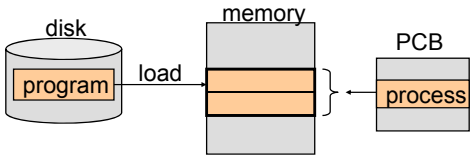
8장. 메모리 관리

목표

- 메모리 하드웨어를 구성하는 다양한 방법 소개
- 프로세스에게 메모리를 할당하는 다양한 기법 설명
- 현대 컴퓨터 시스템에서 paging 동작 방법 논의

8.1 배경 지식

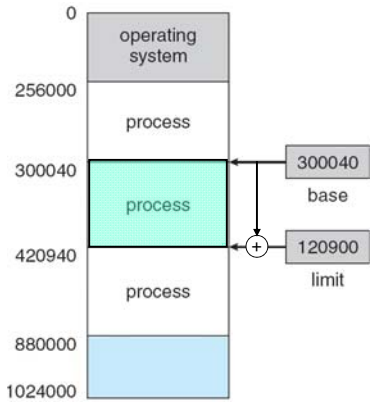
- 프로그램은 디스크에서 메모리로 적재된 후 실행됨 → 프로세스



- CPU가 직접 접근 가능한 기억장치 - CPU register, main memory
  - **CPU registers** – 1 CPU clock에 접근 가능
  - **main memory** – 접근에 여러 CPU cycle이 소요 → processor stall 발생
    - **cache 메모리**를 CPU와 main memory 사이에 추가하여 processor stall 빈도를 감소시킴
- 메모리 보호(Protection)
  - 올바른 동작을 보장하기 위해서 메모리 보호가 필요함
  - 메모리 보호를 위해 **하드웨어 지원**이 필요, 운영체제는 이를 관리함

메모리 보호 – Base와 Limit Registers 사용

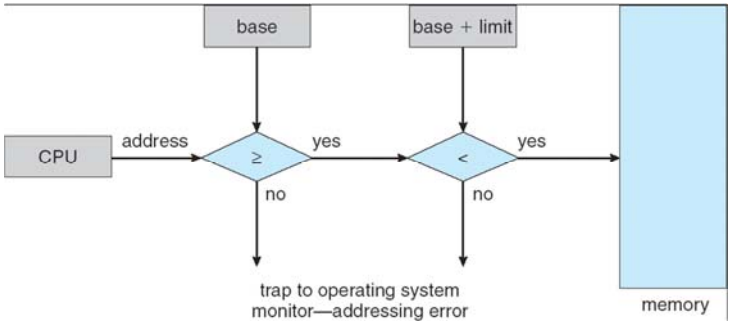
- 메모리 보호 방법 – 프로세스에게 독립된 주소 공간을 제공
- **base** register와 **limit** register를 사용한 방법
  - 프로세스의 합법적인(legal) 주소공간 : base ≤ address < base+limit



프로세스에게 독립적인 주소 공간을 제공하기 위해서는 주소 공간이 서로 중첩되지 않게 모든 프로세스의 base/limit 를 설정해야 함

메모리 보호 - Base와 Limit Registers를 사용한 주소 검사

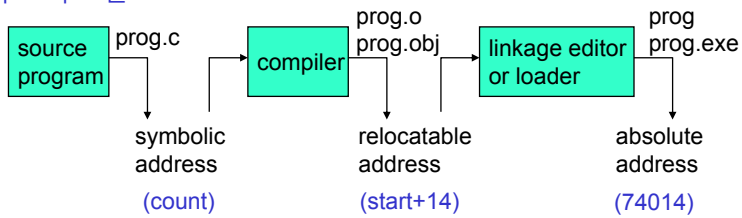
- 하드웨어를 통한 주소 검사
  - 합법적인 아닌 주소에 대해서는 trap을 발생시켜서 메모리를 보호함



- base와 limit registers는 특권 명령어만 값을 적재할 수 있음
  - 운영체제에 의해서만 적재 가능
- 운영체제는 메모리 영역접근에 제약이 없음
  - 사용자 메모리 영역, 운영체제 메모리 영역 모두 접근 가능

