8. Memory Management (1)

2023.07.04 **황재천 신석철 임수형**

8. Memory Management

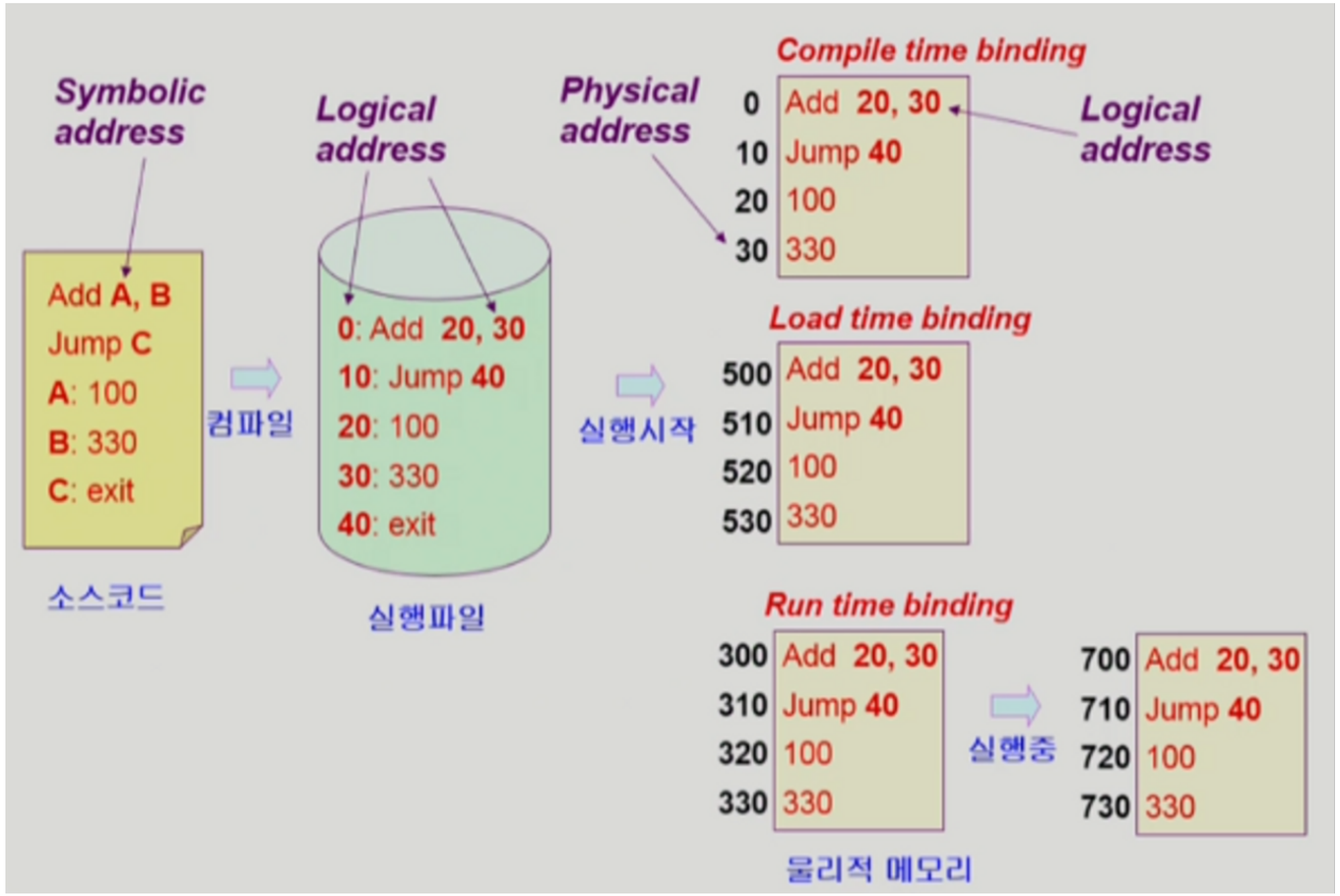
01 Logical Address *VS* Physical Address

- Logical address(=virtual address)
  - 프로세스마다 독립적으로 가지는 주소 공간
  - 각 프로세스마다 0번지부터 시작
  - CPU가 보는 주소는 *logical address*
- Physical address
  - 메모리에 실제로 올라가는 위치
- 주소 바인딩 : 주소를 결정하는 것
  - Symbolic Address → Logical Address →
    (주소 바인딩이 되는 이 시점이 언제인가?) → Physical Address

01 주소 바인딩(Address Binding)

- Compile time binding
  - 물리적 메모리 주소(Physical Address)가 컴파일 시 알려짐
  - 시작 위치 변경 시 재컴파일
- Load time binding
  - Loader의 책임하에 물리적 메모리 주소 부여
  - 컴파일러가 재배치가능코드(relocate code)를 생성한 경우 가능
- Execution time binding(=Run time binding)
  - 수행이 시작된 이후에도 프로세스의 메모리 상 위치를 옮길 수 있음
  - CPU가 주소를 참조할 때마다 binding을 점검(address mapping table)
  - 하드웨어적인 지원 필요
    - e.g., base and limit registers, MMU

01 주소 바인딩(Address Binding)



이거 일단 설명은 재첩국이 할건데 다 알면 걍 넘어갈거임.
수형햄 혹시 검토하다 발견하면 재량껏 두던가 삭제하셈.

01 Memory Management Unit(MMU)

- MMU(Memory Management Unit)
  - Logical Address를 Physical Address로 매핑해주는 하드웨어 장치
- MMU Scheme
  - 정의 : MMU가 사용하는 주소 변환 방식이나 메모리 관리 정책
  - *일반적으로* 사용자 프로세스가 CPU에서 수행되며 생성해내는 모든 주소값에 대해 base register(=relocation register)의 값을 더한다(뒤에 예제 있음)
- user problem
  - (GPT)사용자 레벨의 프로세스가 MMU의 동작에 의해 발생하는 문제를 가리킴
  - logical address만을 다룬다
  - 실제 physical address를 볼 수 없으며 알 필요가 없다

