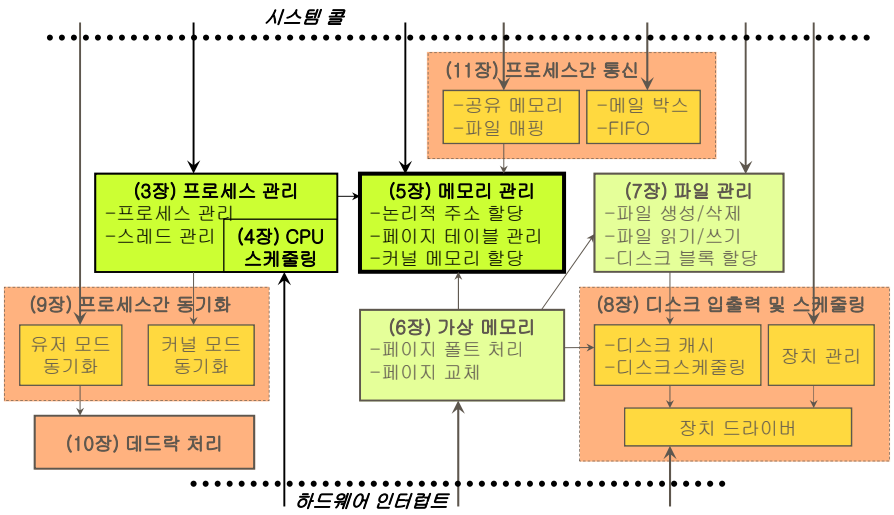
메모리 관리

- ✓실행시점 주소 바인딩
- ✓세그멘테이션의 동작 원리
- ✓페이징의 동작원리
- ✓페이지 테이블의 크기 감소
- ✓프로세스 별 페이지 테이블
- ✓메모리 할당 및 해제
- ✓실행프로그램 파일의 생성

관련 운영체제 구성 모듈

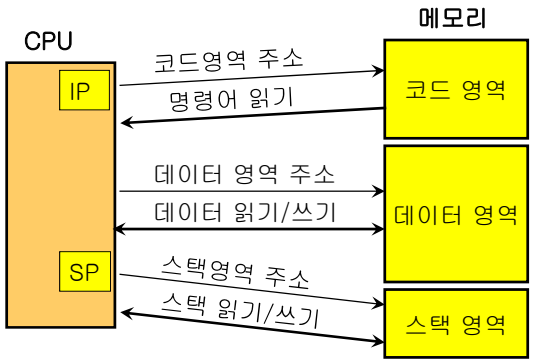


이해하고 넘어가야 할 내용들

- ✓ 실행시점 주소 바인딩의 이해
- ✓ 세그먼테이션의 동작 원리와 페이징의 동작 원리의 이해
- ✓ 문맥 교환 과정에서 페이지 테이블의 처리 방법의 이해
- ✓ 커널이 실행될 때의 페이지 테이블 처리 방안
- ✓ 메모리 할당 정책의 비교
- ✓ 실행 프로그램 파일의 생성 및 실행 과정의 이해
- ✓ 동적 링킹 및 동적 적재의 동작 원리 및 장단점의 이해

CPU의 메모리 읽기/쓰기

- ✓ **명령어 인출 (instruction fetch) : 코드영역의 명령어 주소(IP)를 내보내고 명령어를 읽어 들임**
- ✓ **명령어 실행: 명령어 별 오퍼랜드에 따라 데이터 영역의 주소를 내보내고 데이터를 읽기/쓰기**
- ✓ **스택 명령어 실행: 스택 주소 (SP) 를 내보내고 스택의 데이터를 읽기/쓰기**



주소 바인딩

- ✓ **주소 바인딩 (address binding)**
  - ✓ 코드 영역에서 읽어온 명령어의 주소지정 부분에 표현된 주소를 실제 적재된 메모리의 주소로 변환하는 작업
  - ✓ 주요 대상 : 함수의 주소와 전역변수의 주소
- ✓ **컴파일 결과로 실행파일을 생성하는 시점에는 적재될 메모리의 주소를 미리 알 수가 없음**
  - ➔ 적재시점이나 실행시점에 주소를 변환해 주어야 함

컴파일된 실행프로그램 파일

C 언어 프로그램

```
int val = 12;
…
main()
{
  val = 34;
  val = val + func();
  …
}
func()
{
  return 20;
}
```