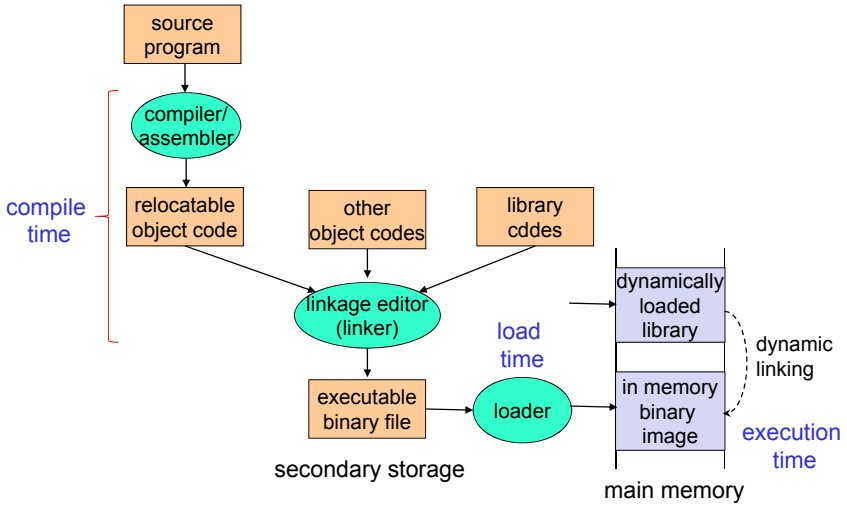
주소 바인딩(Address Binding)

- 주소의 표현



- 주소 바인딩(Address Binding)
  - 한 주소 공간에서 다른 주소 공간으로 맵핑하는 것
  - 프로그램의 **명령어/데이터 주소**를 **물리적 메모리 주소**로 "바인딩"
    (logical address) (physical address)
- 링커(linker, linkage editor)와 로더(loader)
  - **linker** : 모듈을 함께 묶어서 실행파일을 생성함
  - **loader** : 프로그램 실행을 위해서 프로그램의 전부 또는 일부를 메모리에 적재함

사용자 프로그램의 다단계 처리 과정

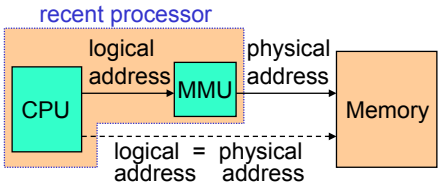


주소 바인딩 시점

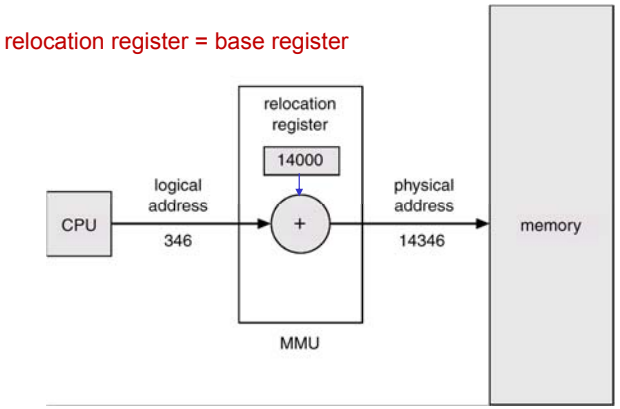
- **Compile time**:
  - 컴파일 시점에서 프로그램이 적재될 메모리 위치를 미리 안다면, 컴파일러는 **absolute code**를 생성할 수 있음
  - 적재될 시작 위치가 변경된다면, 코드를 다시 컴파일해야 함
- **Load time**:
  - 컴파일 시점에서 프로그램이 적재될 위치를 알지 못하면, 컴파일러는 **relocatable code** 생성해야 함
  - 주소 바인딩은 load time에 수행됨 - 메모리 위치는 적재 시에 결정됨
- **Execution time**:
  - 프로세스가 실행 도중에 메모리 위치를 이동할 수 있다면, 주소 바인딩은 load time에 할 수 없고, run time에 수행됨
  - 주소 맵핑을 위한 **하드웨어 지원**이 필요 ➔ MMU
  - 대부분의 운영체제가 이 방법을 사용
- 정적 바인딩(Static Binding)과 동적 바인딩(Dynamic Binding)
  - Compile time, Load time binding → static binding
  - Execution time binding → dynamic binding

논리 주소 공간과 물리 주소 공간

- 논리 주소(logical address)와 물리 주소(physical address)
  - **logical address**(virtual address): 프로그램이 사용하는 주소
  - **physical address**: 물리적 메모리 장치가 사용하는 주소
- 메모리 관리 장치(MMU)
  - 논리 주소를 물리 주소로 맵핑하는 작업을 실행시간에 수행
  - 프로그램은 논리 주소만 사용하며, 실제 물리적 주소를 알지 못함
- 주소 바인딩 시점과 논리/물리 주소
  - compile-time과 load-time 바인딩 → logical addr = physical addr
  - execution-time 바인딩 → logical addr ≠ physical addr



재배치(relocation) 레지스터를 사용한 동적 재배치



- (예) Intel 80x86 family:
  - 4 또는 6 relocation registers (CS, DS, ES, SS / FS, GS)