01 Dynamic Relocation

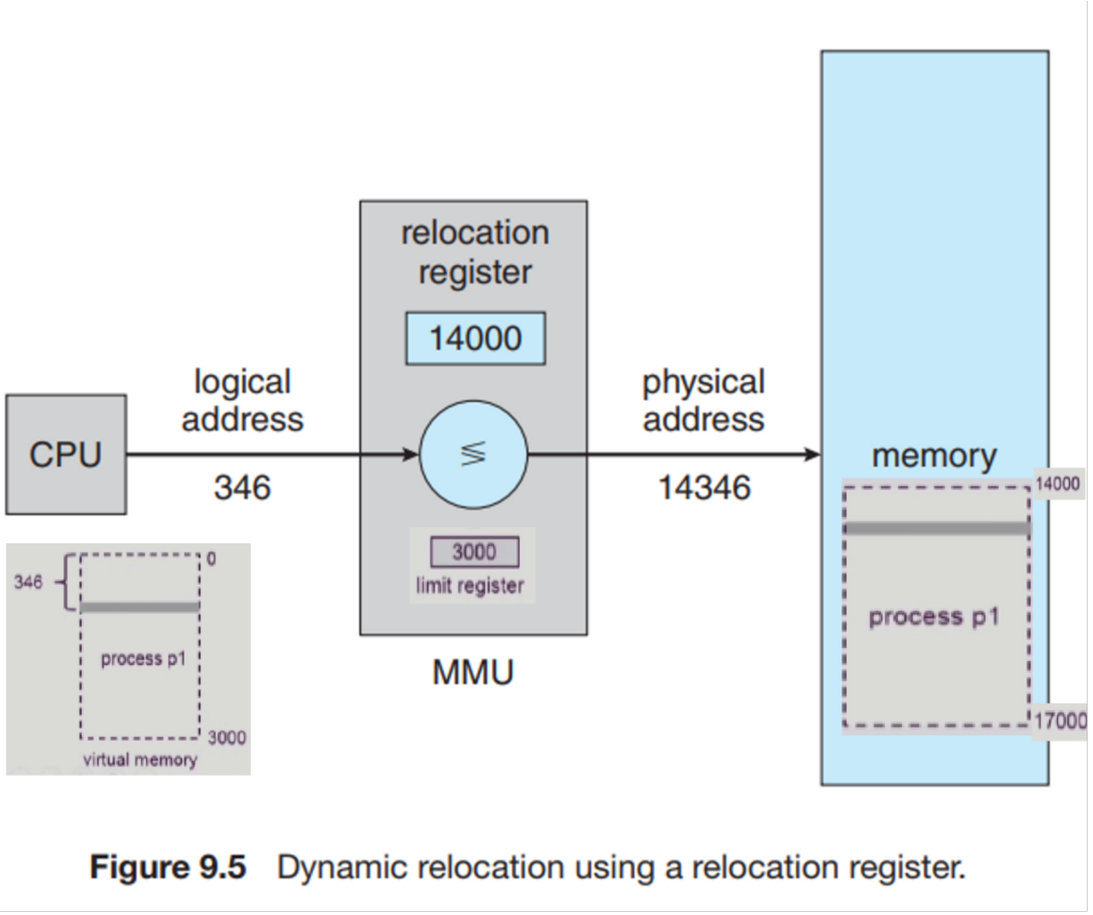


Figure 9.5 Dynamic relocation using a relocation register.

이거도 일단 설명은 재첩국이 할건데 다 알면 걍 넘어갈거임.
수형햄 혹시 검토하다 발견하면 재량껏 두던가 삭제하셈.

01 Hardware Support for Address Translation

운영체제 및 사용자 프로세스 간의 메모리 보호를 위해 사용하는 레지스터

- Relocation Resiger(=Base Register) : 접근할 수 있는 물리적 메모리 주소의 최솟값
- Limit Register : 논리적 주소의 범위

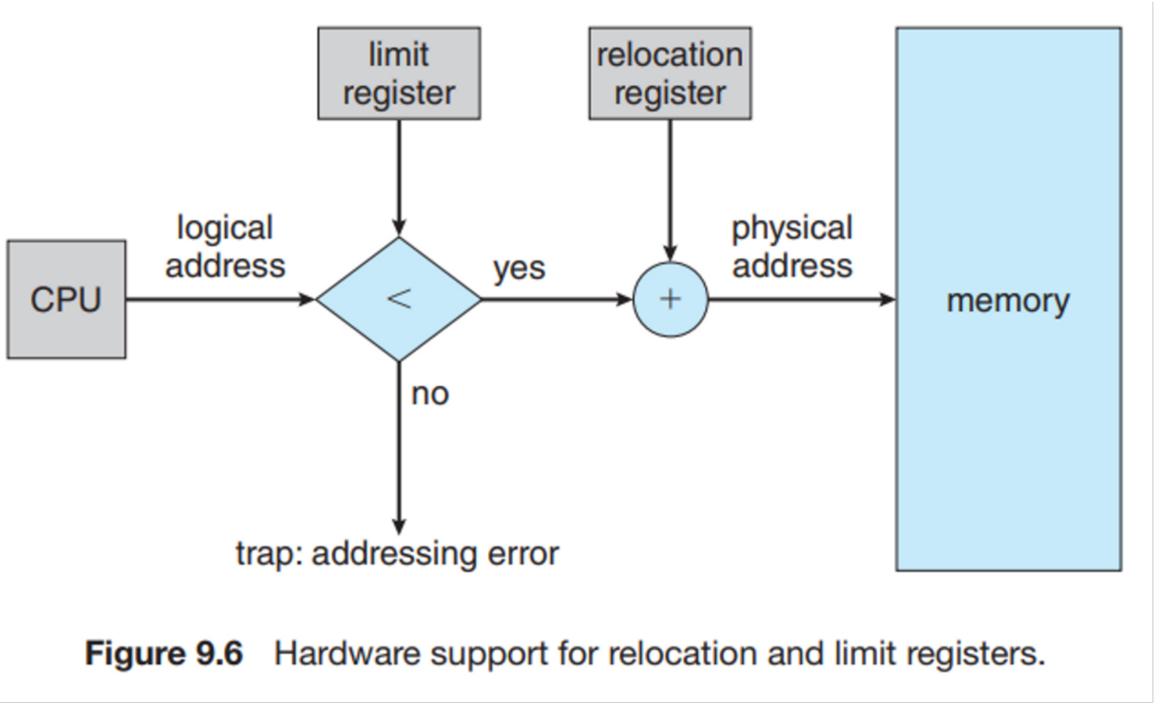


Figure 9.6 Hardware support for relocation and limit registers.