실행 프로그램 파일

주소 코드 섹션

```
main: movl $34, val
      call func
      addl val, %eax
      movl %eax, val
       …
      ret
func: movl $20, %eax
      ret
```

데이터 섹션

```
val: 12
```

*코드 섹션의 **val** 부분은 데이터 영역 적재 주소에 따라, **func** 부분은 코드 섹션의 적재 주소에 따라 결정될 것임*

적재된 주소에 따른 주소 바인딩

1000번지부터 적재시

주소	코드영역
1000:	movl $34, ***1200***
	call ***1030***
	addl ***1200***, %eax
	movl %eax, ***1200***
	…
	ret
1030:	movl $20, %eax
	ret
	데이터영역
1200:	12

2000번지부터 적재시

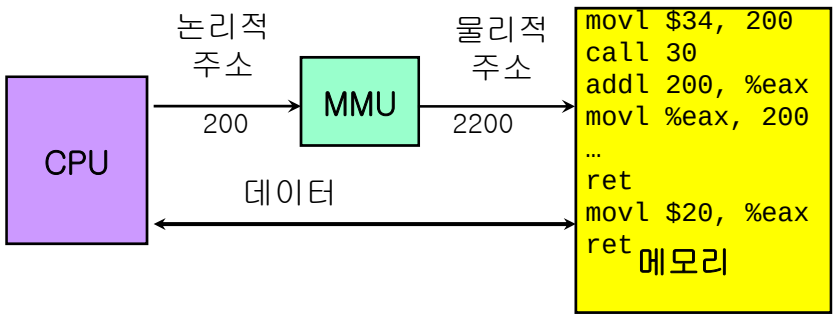
주소	코드영역
2000:	movl $34, ***2200***
	call ***2030***
	addl ***2200***, %eax
	movl %eax, ***2200***
	…
	ret
2030:	movl $20, %eax
	ret
	데이터영역
2200:	12

바인딩 시점

- ✓ **컴파일 시점 바인딩 (compile time binding)**
  - ✓ 컴파일 시점에 적재될 주소가 확정되어 있는 경우 사용
  - ✓ 실행을 위해 메모리에 적재하는 작업은 정해진 주소에 **그대로 적재**
- ✓ **적재 시점 바인딩 (load time binding)**
  - ✓ 실행을 위해 적재할 시점에 임의로 메모리 영역을 할당
  - ✓ 할당된 메모리 주소에 맞게 명령어의 주소 부분을 **수정하여 적재**
- ✓ **실행 시점 바인딩 (execution time binding)**
  - ✓ 적재 시점에 임의로 할당된 메모리에 코드섹션을 수정없이 **그대로 적재**
  - ✓ 실행시에 CPU로 부터 나오는 주소(논리적 주소)를 하드웨어 로직을 통해 할당된 주소에 맞게 변경하여(물리적 주소) 메모리로 전달
  - ✓ 메모리 관리 유닛 (MMU: Memory Management Unit)이 필요함
    - ✓ 논리적 주소를 물리적 주소로 변환하는 하드웨어 모듈
    - ✓ 세그멘테이션 (segmentation) 방식과 페이징 (paging) 방식이 있음

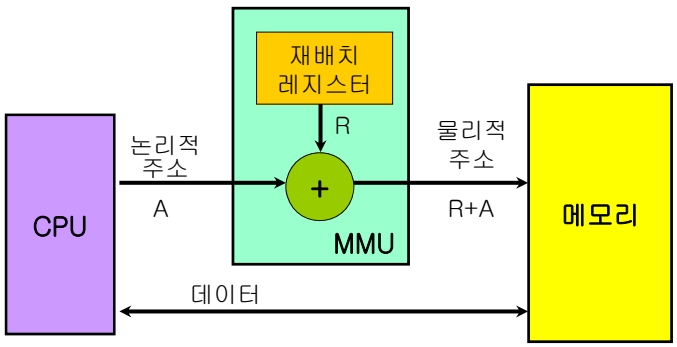
실행시점 주소 바인딩

- ✓ 논리적 주소는 0번지 부터 시작되는 것을 전제로 결정
- ✓ 실행시 2000번지에 적재되었을 경우 주소매핑은 논리적 주소에 2000을 더하여 결정

