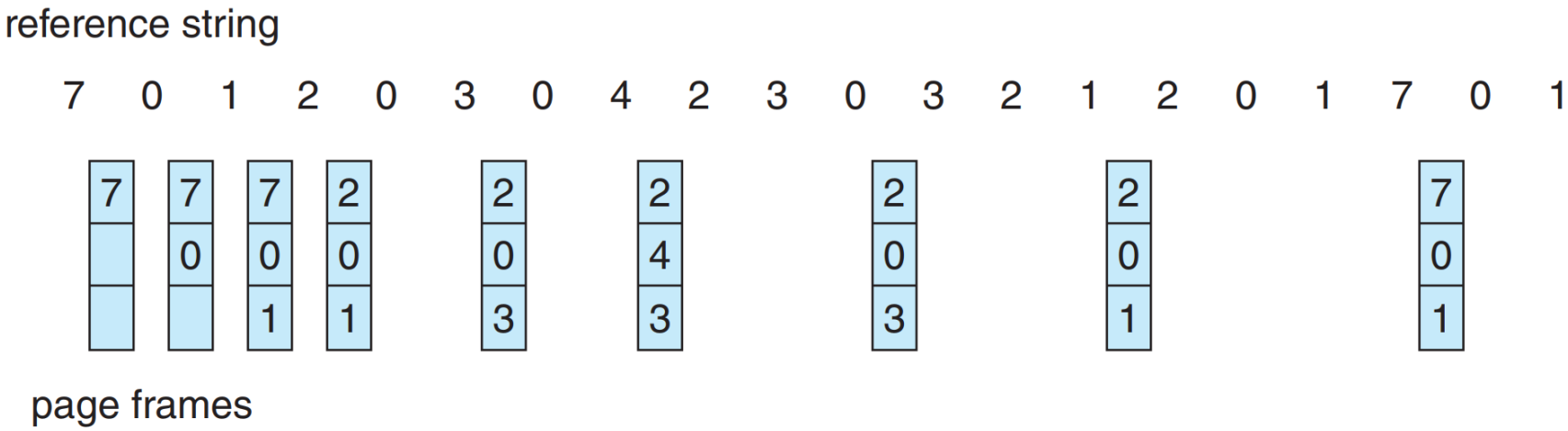
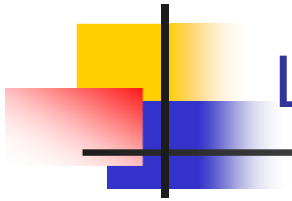


Figure 10.14 Optimal page-replacement algorithm.