01 Some Terminologies

- Dynamic Loading
- Overlays
- Swapping
- Dynamic Linking

01 Dynamic Loading

직역하면 '메모리에 동적으로 올린다'는 뜻

- 프로세스 전체를 메모리에 미리 다 올리는 것이 아니라 해당 루틴이 불려질 때 메모리에 load하는 것
- memory utilization 향상
- 가끔 사용되는 많은 양의 코드의 경우 유용
  - 예) 오류 처리 루틴
- 운영체제의 특별한 지원 없이 프로그램 자체에서 구현 가능(OS는 라이브러리를 통해 지원 가능)

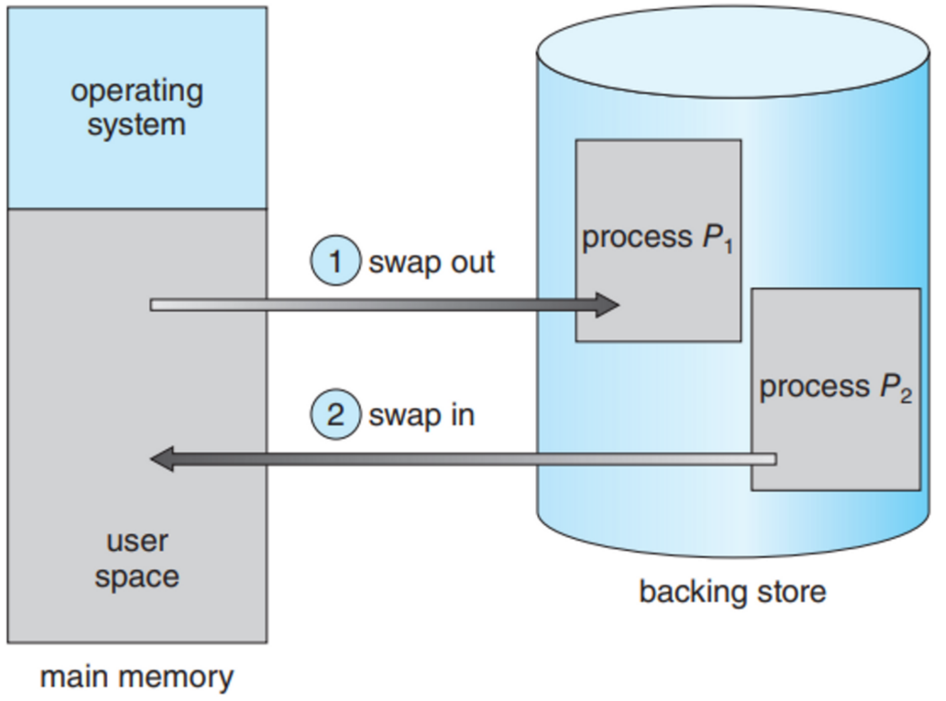
프로그래머가 dynamic loading을 명시적으로 해서 이뤄지는것이 원래 dynamic loading이지만 명시하지 않고 운영체제(CPU)가 알아서 하는것도 dynamic loading에 속한다고 함

01 Overlays

- Dynamic Loading과 비슷함, 역사적으로 다름. 옛날에는 프로세스의 크기가 메모리보다 컸고, 이 때 프로세스를 쪼개고 알고리즘적으로 구현했었음
- 메모리에 프로세스의 부분 중 실제 필요한 정보만을 올림
- 운영체제의 지원없이 사용자에 의해 구현
- 작은 공간의 메모리를 사용하던 초창기 시스템에서 수작업으로 프로그래머가 구현
  - Manual Overlay
  - 프로그래밍이 매우 복잡

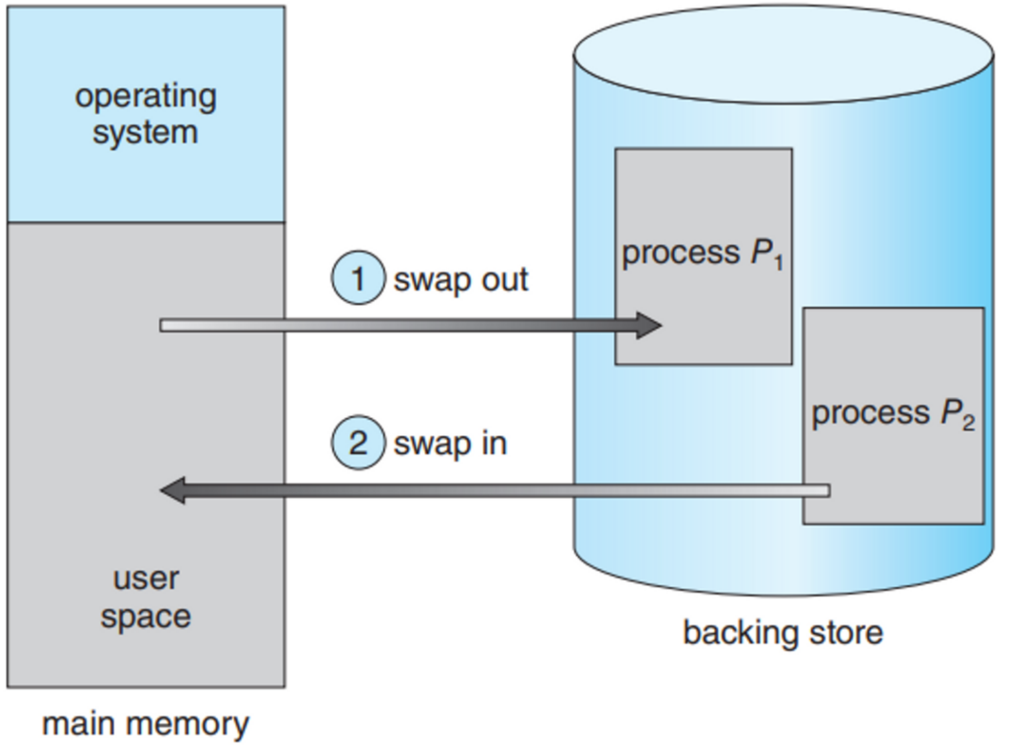
01 Swapping

- Swapping
  - 프로세스를 일시적으로 메모리에서 backing store로 쫓아내는 것
- Backing store(=swap area)
  - 디스크
    - 많은 사용자의 프로세스 이미지를 담을 만큼 충분히 빠르고 큰 저장 공간
- Swap in / Swap out