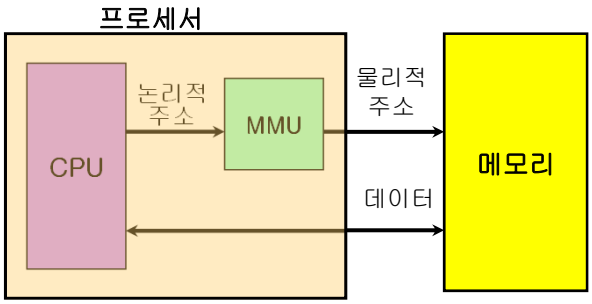


재배치 레지스터를 이용한 MMU

- ✓ 재배치 레지스터 (relocation register)
  - ✓ 할당된 메모리 영역의 시작 주소 기록
- ✓ MMU는 이진 가산기로 구성
- ✓ 논리적 주소에 재배치 레지스터 값을 더한 주소를 메모리에 전달

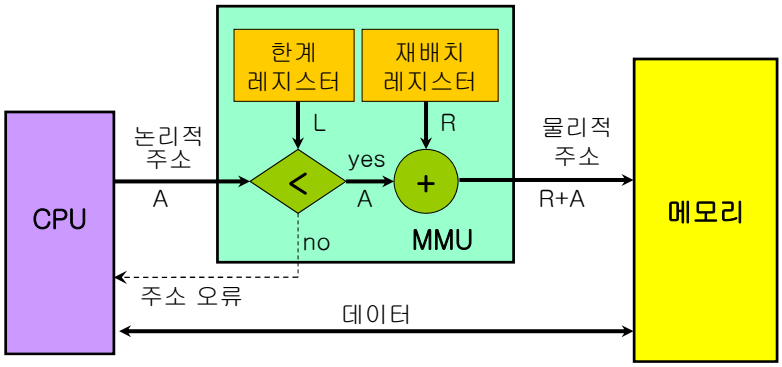


프로세서 속의 MMU



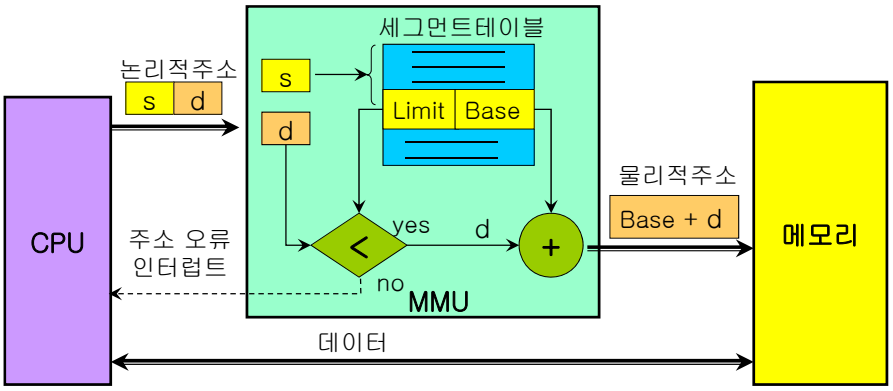
세그먼테이션의 동작원리

- ✓ **주소바인딩: 재배치 레지스터의 값을 더함 (이진 가산기 적용)**
- ✓ **메모리 영역 보호: 한계 레지스터와 비교 (이진 비교기 적용)**
  - ✓ 할당된 메모리 내에서만 허용
  - ✓ 논리적 주소가 한계 레지스터 값을 초과시 주소오류 인터럽트 발생