LRU Page Replacement

- 가장 오랜 기간동안 사용하지 않은 페이지를 교체
- **LRU는 좋은 알고리즘으로 평가받아 매우 자주 사용됨**

LRU Page Replacement

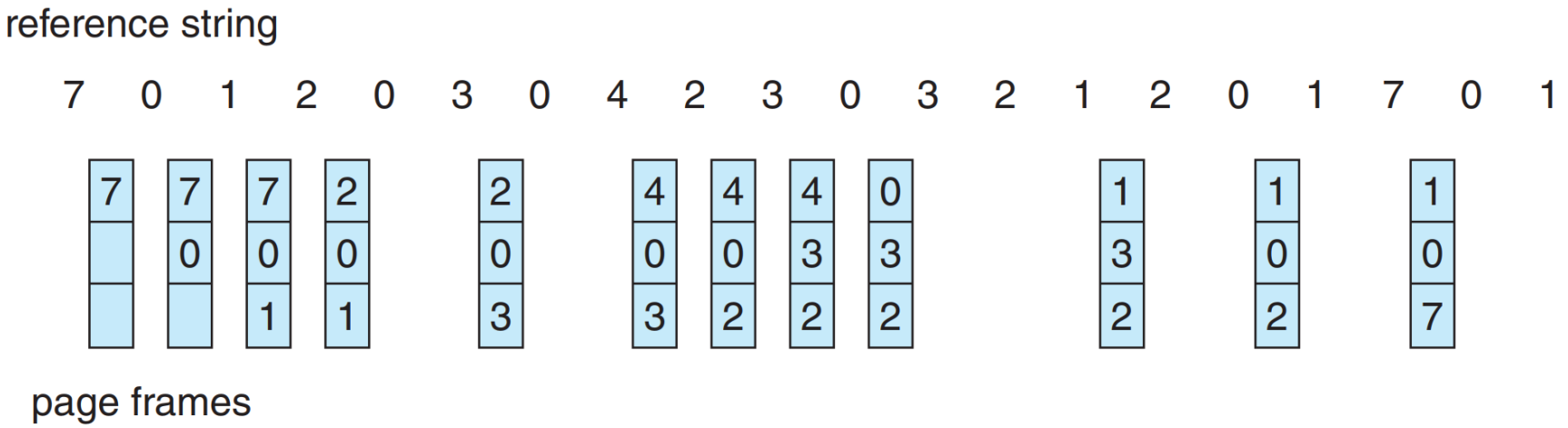


Figure 10.15 LRU page-replacement algorithm.

LRU Page Replacement

- **문제점:** 바로 가장 최근에 사용한 시간에 따른 프레임 순서를 어떻게 정의할 것인가?

- **Counter :** 메모리 참조할 때 마다 counting을 하고, 해당 page가 사용 될 때 해당 카운팅 값을 Page Table의 엔트리에 복사해줌. 즉 시간을 저장하는 것.
- **Stack:** 가장 최근에 사용한 페이지를 스택 중간에서 빼서, 스택의 가장 위에 올려놓는 것임.
  - 가장 최근에 사용한 페이지는 언제나 스택의 맨 위, 최근에 가장 덜 사용한 페이지는 언제나 맨 밑에 위치함.
  - 링크드 리스트로 구현하는 것이 좋음.

LRU Page Replacement

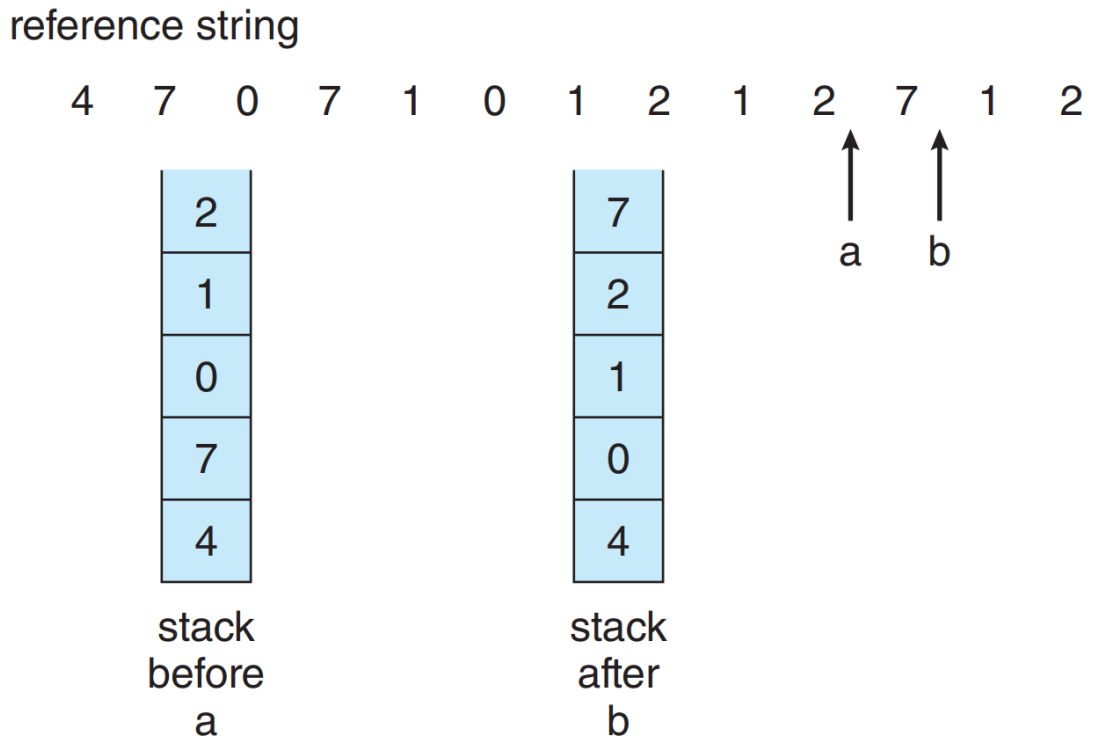
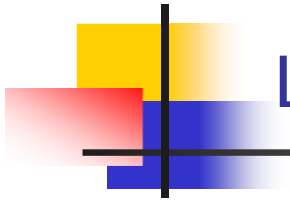