동적 적재(Dynamic Loading)

- 동적 적재(Dynamic loading)
  - 실제 호출되기 전에는 각 루틴은 메모리로 적재되지 않음
  - 호출될 때에 메모리에 적재되어 있지 않으면 relocatable linking loader를 호출하여 요구하는 루틴을 메모리에 적재한 후 실행
- 장점
  - 향상된 메모리 공간 활용; 사용하지 않는 루틴은 적재되지 않음
  - 많은 양의 코드가 덜 빈번히 사용되는 경우에 유용 (예) 오류 처리 코드
- 동적 적재는 운영체제의 특별한 지원을 필요로 하지 않음
  - 프로그래머가 동적 적재를 위한 설계를 책임짐
  - 운영체제는 동적 적재를 지원하는 library 루틴을 제공할 수 있음

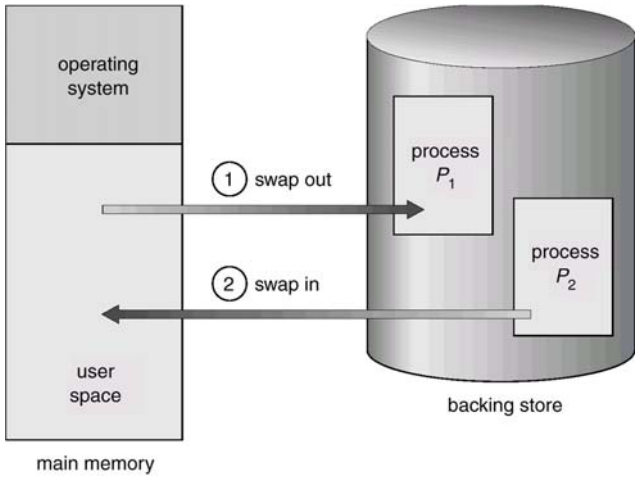
동적 링크와 공유 라이브러리

- 정적 링크(Static linking)와 동적 링크(Dynamic linking)
  - 정적 링크 – linkage editor에 의해 library가 실행 이미지에 링크됨
  - 동적 링크 – library의 링크가 실행시간까지 미루어짐
- Stub
  - library를 어떻게 찾는 지를 알려주는 작은 코드
    - library 루틴이 메모리에 존재하지 않으면 디스크에서 적재함
  - library 함수를 호출하면 처음에는 stub코드가 호출됨
  - stub는 실제 library 함수의 주소로 대체되어,
    다음 호출 시에는 library 함수를 직접 호출하여 실행
- Shared library: 동적링크 라이브러리(dynamic linked library:DLL)
  - 공유 라이브러리를 사용하는 모든 프로세스는 한 개의 library 코드만 공유하여 사용
  - library 업데이트가 용이함 – 버전 정보 사용하여 구분
- 동적 적재와 다르게 동적 연결은 운영체제의 도움이 필요
  - 운영체제가 라이브러리 루틴이 메모리에 있는 지 여부를 검사하며
  - 여러 프로세스가 라이브러리를 공유할 수 있게 해줌.

8.2 스와핑(Swapping)

- 스와핑(Swapping)
  - 실행을 계속할 수 없는(예: 입출력 완료 대기) 프로세스를 메모리에서 일시적으로 **예비 저장장치**(backing store)로 내보내고(**swap out**), 실행을 계속할 수 있는 프로세스를 예비 저장장치에서 메모리로 불러오기(**swap in**)하여 실행을 재개할 수 있게 하는 것 (다음 쪽 그림)
- 예비 저장장치(Backing store)
  - 모든 프로세스의 **메모리 image**의 복제본을 저장할 수 있을 정도의 저장 용량을 가진 **빠른** 디스크를 사용 -
  - 빠른 접근을 위해서 저장된 메모리 image에 대해 직접 접근이 가능해야 함
- 스와핑의 효과
  - "모든 프로세스의 물리적 주소 공간의 합 > 물리적 메모리 크기" 인 경우에도 모든 프로세스가 동시에 실행되는 것이 가능