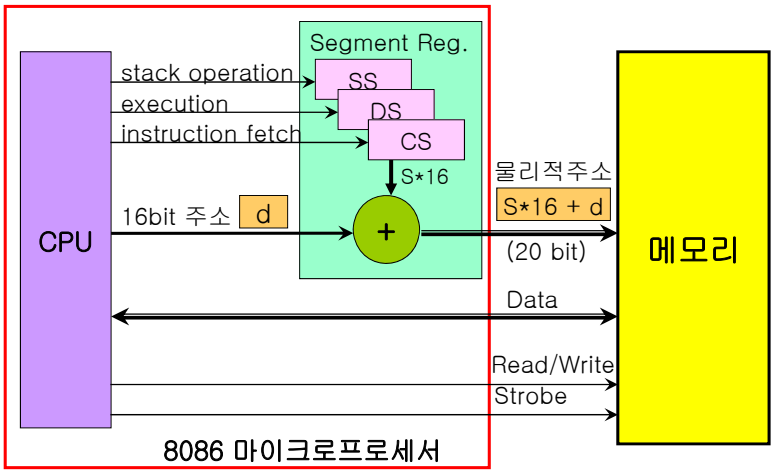


세그먼테이션 MMU의 동작

- ✓ 다수의 세그먼트 제공, 세그먼트 별로 메모리 영역 할당
- ✓ 논리적 주소를 비트 단위로 세그먼트 번호 부분 (s)과 오프셋 부분 (d)으로 구분

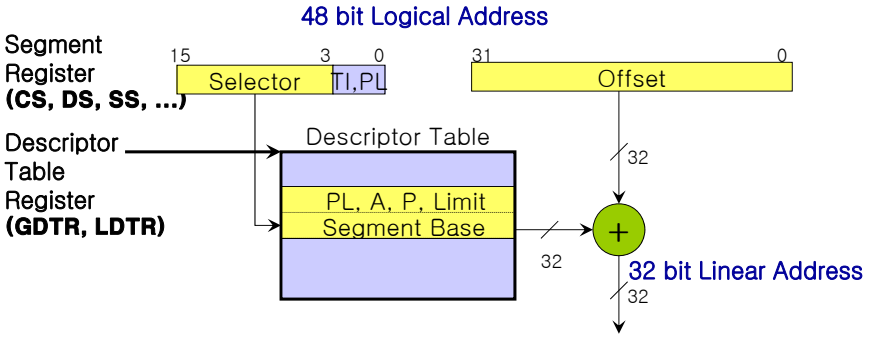


(참고) 8086 프로세서의 MMU 기능



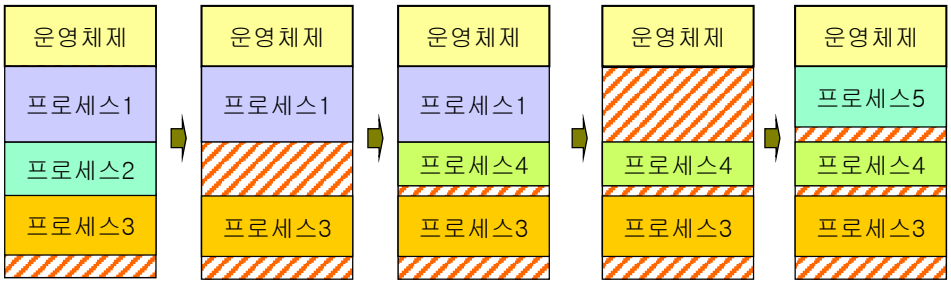
x86 프로세서의 세그먼테이션

- ✓ 세그먼트 테이블 (descriptor table)은 메모리에 작성
- ✓ MMU의 Descriptor Table Register에 테이블 시작주소 지정
- ✓ ㈜ x86 프로세서는 세그먼테이션에 페이징을 동시에 제공



세그멘테이션과 외부단편화

- ✓ 빈 메모리 영역 정보관리 → 적절한 크기의 메모리 공간을 찾는 데 시간 소요
- ✓ 메모리 할당과 해제를 반복하면서 외부 단편화 발생 → 메모리 압축 작업 필요