Figure 10.16 Use of a stack to record the most recent page references.



LRU Page Replacement

- 하드웨어 지원 없이는 LRU 교체 구현은 불가능함.
- Counter, Stack은 매 메모리 참조 때마다 갱신이 됨.
- 이때 갱신할 때마다 인터럽트를 발생시킨다면 매우 부담됨.

LRU-Approximation Page Replacement

- LRU-Approximation 알고리즘은 **Reference Bit**를 이용하여 구현하는 알고리즘 기법
  - 초기엔 운영체제가 모든 비트를 (0으로) 비워줌.
  - 프로세스가 실행되면서 하드웨어가 참조된 페이지의 비트를 (1로) 설정해줌.
- 이를 통해 참조된 페이지와 그렇지 않은 페이지를 구분 가능.
  - 물론 참조 순서는 모름
  - LRU에 참고할 기본 데이터로 사용

Second-chance(clock) page-replacement algorithm

- FIFO 알고리즘의 변형.
- 처음에 교체할 페이지를 선택할 때 **Reference Bit**을 확인.
  - 0이라면 이 페이지를 교체.
  - 1이라면 이 페이지에 한 번 더 기회를 더 줌. 즉 참조 비트를 0으로 변경한 후 남겨놓고, 다음 FIFO 페이지로 넘어감.

- 모든 비트가 1로 설정된 상태라면 FIFO와 같아짐.