01 Swapping - Swap in / Swap out

- 일반적으로 중기 스케줄러(=swapper)에 의해 swap out 시킬 프로세스 선정
- Compile time binding혹은 Load time binding에서는 원래 메모리 위치로 swap in 해야함
- Execution time binding에서는 추후 빈 메모리 영역 아무데나 올릴 수 있음
- 데이터 양이 대부분 방대하기 때문에 swap time은 대부분 transfer time(swap되는 양에 비례하는 시간)임



Schematic View of Swapping

01 Dynamic Linking

- Linking을 실행 시간(execution time)까지 미루는 기법
- Static linking
  - 라이브러리가 프로그램의 실행 파일 코드에 포함됨
  - 실행 파일의 크기가 커짐
  - 동일한 라이브러리를 각각의 프로세스가 메모리에 올리므로 **메모리 낭비**
    - (eg. printf 함수의 라이브러리 코드)
- Dynamic linking(=Shared Library)
  - 해당 라이브러리에 접근 시 연결(link)됨
  - 라이브러리 호출 부분에 라이브러리 루틴의 위치를 찾기 위한 stub(소위 포인터)이라는 작은 코드를 둠
  - 라이브러리가 이미 메모리에 있으면 그 루틴의 주소로 가고 없으면 디스크에서 읽어옴

    → 운영체제의 도움이 필요

01 Dynamic Linking - 석철

Static linking

- 장점
  a. 실행 파일 자체가 독립적이며 이식성이 높아짐
  b. 실행 속도가 빠름
  c. 버전 충돌 문제가 발생하지 않음

- 단점
  a. 디스크 공간을 더 많이 차지
  b. 각각의 응용 프로그램이 라이브러리의 복사본을 가지게 되어 메모리 사용량이 증가
  c. 라이브러리 업데이트 시, 응용 프로그램 전체를 다시 빌드할 필요 있음

Dynamic linking

- 장점
  a. 실행 파일의 크기가 작아지며 디스크 공간을 절약
  b. 여러 응용 프로그램이 같은 라이브러리를 공유할 수 있으므로 메모리 사용량이 감소
  c. 라이브러리가 업데이트되어도 응용 프로그램을 다시 빌드할 필요가 없음

- 단점
  a. 실행 시간에 라이브러리 로딩과 링크 과정이 추가됨
  b. 응용 프로그램과 라이브러리 간의 버전 충돌 문제가 발생할 수 있음

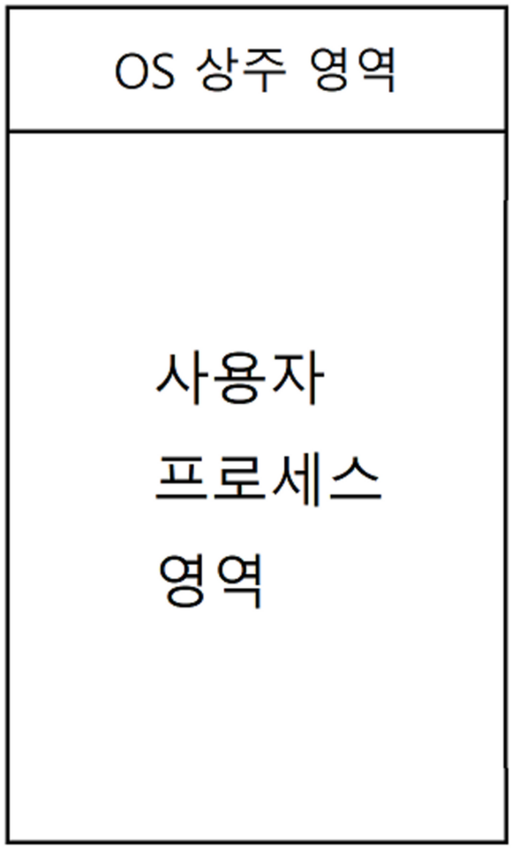
01 Dynamic Linking - 석철

- 정적 링킹은 응용 프로그램을 완전히 독립적으로 배포하고자 할 때 유용
- 이식성과 실행 속도가 중요한 경우에 사용됨
- 라이브러리의 버전 관리와 충돌 문제를 피하고자 할 때도 정적 링킹을 선택할 수 있슴

- 동적 링킹은 여러 응용 프로그램이 동일한 라이브러리를 공유하거나 라이브러리의 업데이트와 관리를 용이하게 하기 위해 사용됨.
- 메모리 사용량을 줄이고, 라이브러리의 관리 및 버전 업데이트를 효율적으로 수행가능

정리
- 정적 링킹(Static linking)은 응용 프로그램의 크기가 작고 이식성과 실행 속도가 중요한 경우에 유용
- 동적 링킹(Dynamic Linking)은 라이브러리의 공유와 관리가 필요한 경우에 적합