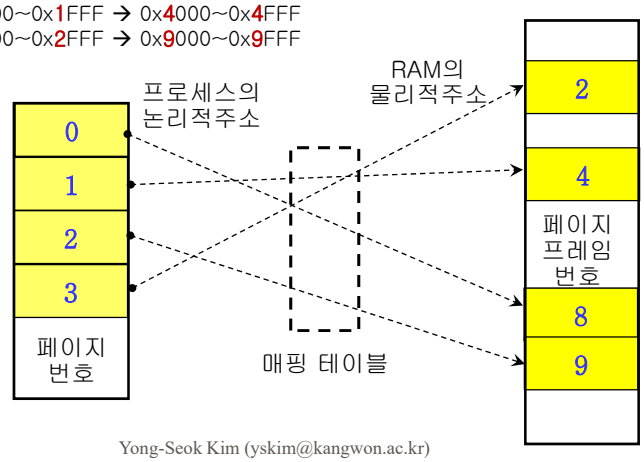


페이징의 동작원리

- ✓ **페이지 (page) 및 페이지 프레임 (page frame)**
  - ✓ 페이지: 프로세스의 논리적 주소영역을 일정한 크기로 나눈 것
  - ✓ 페이지 프레임: 전체 메모리를 페이지와 동일한 크기로 나눈 것
- ✓ **페이징 (paging)**
  - ✓ 메모리 할당: 프로세스 별로 페이지 수만큼의 페이지 프레임 할당
  - ✓ 주소바인딩: 하나의 페이지는 하나의 페이지 프레임으로 매핑
  - ✓ 주소 매핑 정보: 페이지 테이블에 관리
- ✓ **메모리 영역 보호**
  - ✓ 프로세스 별로 정상적인 페이지들에 대해서만 페이지 테이블 항목에 유효 (valid) 비트 설정
  - ✓ 유효비트가 설정되지 않은 페이지 테이블 항목 접근시 CPU로 주소 오류 인터럽트 발생

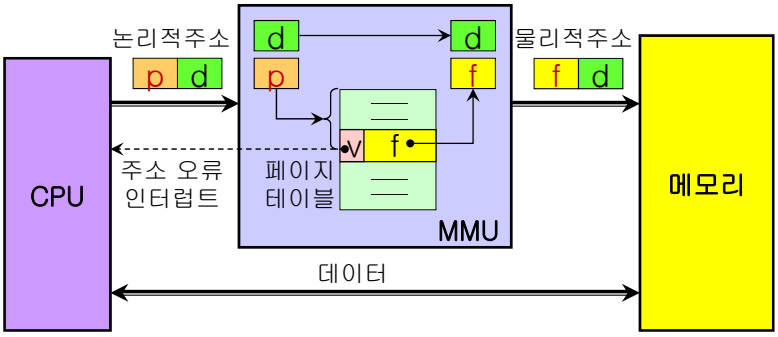
(추가) 페이징의 주소바인딩

- ✓ **논리적 주소를 페이지 번호 부분(p)과 페이지 내의 오프셋 부분(d)으로 구분**
- ✓ **페이지 번호 부분(p)만 페이지 프레임 번호(f)로 대체**
- ✓ **예시: 페이지 크기가 4KB 이고 매핑이 다음과 같다면**
  - ✓ 0x0000~0x0FFF → 0x8000~0x8FFF
  - ✓ 0x1000~0x1FFF → 0x4000~0x4FFF
  - ✓ 0x2000~0x2FFF → 0x9000~0x9FFF
  - ✓ …



페이징 MMU의 동작

- ✓ 논리적 주소를 페이지 번호 부분(p)과 오프셋 부분(d)으로 구분
- ✓ 페이지 번호 부분(p)만 프레임 번호(f)로 대체
- ✓ 페이지 내의 오프셋 주소 부분(d)은 그대로 메모리로 전달
- ✓ (예) 0x2000~0x2FFF → 0x9000~0x9FFF