Second-chance(clock) page-replacement algorithm

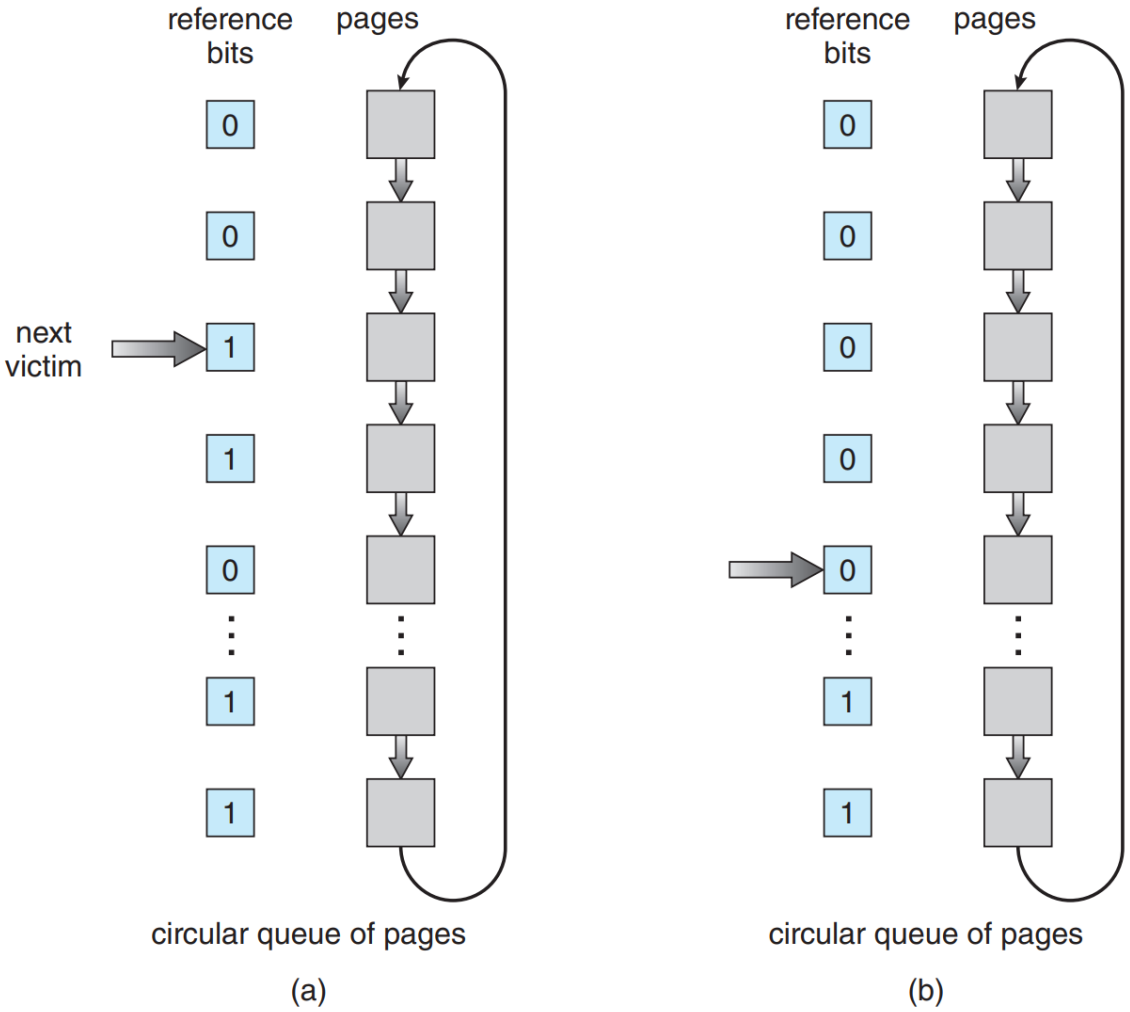
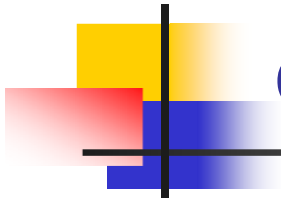
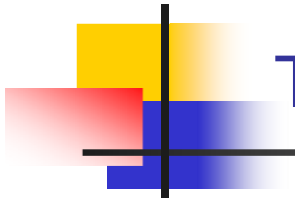


Figure 10.17 Second-chance (clock) page-replacement algorithm.



Counting-Based Algorithms

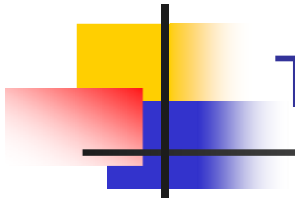
- Counting-Based 알고리즘은 각 페이지의 참조 수를 Page table 에 저장하고, 이를 통해 Victim 페이지를 결정하는 알고리즘.
- 1. LFU (Least Frequently Used)
  - LFU 알고리즘은 참조 수가 가장 적은 페이지를 Victim 페이지로 선택하는 것.
  - 페이지가 앞으로도 선택이 안될 것으로 판단한 것
- 2. MFU(Most Frequently Used)
  - MFU 알고리즘은 참조 수가 가장 많은 페이지를 Victim 페이지로 선택하는 것.
  - 페이지가 지금까지 많이 참조됐으니 앞으로는 참조되지 않을 것으로 판단한 것
- MFU나 LFU는 구현도 힘들지만, 최적 교체 알고리즘을 그리 잘 근사하지도 않음.



Thrashing

- CPU의 효율성을 높이기 위해, OS는 많은 프로세스를 병렬로 실행하고, 이를 위해서 메모리에 많은 프로세스를 올림.
- 하지만 동시에 실행 중인 프로세스의 수가 많아질수록 하나의 프로세스가 할당받는 Frame의 수도 적어짐.
- Frame의 수가 줄어들게 되면 그만큼 Page Fault가 많이 발생
- 자원의 활용보다는 I/O 작업에 시간을 더욱 소비하게 됨.

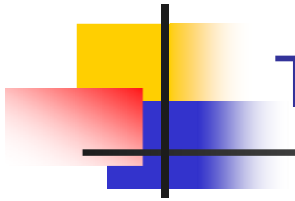
- 이렇게 되면 프로그램의 진행속도는 굉장히 느려지고 CPU의 효율성 역시 굉장히 떨어짐.