Schematic View of Swapping

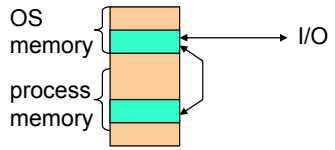


스와핑과 Context switching 시간

- 스와핑 동작
  1. CPU scheduler가 ready queue에서 다음 프로세스 결정하고 dispatcher를 호출
  2. Dispatcher는 다음 프로세스가 메모리에 있는 지 또는 예비 저장장치에 있는 지 확인
     - 메모리에 없고, 여유 메모리 영역이 없다면, 메모리에 있는 한 프로세스를 내보내고(swap out), 원하는 프로세스를 메모리로 불러오기(swap in)를 수행 → swap out 후 swap in 수행
     - 메모리에 없지만 여유 메모리 영역이 있다면, 원하는 프로세스를 메모리로 불러오기(swap in)를 함 → swap in만 수행
  3. Dispatcher는 context switching을 수행
- Context switch 시간
  - **context switch time > swap in time + swap out time**
  - swap 시간의 대부분이 디스크 전송 시간(transfer time)임
    - 전송 시간은 스왑되는 메모리 양에 비례함
  - 시간 감소 방안: 실제로 사용되는 부분만 불러오기 함(swap in)
    - 메모리 요구사항 변화 발생 시에 시스템에 알려주어야 함

스와핑과 입출력

- 스와핑되는(swap out) 프로세스는 완전히 idle 상태이어야 함
- 진행 중인(pending) I/O를 갖는 프로세스
  - 스와핑하지 않음
  - 더블 버퍼링(double buffering)
    - 운영체제 buffer를 통하여 입출력 연산 수행
    - 프로세스가 swap in 될 때, 운영체제 버퍼와 프로세스 메모리 간에 입출력 데이터 전송



여러 가지 Swapping

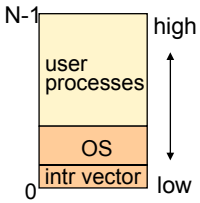
- 기본 스와핑(Standard Swapping)
  - RR 스케줄링 - quantum이 종료된 프로세스를 swap out, 여유 메모리를 다른 프로세스에게 할당
  - 우선순위 스케줄링 – 낮은 우선순위 프로세스를 swap out, 높은 우선순위 프로세스들에게 할당
  - 현대 운영체제에서는 사용하지 않음
- 변형된(modified) 스와핑
  - 많은 시스템에서 널리 사용 – UNIX, Linux, Windows
  - 평상시에는 Swapping 비활성화
  - 여유 메모리 양이 일정 기준(threshold)이하가 되면 Swapping 시작
  - 여유 메모리 양이 증가하면 Swapping은 다시 정지됨
- 또 다른 변형된 스와핑
  - 프로세스의 일부만 스와핑 → swap 시간 감소

모바일 시스템에서의 Swapping

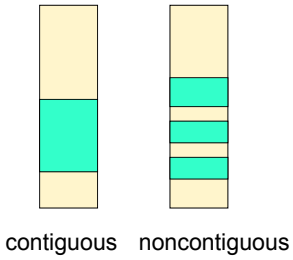
- 모바일 시스템은 대개 swapping을 지원하지 않음
  - 영구 저장장치로 **하드 디스크** 대신에 **플래시 메모리**를 사용
  - 스와핑을 하지 않는 이유 - 플래시 메모리는 다음 특성을 가짐
    - (1) 공간 제약 (2) 허용된 쓰기 횟수의 제한
- Swapping 대신 사용하는 방법
  - iOS는 여유 메모리가 threshold보다 작아지면 응용 프로그램에게 메모리를 반환하도록 요청 → 읽기 전용 데이터만 제거
    → 충분한 메모리를 반환하지 않은 응용은 프로세스를 종료시킴
  - Android는 여유 메모리가 충분하지 않으면 응용 프로그램의 상태를 플래시 메모리에 저장하고 프로세스를 종료시킴 → 빠른 재시작 가능

8.3 연속(Contiguous) 메모리 할당

- 주 메모리는 두 부분으로 구분
  - 상주 운영체제 용
    - 인터럽트 벡터를 포함하는 위치 사용
    - 대개 하위 메모리(low memory)에 위치
  - 사용자 프로세스 용
    - 상위 메모리(high memory)에 위치



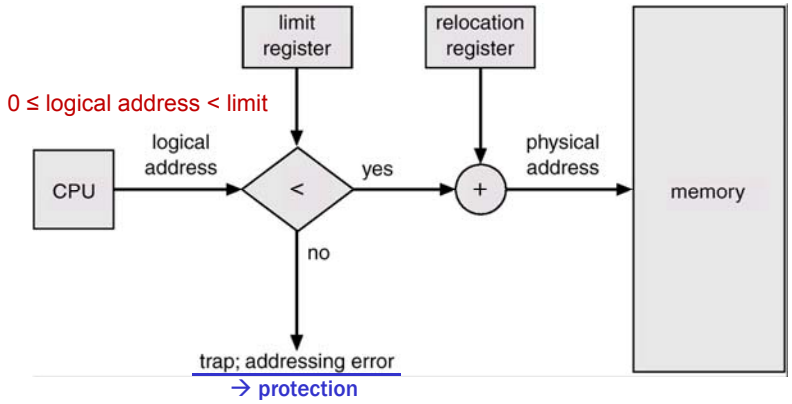
- 프로세스 메모리 할당 방법
  - 연속(contiguous) 메모리 할당
  - 비연속(Noncontiguous) 메모리 할당