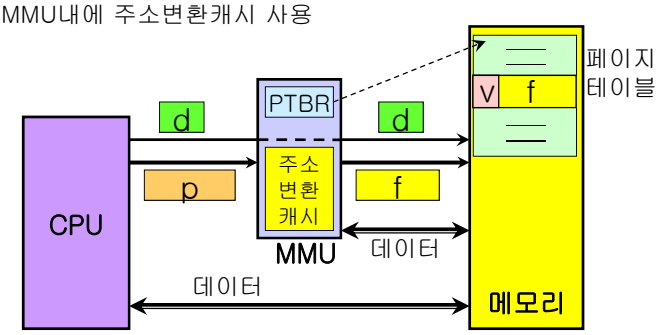


페이징의 장단점

- ✓ **메모리 할당이 단순함**
  - ✓ 페이징: 필요한 페이지 수 만큼의 빈 페이지 프레임들을 할당
  - ✓ 세그먼테이션: 세그먼트별 크기에 맞는 연속된 메모리 영역 선택
- ✓ **단편화 (fragmentation)가 작음**
  - ✓ 세그먼테이션에서는 할당된 메모리 영역 사이에 활용하지 못하는 단편들이 존재 (외부 단편화)
  - ✓ 페이징은 페이지 단위로 할당하므로 마지막 페이지 프레임은 일부가 사용되지 않는 단편 존재 (내부 단편화)
  - ✓ 내부 단편화는 외부 단편화 보다 메모리 낭비가 적음
- ✓ **페이지 테이블의 크기가 큼**
  - ✓ 논리적 주소 공간 전체 페이지 수만큼 페이지 테이블 항목 필요
  - ✓ (예) 32비트 주소에 4KB 크기 페이지 사용시 (p 20비트, d 12비트) : 2^{20}개의 페이지테이블 항목 필요
  - ✓ 페이지 개수가 세그먼트 개수보다 월등히 많음
  - ➔ 페이지 테이블의 크기를 줄이기 위해 다단계 페이징 사용

메모리 상의 페이지 테이블

- ✓ **페이지 테이블이 크므로 메모리에 페이지 테이블 정보 관리**
- ✓ **MMU는 메모리 상의 페이지 테이블 정보를 읽어들여 사용**
- ✓ **페이지 테이블 읽기 오버헤드 발생**
  - ➔ MMU내에 주소변환캐시 사용

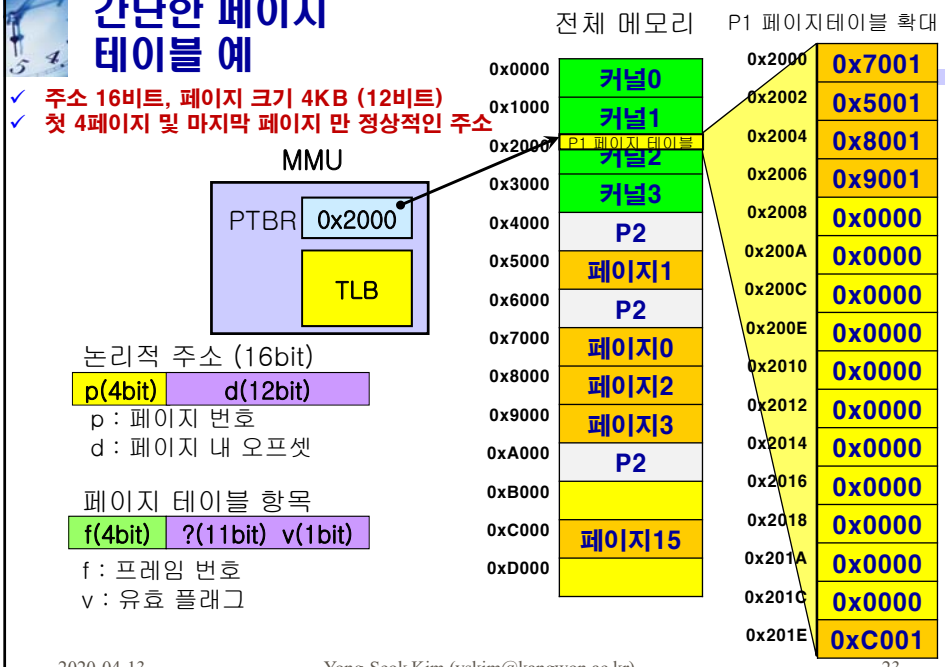


평균 메모리 접근 시간

- ✓ **메모리 접근시간 2배로 증가**
  - ✓ CPU의 메모리 접근 시 마다 MMU는 주소 바인딩을 위해 메모리 (페이지 테이블) 읽기 실행
- ✓ **주소변환캐시 (address translation cache 또는 translation look-aside buffer)**
  - ✓ MMU내부의 캐시에 페이지 테이블 내용 일부 저장
  - ✓ 메모리로부터 읽어온 페이지 테이블 내용을 읽어오는 회수 감소
- ✓ **평균 메모리 접근 시간 (effective access time)**
  - ✓ 가정: 메모리 접근시간 100ns, 캐시 검사시간 10ns
  - ✓ 캐시에 항목이 있을 때의 소요시간: 10 + 100 ns
  - ✓ 캐시에 없을 때의 소요시간: 10 + 100(테이블 읽기) + 100 ns
  - ✓ r : 캐시 성공률
  - ✓ EAT = 110r + 210(1-r) = 210 – 100r
  - ✓ r=0.98 이면 EAT = 112ns ➔ 12%의 속도저하