Thrashing

- 그러나 OS는 이러한 CPU 효율성의 저하를 극복하기 위해 메모리에 프로세스를 더 올림
- 이런 악순환으로 CPU 효율성은 기하급수적으로 떨어짐.

- **이러한 현상을 Thrashing이라고 함**



Thrashing

- 해결 방법
  - 물리 메모리 늘려주기
    - Belady's anomaly를 기억할 것.
      결국 page replacement를 잘 해준다는 것이 전제임
  - HDD를 보조기억장치로 쓴다면 SSD로 바꿔주기
    - effective access time 계산 때를 기억할 것.
      변경된 페이지를 보조기억장치에 갱신할 때의 시간
  - working set
  - page-fault frequency (PFF)

Working set

- 지역성(locality)을 기반으로, 가장 최근에 접근한 프레임이 이후에 또 참조될 가능성이 높다는 가정에서 나옴.
- 최근 일정 시간 동안 참조한 페이지를 물리 메모리에 상주시킴.

- **Working set**: 메모리에 상주시킬 페이지의 집합

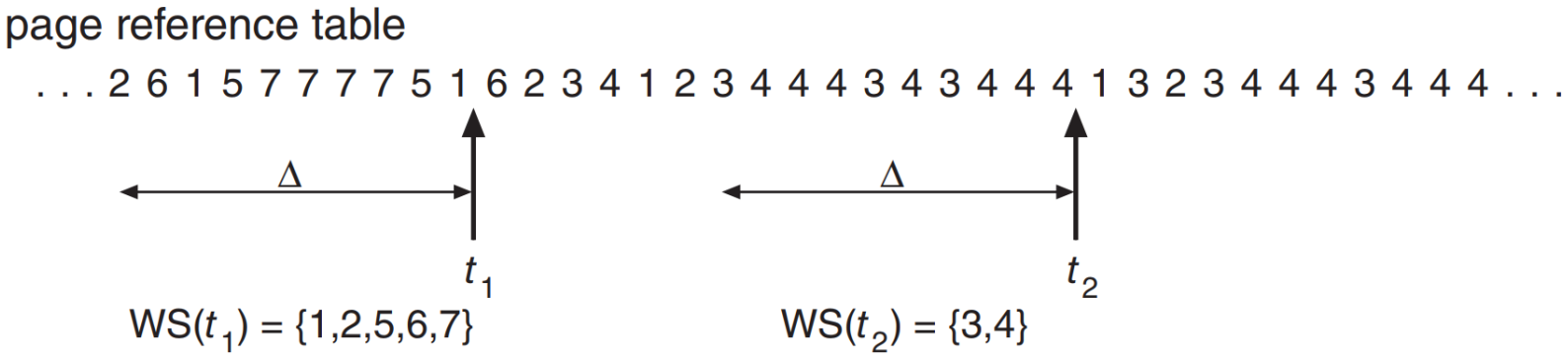


Figure 10.22 Working-set model.

working set size: 10

Page Fault Frequency (PFF)

- 프로세스가 필요로 하는 페이지 양을 동적으로 결정하는 방법
- page fault 비율의 상한선과 하한선을 설정.

- page fault 발생 비율 > 상한선
  - 프레임 할당이 너무 적음 → 프레임 할당
- page fault 발생 비율 < 하한선
  - 메모리가 낭비됨 → 할당한 프레임 회수

Page Fault Frequency (PFF)