메모리 맵핑과 보호

- **재배치 레지스터**(Relocation register) 방법
  - 메모리 관리 장치(MMU: memory management unit)는 다음 두 레지스터를 사용하여 메모리 주소 맵핑
    - 재배치(Relocation) register: 프로세스가 배치된 물리적 주소의 시작 주소 저장
    - 상한(Limit) register: 논리 주소(0부터 시작)의 범위를 저장
- 메모리 보호
  - 메모리 주소 범위를 하드웨어를 사용하여 합법적인 범위로 제한하여 이 프로세스가 다른 프로세스와 운영체제를 수정하는 것을 방지함
  - (다음 슬라이드)
- MMU는 동적으로 논리 주소를 물리 주소로 맵핑
  - 운영체제와 프로세스 크기를 동적으로 변경하는 효과적 수단 제공
    (예) transient OS code and buffer

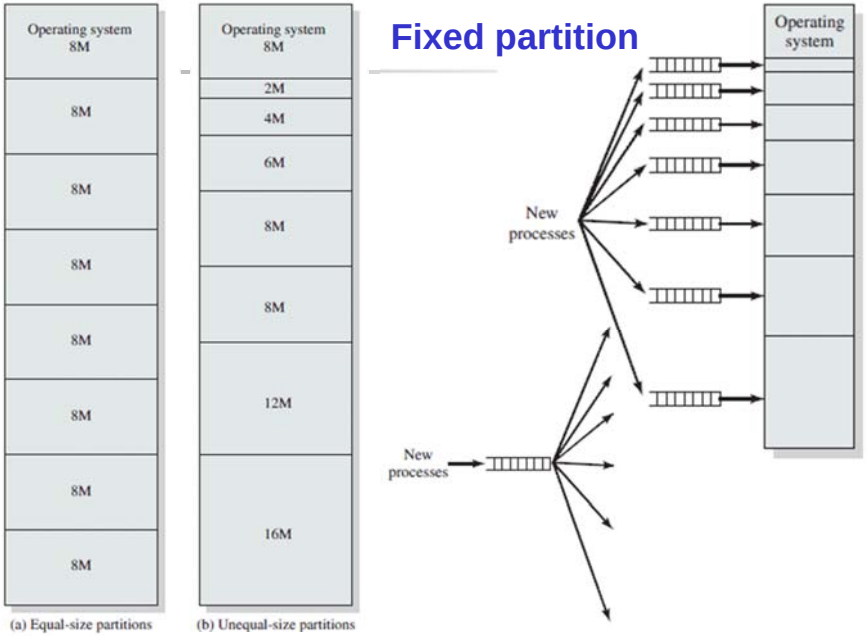
Hardware support for relocation and limit registers



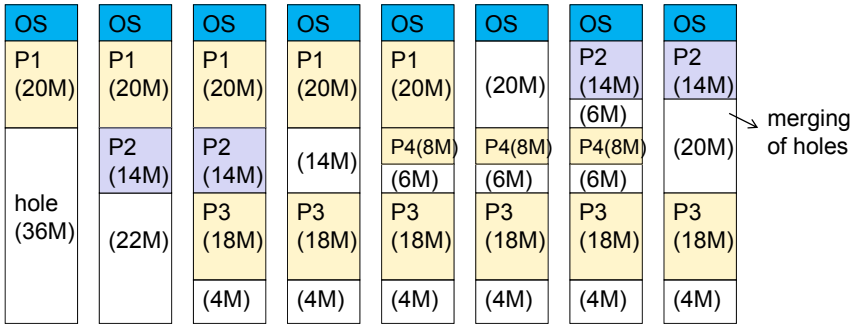
메모리 할당(Memory Allocation)

- 고정 분할 방식(Fixed partition scheme)
  - 메모리를 몇 개의 고정 크기 영역으로 분할
    - 동일 크기(equal size) 분할
    - 다른 크기(unequal size) 분할
  - 각 분할은 정확히 한 프로세스를 포함할 수 있음
    - degree of multiprogramming ≤ the number of partition
- 가변 분할 방식(Variable partition scheme)
  - 분할은 가변 길이 및 개수를 가짐
  - 운영체제는 다음 정보를 관리
    - allocated partitions
    - free partitions (**holes**) – available
  - 프로세스를 수용할 수 있는 크기의 hole에서 메모리를 할당
    - hole이 너무 크면, 한 부분을 할당하고 나머지 부분은 hole의 집합을 반환함.