01 Allocation of Physical Memory



- 메모리는 일반적으로 두 영역으로 나누어 사용

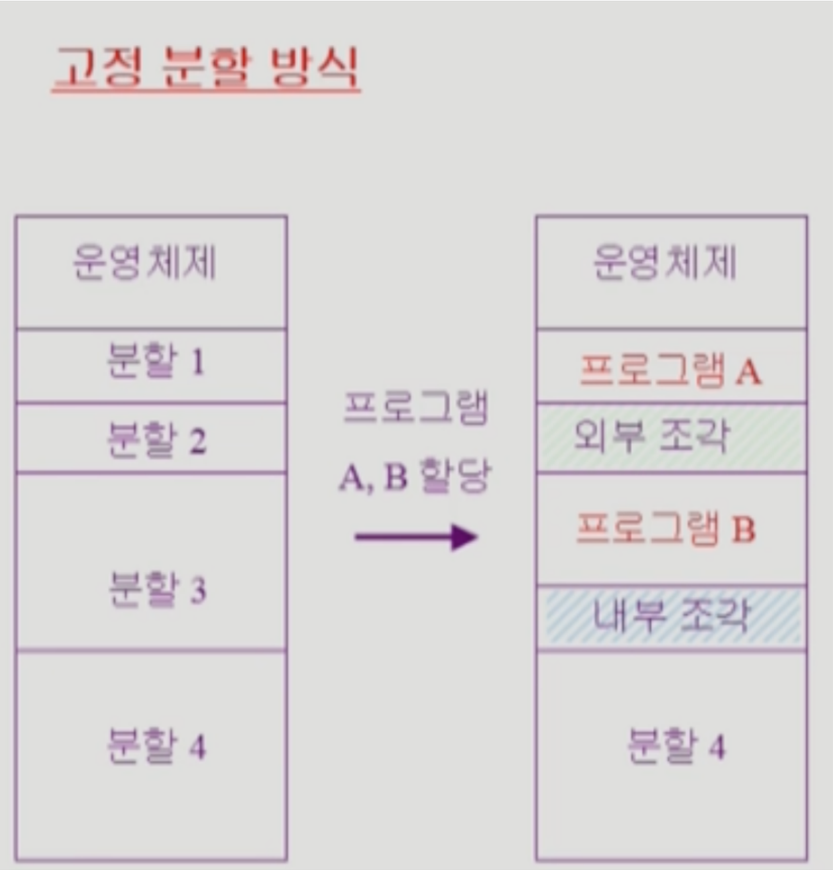
< **OS 상주 영역** > : 높은 영역_ 커널 코드

< **사용자 프로세스 영역** > : 낮은 영역_ 사용자 코드

< 사용자 프로세스 영역 >의 할당 방법에는 두 가지가 있음

1. Contiguous allocation(연속 할당)
   a. 고정 분할 방식
   b. 가변 분할 방식
2. Noncontiguous allocation(불연속 할당)
   a. Paging
   b. Segmentation

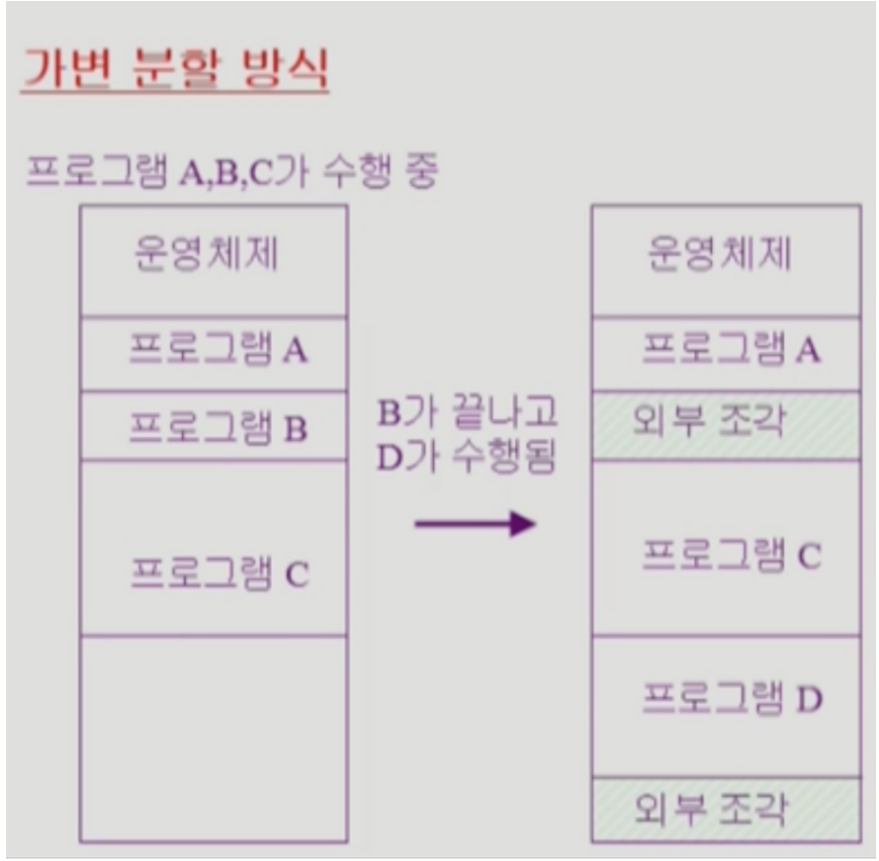
01 Contiguous allocation (연속 할당) _ 고정 분할 방식



고정분할(Fixed partition) 방식

- 사용자 프로그램이 들어갈 메모리 영역을 미리 나눠놓음 (융통성 X)
- 내부 조각, 외부 조각 발생 가능
- 메모리의 낭비가 발생할 수 있다

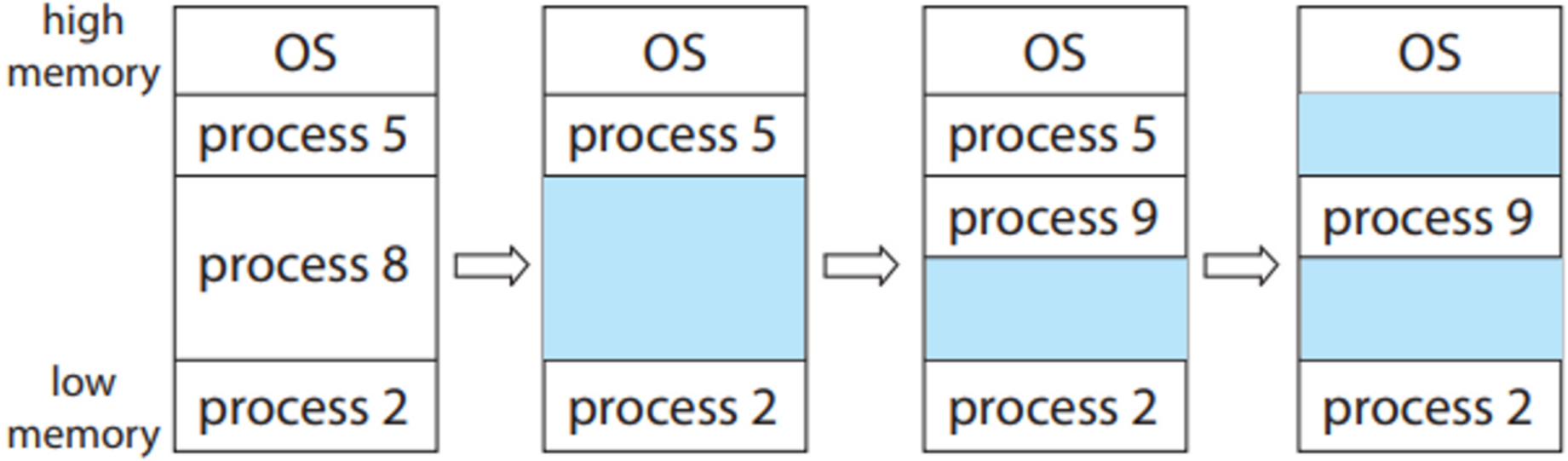
01 Contiguous allocation (연속 할당) _ 가변 분할 방식