간단한 페이지 테이블 예

- ✓ 주소 16비트, 페이지 크기 4KB (12비트)
- ✓ 첫 4페이지 및 마지막 페이지 만 정상적인 주소



논리적 주소 (16bit)

p(4bit)	d(12bit)

p : 페이지 번호
d : 페이지 내 오프셋

페이지 테이블 항목

f(4bit)	?(11bit) v(1bit)

f : 프레임 번호
v : 유효 플래그

메모리 할당 시나리오 예

0. 현재 빈 페이지 프레임 목록 = [7,5,8,9,12,11,13]

1. 프로세스 P1 생성 요청
PCB = PCB 할당
PageTable = 메모리 할당 (32바이트) ➔ 0x2000 ~ 0x2031 할당
PCB.PageTableAddr = PageTable (0x2000)

2. 실행 파일의 헤더 부분 읽기
코드영역 크기 (0x0000~0x1FFF), 데이터 영역 크기 (0x2000~0x3FFF) 확인

3. 코드영역을 위해 빈 페이지 프레임 2개 할당 ➔ 7,5번 프레임
실행파일 코드섹션의 두 페이지를 차례대로 0x7000 번지 대, 0x5000 번지 대에 읽어 들임
페이지 테이블의 0,1번 항목 등록 ➔ 0x7001, 0x5001

4. 데이터 영역을 위해 페이지 프레임 2개 할당 ➔ 8,9번 프레임
실행파일 데이터 섹션의 세 페이지를 차례대로 0x8000 번지 대, 0x9000 번지 대에 읽어 들임
페이지 테이블 2,3번 항목 등록 ➔ 0x8001, 0x9001

5. 스택 영역을 위해 페이지 프레임 1개 할당 ➔ 12번 프레임
페이지 테이블 15번 항목 등록 ➔ 0xC001
PCB.StackAddr = 0xFFFE; PCB.StackSize = 0x1000

6. PCB를 레디 큐에 등록
이후에 이 프로세스가 선택되어 문맥교환될 때 PCB.PageTableAddr 을 MMU의 PTBR에 기록

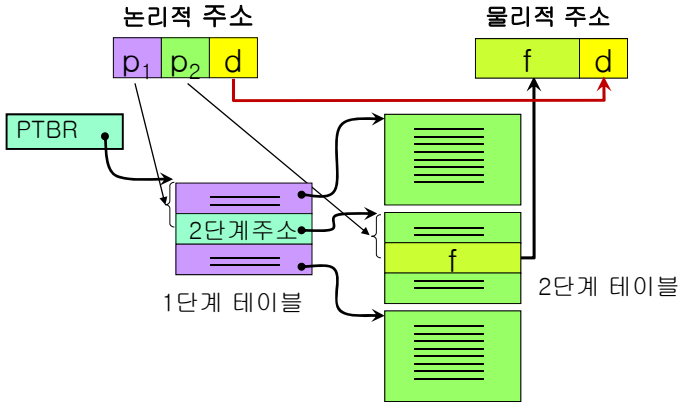
➔ 현재 빈 페이지 프레임 목록 = [11,13]

8. P1 종료 시 사용하던 페이지 프레임들을 빈 페이지 프레임 목록에 반환

➔ 현재 빈 페이지 프레임 목록 = [11,13,7,5,8,9,12]

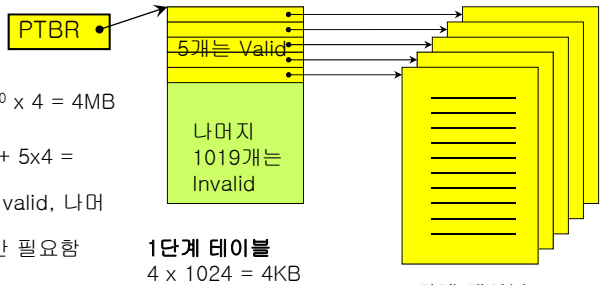
다단계 페이징

- ✓ **논리적 주소를 여러 단계에 걸쳐서 주소 변환**
- ✓ **2단계 페이징 예**
  - ✓ 논리적 주소 32비트, 1단계 (p_1) 10비트, 2단계 (p_2) 10비트, 페이지내의 주소 (d) 12비트