Fixed partition



Variable Partition Scheme



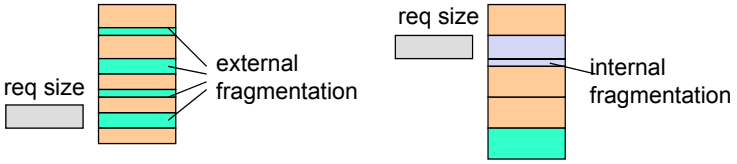
동적 메모리 할당 문제

(문제) free hole의 리스트에서 크기 n의 메모리 요청

- First-fit:
  - 요청을 수용할 수 있는 first hole을 할당
  - 검색 시작 위치 – (1) 리스트의 시작
    (2) 이전 first-fit 검색이 종료된 위치의 hole
- Best-fit:
  - 요청을 수용할 수 있는 smallest hole 을 할당
  - 리스트가 크기 순서로 정렬되어 있지 않으면 전체 리스트를 검색해야 함.
  - 가장 작은 크기의 남은 hole 생성
- Worst-fit:
  - 요청을 수용할 수 있는 largest hole 을 할당
  - 전체 리스트를 검색해야 함.
- Performance
  - First-fit 와 best-fit 방식이 검색시간과 기억장치 이용률에 있어서 worst-fit 방식보다 좋음

메모리 단편화(Memory Fragmentation)

- Memory Fragmentation
  - 사용할 수 없는 작은 메모리 영역이 존재함
- 메모리 단편화의 종류
  - 외부 단편화(**External fragmentation**)
    - 요청한 크기보다 작은 메모리 블록들만 존재
      → 전체 메모리 크기 합이 요청한 크기보다 큰 경우에도 요청한 크기의 메모리를 할당할 수 없음
  - 내부 단편화(**Internal fragmentation**)
    - 요청한 것 보다 더 큰 메모리를 할당
      → 할당된 메모리의 일부가 사용되지 않음

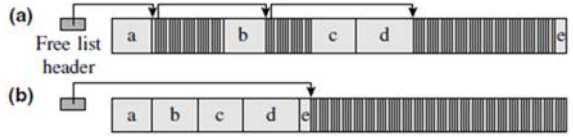


단편화(Fragmentation) 방식의 문제

- External fragmentation
  - 50-percent rule: N개의 블록이 할당되었을 때, first-fit 방식은 0.5N개의 블록이 외부 단편화로 사용될 수 없음
    → 메모리의 1/3을 낭비
  - hole을 추적하는 오버헤드가 큼
- Internal fragmentation
  - 메모리를 고정된 크기의 블록으로 분할 → hole의 추적이 필요 없음
  - 할당된 메모리는 요청한 메모리보다 더 클 수 있음 → 할당된 메모리의 일부 영역은 사용되지 않음

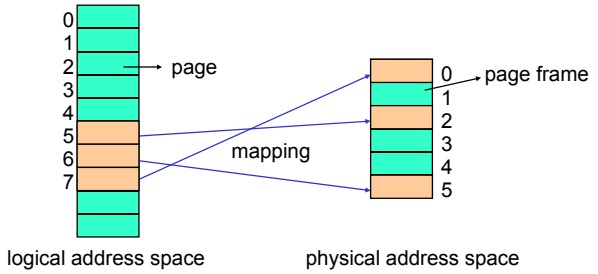
압축(Compaction)

- 외부 단편화 문제에 대한 해결책
  1. 압축(compaction)
  2. 비연속 물리 메모리 공간 허용
     - paging
     - segmentation
- Compaction
  - 모든 free memory를 하나의 큰 블록으로 합쳐서 재배치.
  - Compaction은 relocation이 동적이고 실행시간에 수행될 때에만 가능



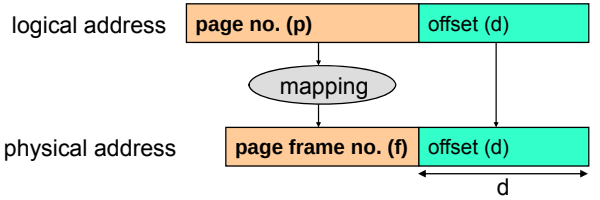
8.5 페이징(Paging)

- Paging
  - 비연속(noncontiguous) 물리적 메모리 할당 방법 중 하나
  - **page frame** : 물리적 메모리를 고정된 크기의 블록들로 나눔 (크기는 2^n, 대개 512B ~ 8KB).
  - **page**: 논리 메모리를 같은 크기의 블록으로 나눔
  - page를 page frame에 맵핑