가변분할(Variable partition) 방식

- 사용자 프로그램이 들어갈 메모리 영역을 미리 나눠놓지 않고 메모리에 차곡차곡 올리는 것
- 프로그램의 크기를 고려해서 할당
- 분할의 크기와 개수가 동적으로 변함
- 프로그램 실행 종료 과정에서 외부 조각 발생

01 hole _ 가용 메모리 공간



1. 다양한 크기의 hole들이 메모리 여러 곳에 흩어져 있음
2. 프로세스가 도착하면 수용가능한 hole을 할당
3. 운영체제는 할당공간과 가용공간(hole)을 check하고 있음

<가변 분할 방식에서 프로그램 사이지를 만족하는 적절한 hole을 찾는 방법을 뒤에서 살펴볼 것>

01 Dynamic Storage-Allocation Problem

가변 분할 방식에서 size n인 요청을 만족하는 가장 적절한 hole을 찾는 문제 (n: 프로그램 크기)

(1) Frist-fit

Size가 n 이상인 것 중 처음으로 발견되는 hole에 할당

(2) Best-fit

Size가 n 이상인 가장 작은 hole을 찾아서 할당

hole들의 리스트가 크기순 정렬되어있지 않으면 모든 hole의 리스트를 탐색해야 함

많은 수의 아주 작은 hole들이 생성됨

(3) Worst-fit

가장 큰 hole에 할당

모든 리스트 탐색해야 하고 상대적으로 아주 큰 hole들이 생성됨

실험적으로 First-fit, Best-fit이 속도와 공간 이용률(memory utilization) 측면에서 효과적

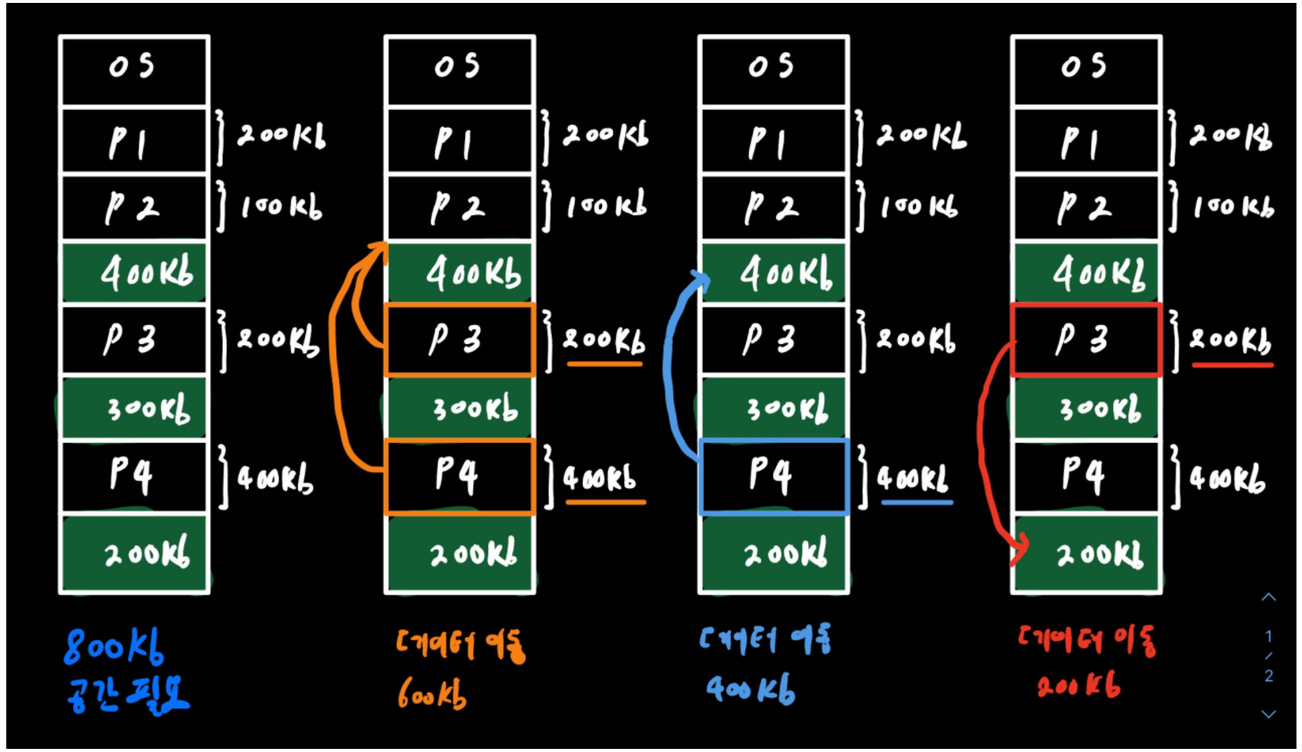
01 Compaction (압축)

앞서 보았던 방식을 진행해도 중간중간에 여전히 작은 hole들이 생길 수 있음 -> compaction 사용

- external fragmentation 문제를 해결하기 위해 외부 조각으로 인해 생기는 hole들을 한군데로 밀어서 큰 hole을 만드는 것
- 사용 중인 메모리 영역을 한군데로 몰고 hole들을 다른 한 곳으로 몰아서 큰 block을 만드는 것이며 매우 비용이 많이 들고 최소한의 메모리 이동으로 구현하는 것이 복잡함
- 프로세스의 주소가 실행 시간에 동적으로 재배치 가능한 경우에만 수행될 수 있음

01 Compaction (압축) 과정

Q. 재천 : Compaction의 과정에 대해 가볍게 부탁드립니다. 깊이가진 말고. 교수님이 매우 복잡하다고 경고했음.



프로세스의 의존성

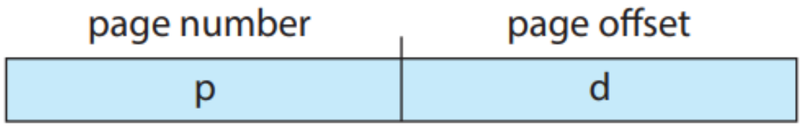
실행 중인 프로세스들은 서로의 데이터나 상태에 의존할 수 있기 때문에 프로세스를 이동시킬 때에는 올바른 순서로 진행시켜야 문제가 발생하지 않음

성능 영향

실행 중인 프로세스들을 이동시키는 작업이므로 추가적인 시간과 리소스가 필요.. 큰 시스템에서는 수천 개 이상의 프로세스를 처리해야 할 수도 있기 때문에 이로 인한 성능 저하가 발생할 수 있음

04 Noncontiguous Allocation - Paging

물리 메모리를 Frame이라는 고정 크기의 block으로 자름.
논리 메모리를 동일한 크기의 pages로 쪼개고, 프로그램
실행시 backing store에서 가용 메모리 frame에 로드.