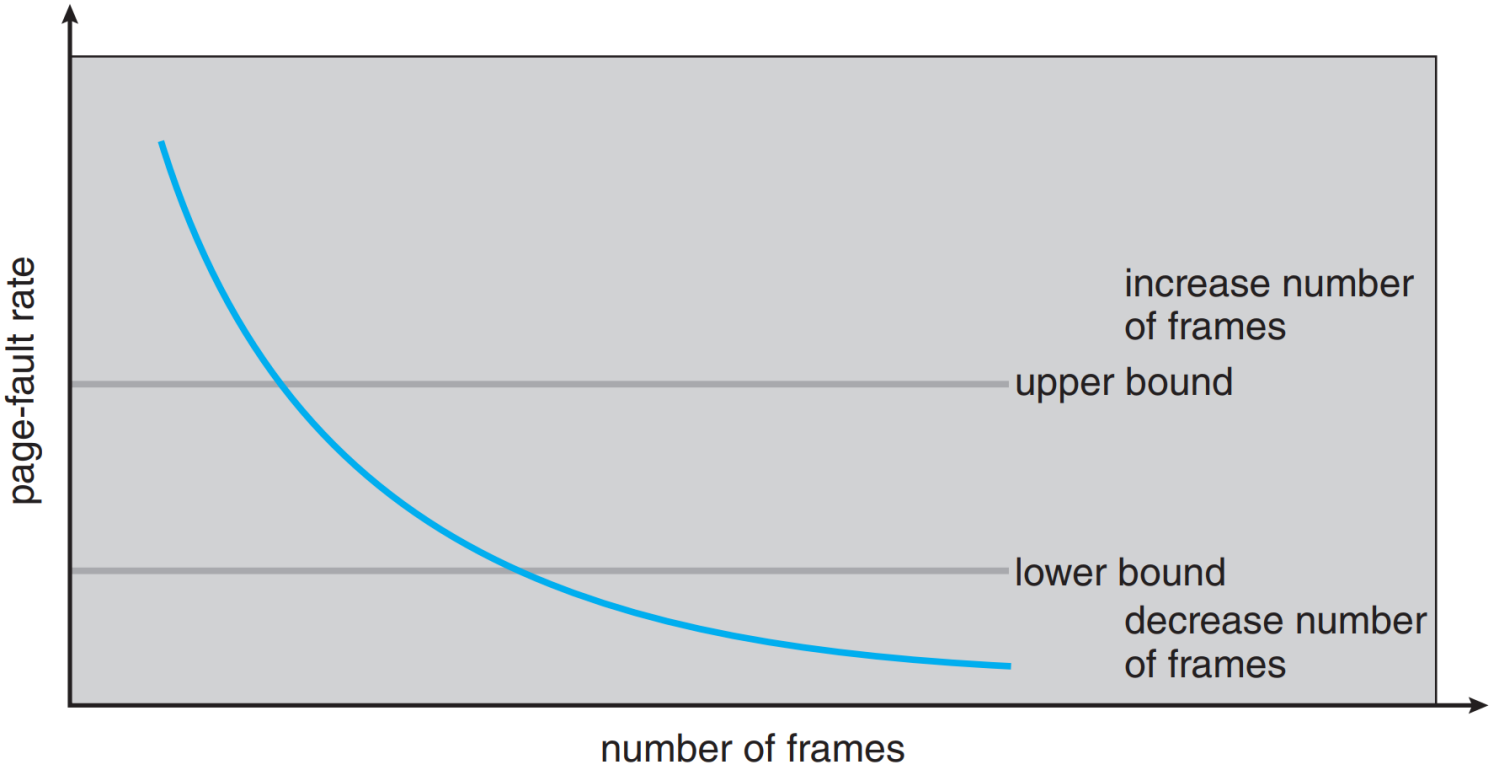


Figure 10.23 Page-fault frequency.

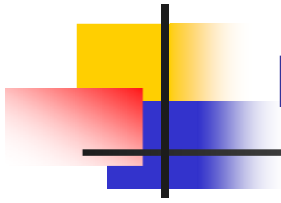
Allocating Kernel Memory

Allocating Kernel Memory

- 커널 메모리는 사용자 메모리와 다르게 취급됨
  - Memory pool에서 할당하는 정책 사용
  - **Memory pool :** 미리 메모리를 할당하여 놓은 뒤 필요에 따라 할당 및 해제하여 사용하는 기법

- 커널은 다양한 크기의 구조에 대한 메모리가 필요함

- 커널은 하드웨어와 직접 상호작용하기에
  물리적으로 연속적인 페이지 프레임을 할당 받음



Buddy System Allocator

- 2의 지수승으로 커널 프로세스에 메모리를 할당하는 방식
- 요청된 메모리 크기를 수용할 수 있는 최소 크기로 메모리를 쪼개어 할당함.
- 이런 메모리 조각을 buddy 라고 부름.