주소 변환 방식

- 논리 주소를 다음과 같이 구분
  - 상위 부분 : page number
  - 하위 부분 : page내에서의 page offset (page size=2^d이면 d 비트)
- 주소 변환:
  - 논리 주소의 page number를 물리 주소의 page frame number로 변환
  - 주소 변환에 page table을 사용

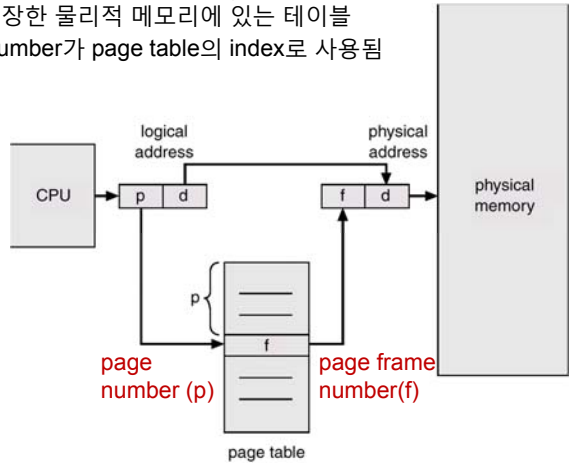


page size는 하드웨어에서 정해짐

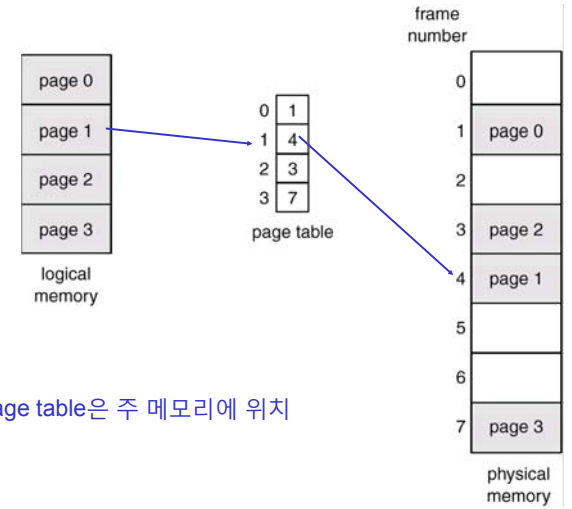
(page size = 2^d)

Paging hardware: page table

- Page Table
  - 각 페이지의 base address(page frame number)들을 저장한 물리적 메모리에 있는 테이블
  - page number가 page table의 index로 사용됨

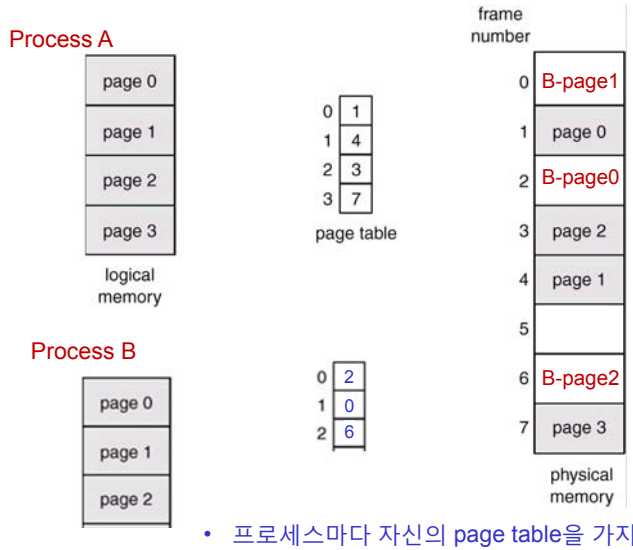


Paging 예



page table은 주 메모리에 위치

Paging 예(2)

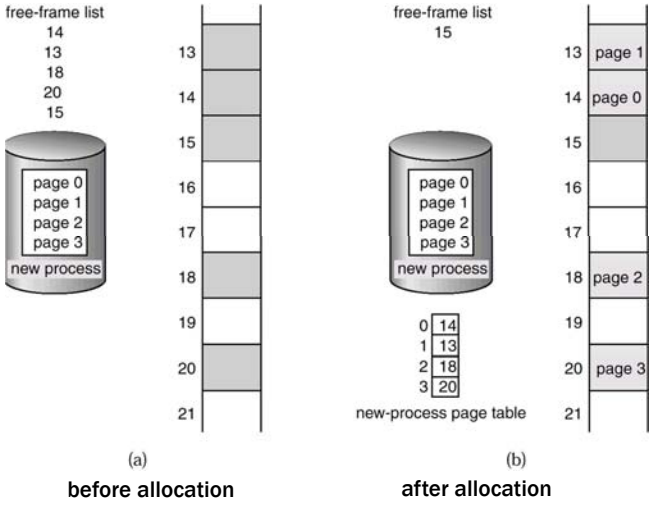


- 프로세스마다 자신의 page table을 가지고 있으며 자신의 논리 주소 공간을 가짐

Page 할당 ...