논리 주소는 page number, page offset로 나뉘어지고,
page와 frame이 매칭되어있는 page table을 통해 주소 변환이 이루어진다.
page table은 각 프로세스마다 존재하고, memory(kernel)상에 존재한다.
-> 원하는 데이터에 접근하기 위해서는 메모리를 두 번 접근해야 함.
+ 메모리에 저장하기 때문에 메모리 공간 필요.

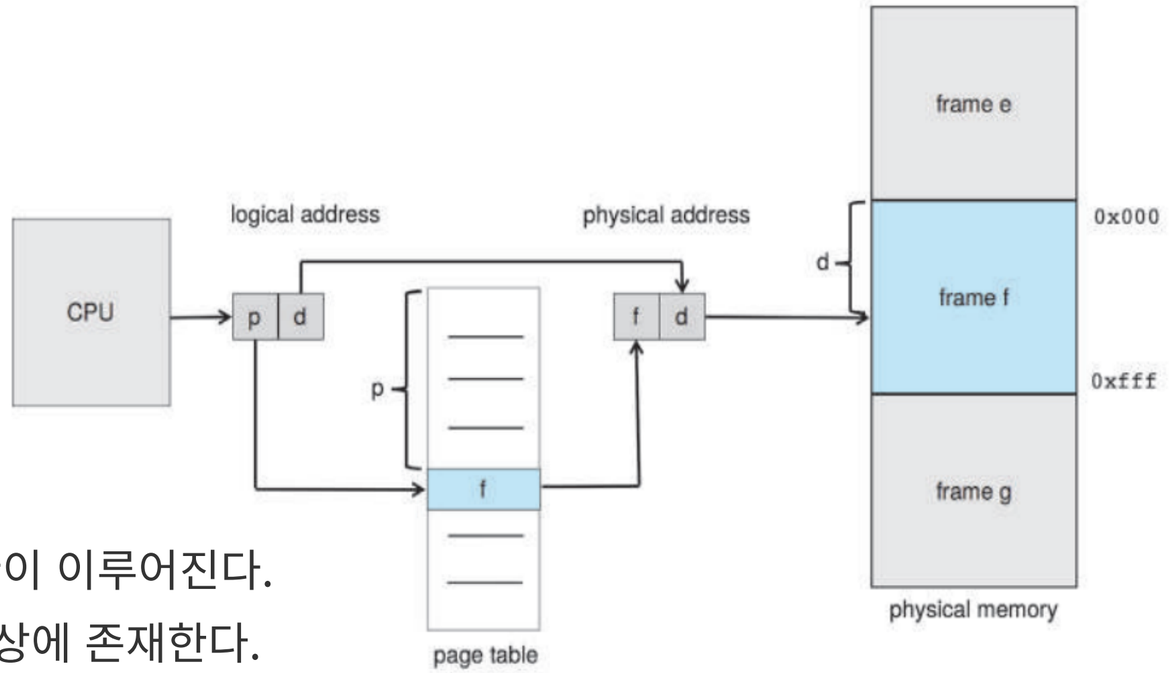


Figure 9.8 Paging hardware.

04 Paging with TLB

Translaction look-aside buffer
page table을 탐색하기 위한 전용 HW

associative한 register로 parellel한 탐색이 가능하다. 검색이 매우 빠름

비싸기 때문에 작은 크기의 TLB 사용, 메모리에 page table 전부 담아둠.

-> 최근에 사용된 page에 대한 entry만 tlb에 저장. (Locality)