Buddy System Allocator

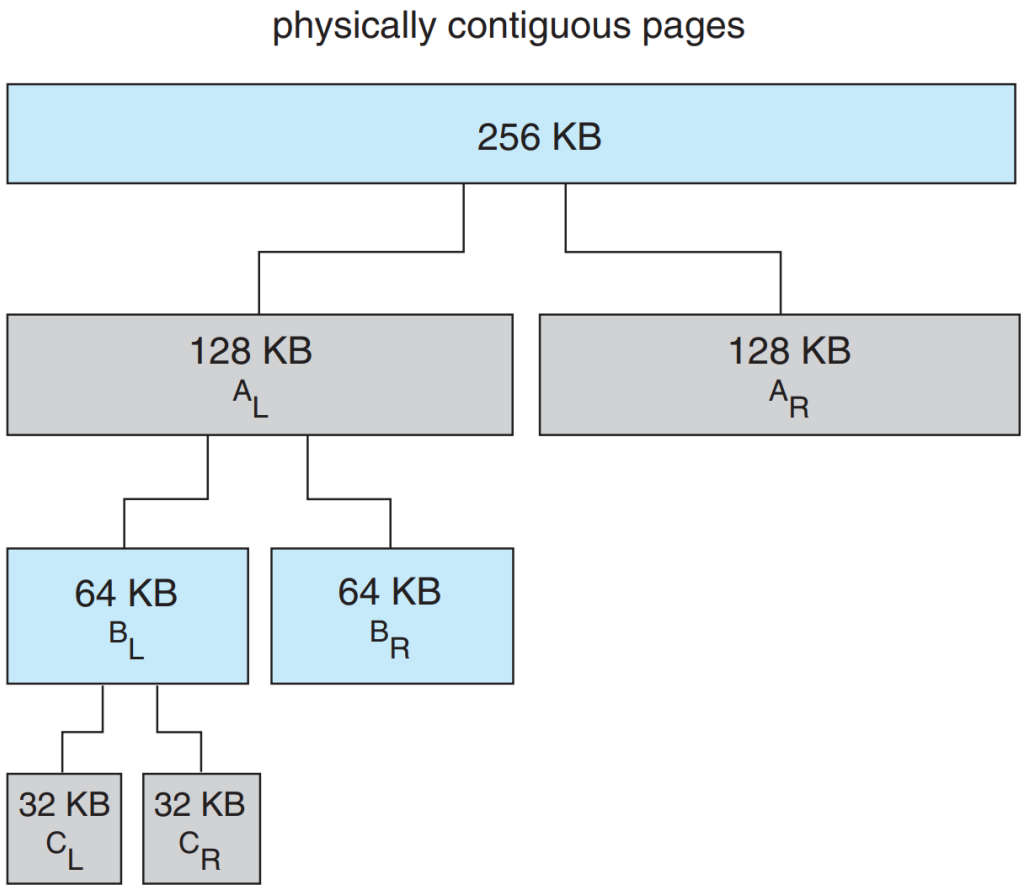
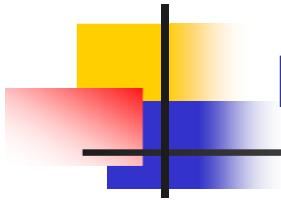


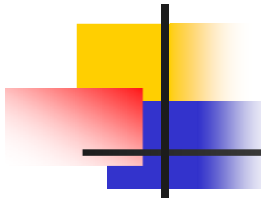
Figure 10.26 Buddy system allocation.



Buddy System Allocator

- 2의 지수승으로 커널 프로세스에 메모리를 할당하는 방식
- 요청된 메모리 크기를 수용할 수 있는 최소 크기로 메모리를 쪼개어 할당함.
- 이런 메모리 조각을 buddy 라고 부름.