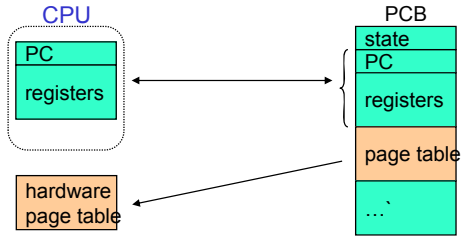
- Page 할당
  - 모든 free frame을 추적 → **free frame list**를 사용
  - *n* page 크기의 프로그램을 실행시키려면 *n* free frame을 찾아서 page를 할당하고, 할당된 frame에 프로그램을 적재함
  - 주소 변환을 위해서 page table을 할당된 page frame 번호들로 설정
  (See next slides)
- 내부 단편화
  - 메모리 할당 단위 : page frame (고정 크기)
    → 내부 단편화 발생, 외부 단편화 없음
  - 내부 단편화 크기 : 프로세스 당 1/2 페이지 크기
    - 작은 페이지 크기 → (장점) 작은 내부 단편화
      (단점) 많은 수의 페이지로 인해 큰 페이지 테이블 필요
    - 큰 페이지 크기 → (단점) 큰 내부 단편화
      (장점) 작은 페이지 테이블, 효율적 디스크 I/O
- 일부 CPU와 커널은 복수의 페이지 크기를 지원
  - Solaris : 8KB, 4MB
  - Pentium : 4KB, 4MB

Free frames



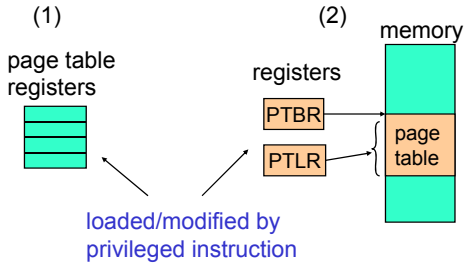
Frame table과 Page table

- Frame table
  - physical page frame 당 한 엔트리
    - frame의 free/allocated 여부 표시
    - 할당되었으면 어느 프로세스(들)의 어느 페이지에서 맵핑되었는 지를 기록
- 운영체제는 각 프로세스마다 자신의 페이지 테이블을 가짐
  - 하드웨어 page table을 사용하는 경우 context switching 시에 page table도 백업해야 하므로context-switch time 증가



Page Table의 구조

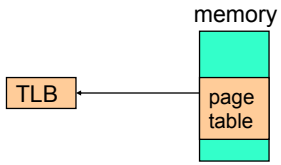
- Page table의 하드웨어 구현
  1. page table 전용 레지스터 집합
     - page table이 작을 때에만(≤ 256) 유용함
  2. page table을 주 메모리에 유지
     - page-table base register (PTBR) : page table 시작 주소
     - page-table length register (PTLR) : page table 크기



메모리에 있는 Page Table과 TLB

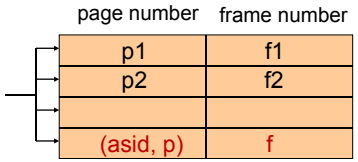
- 모든 data/instruction 접근에 두 번의 메모리 접근 필요
  - the page table
  - the data/instruction

  } 실제로 허용될 수 없음
- Translation Look-aside buffer(TLB)
  - 메모리 2회 접근 문제를 해결하기 위하여 TLB라고 하는 page table entry용 캐시를 사용
  - 대개 Associative memory를 사용하여 구현
  - 새로운 page table을 사용할 때마다 TLB 내용은 flush되어야 함
    - 일부 TLB는 address-space identifier(ASIDs)를 함께 저장하여 TLB flush를 필요 없게 함



Translation Lookaside Buffer(TLB)

- TLB 구성
  - Associative memory – parallel search(fully-associative mapped cache)



  - 일부 TLB는 set-associative mapped cache 사용
- Address translation (p, d)
  - ***p*** is in page number field of TLB, get the frame number → TLB hit
  - Otherwise, get frame number from page table in memory → TLB miss (그리고 page number와 frame number 를 TLB에 추가)
- 모든 TLB entry가 사용 중이면 운영체제는 새로운 entry를 위해 교체 대상 entry를 선택해야 함
  - 교체 정책: LRU(least recently used), random …

Paging hardware with TLB