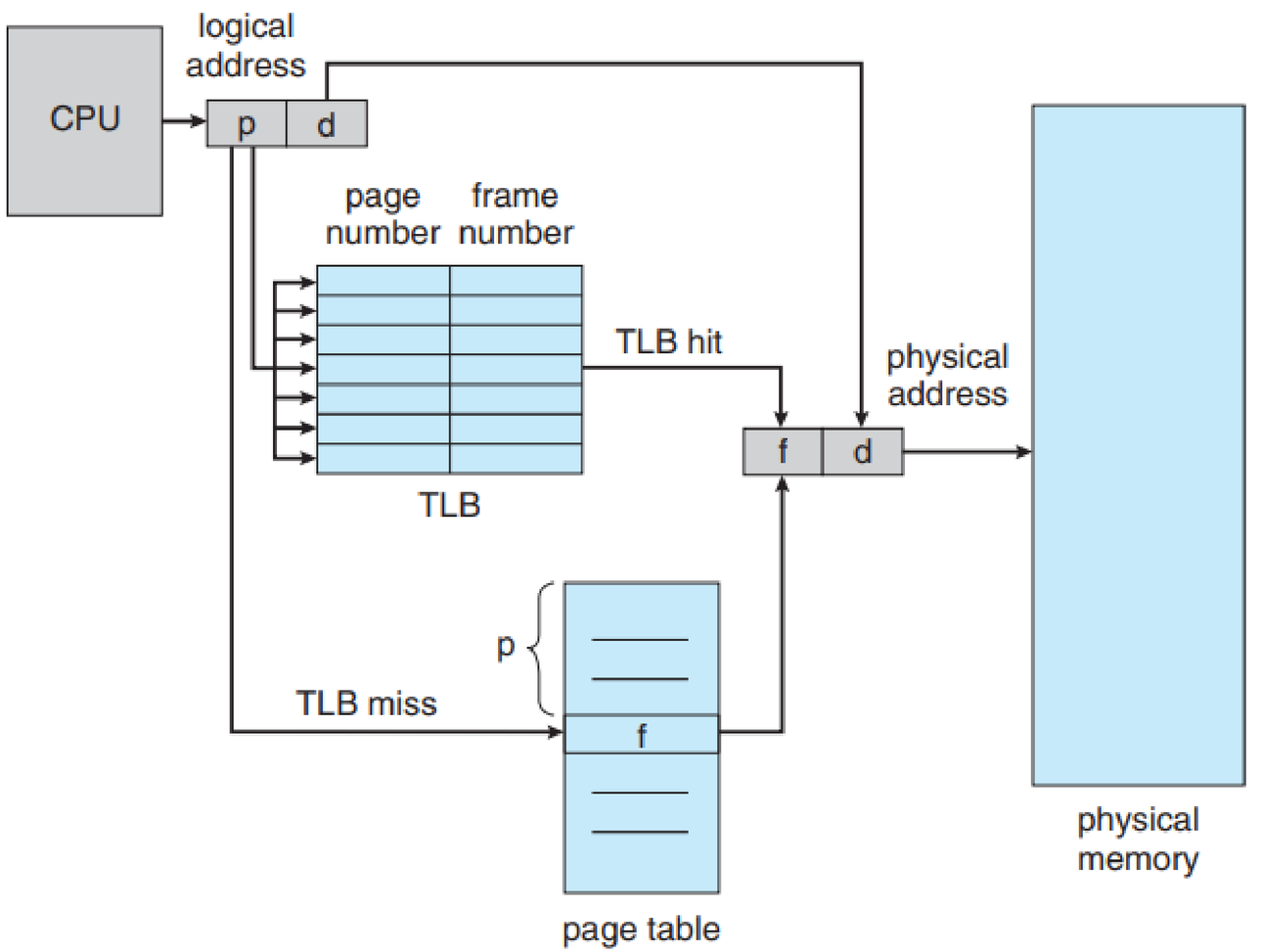
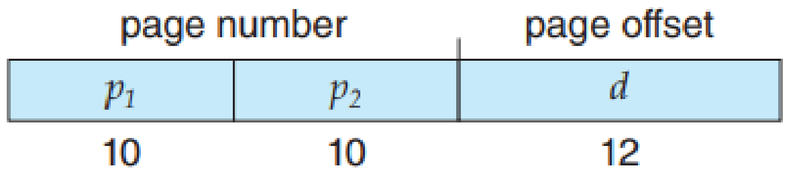


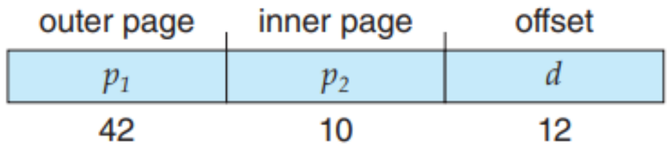
Figure 9.12 Paging hardware with TLB.

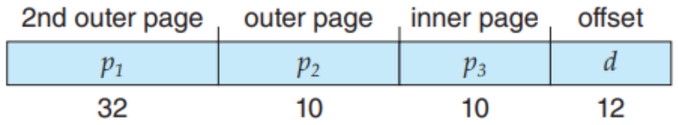
04 Two-Level page system

현대 컴퓨터 시스템은 큰 논리 주소 공간을 사용

32bit에서 page size가 4k면, entry는 백만개.
거기서 entry가 4byte로 구성되어있다면, page table은 4MB.
-> 메모리에 일부만 올려두자. 역시 TLB로 관리하게됨.
다만 TLB miss가 2개로 늘어남. TLB hit가 중요해짐.







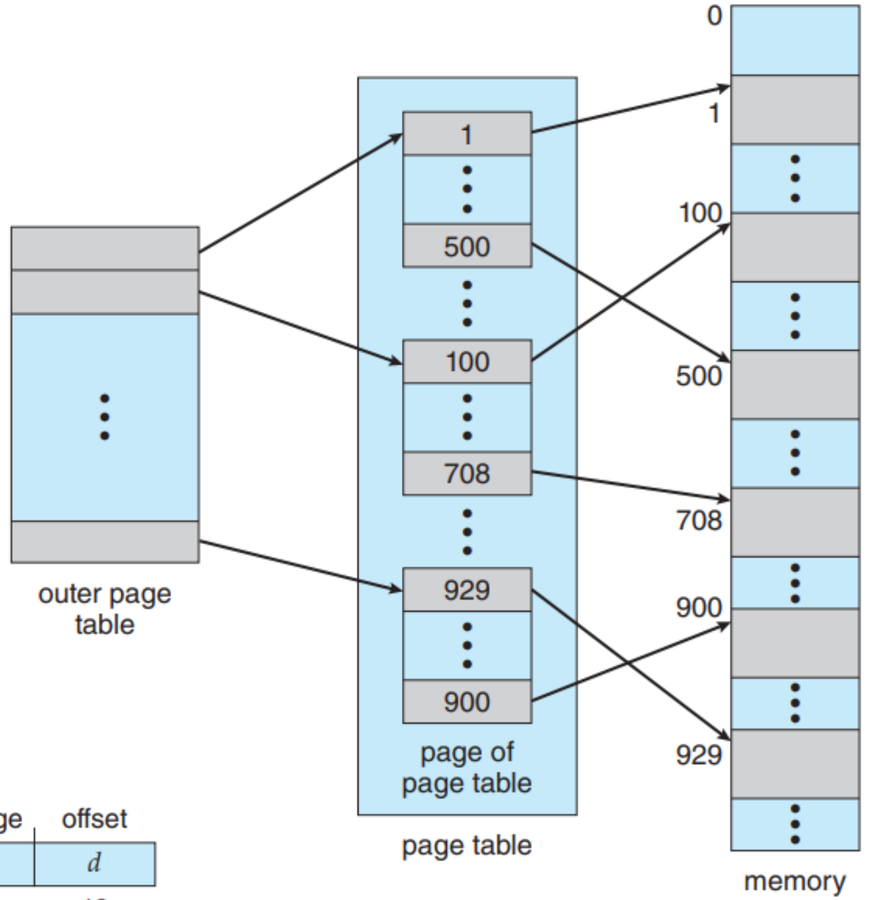


Figure 9.15 A two-level page-table scheme.