- **문제점:**
  - Internal fragmentation
  - buddy 생성 시 메모리를 쪼개는 것이 오버헤드가 큼



Slab allocation

- **Slab:** 물리적으로 인접한 하나 이상의 페이지
- **Cache:** 하나 이상의 슬랩으로 구성된 Slab의 집합

- 캐시에서 할당된 slab을 모두 사용했는데 추가적인 메모리가 필요하다면, 추가로 slab을 할당해 줌.
  - 각 커널 구조(File descriptor, File 객체, Semaphore 등)마다 Cache가 존재함

- Slab states
  - Full(모든 페이지가 사용 중임)
  - Empty
  - Partial(일부 페이지만 사용 중임)

Slab allocation

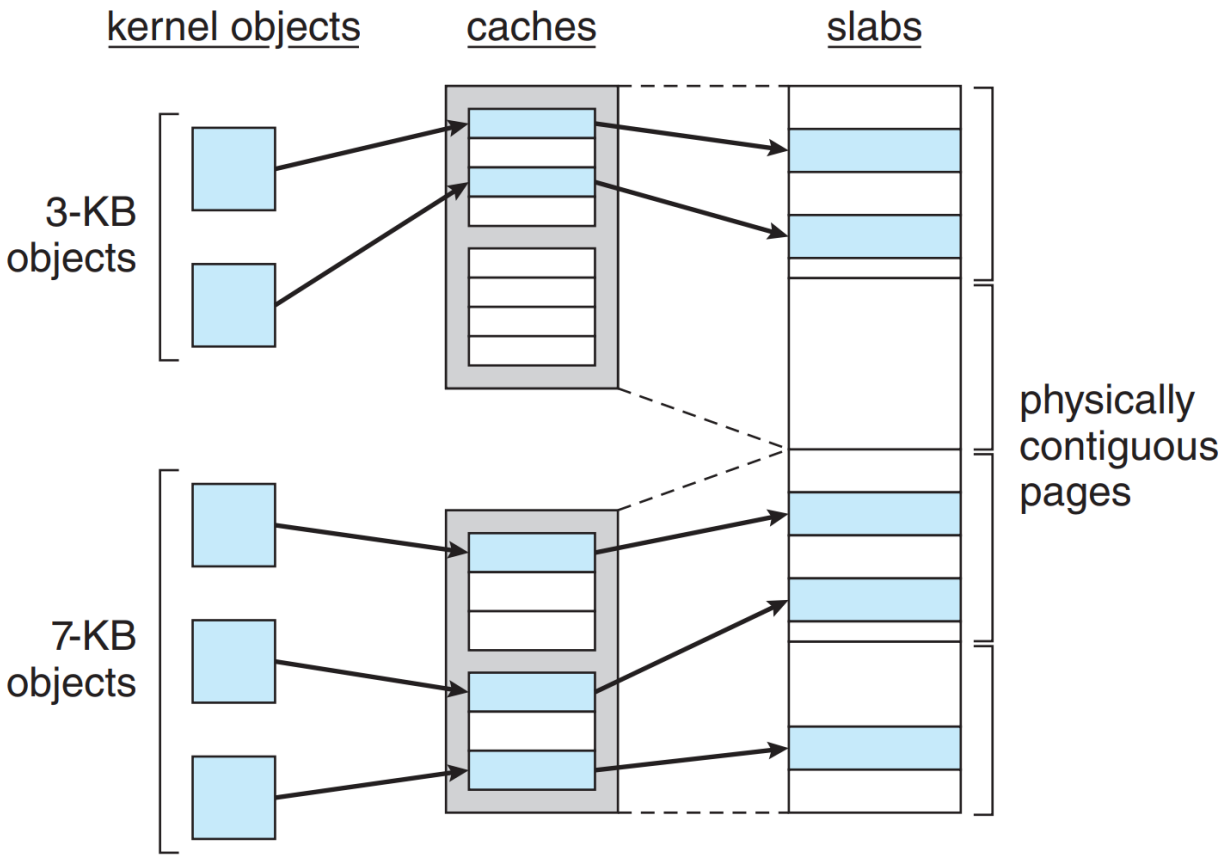


Figure 10.27 Slab allocation.

이미 정해진 크기의 slab를 할당해주는 것이므로, Buddy allocator 보다 빠름

Chapter 10
Finish