페이지 테이블 크기 감소

- ✓ **크기감소 근거**
  - ✓ 각 프로세스는 전체 논리적 주소공간의 일부만 사용함
  - ➔ 하위단계의 테이블 중 일부는 생략 가능
- ✓ **가정**
  - ✓ 값 논리적 주소 32비트, 한 페이지 크기 4KB, 페이지 테이블 항목당 4바이트
  - ✓ 프로세스는 0 ~ 20MB 까지의 주소 사용
- ✓ **1단계 페이징 사용시**
  - ✓ 페이지 테이블 크기: 2^{20} x 4 = 4MB
- ✓ **2단계 페이징 사용시**
  - ✓ 페이지 테이블 크기: 4 + 5x4 = 24KB
  - ✓ 1단계는 첫 5개 항목만 valid, 나머지 1019 항목은 invalid
  - ✓ 2단계는 5개의 테이블만 필요함



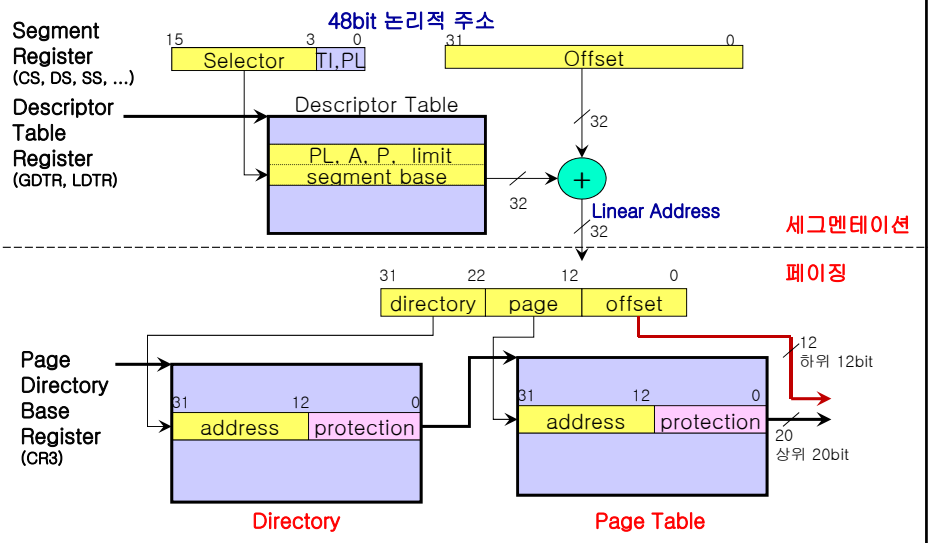
1단계 테이블
4 x 1024 = 4KB

2단계 테이블
4KB x 5개 = 20KB

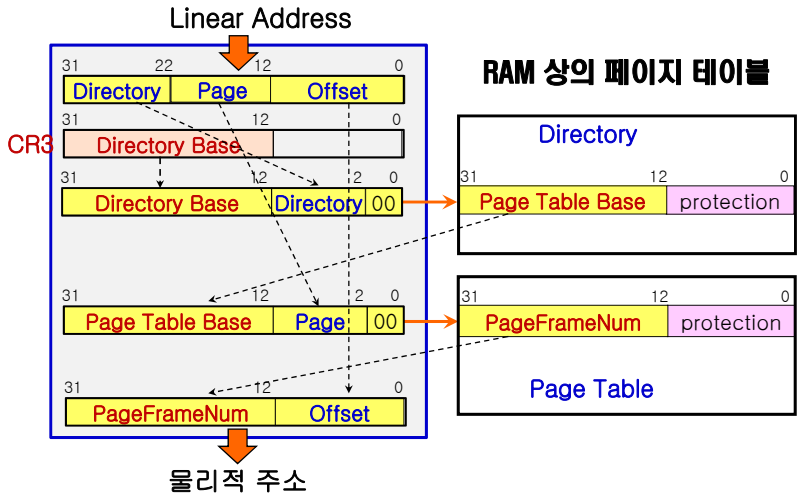
다단계 페이징의 EAT

- ✓ **기본 파라미터**
  - ✓ 메모리 접근시간 100ns, 캐시 검사시간 10ns
  - ✓ 캐시에 항목이 있을 때의 소요시간: 10 + 100 ns
  - ✓ r = 0.98
- ✓ **1단계 페이징**
  - ✓ 캐시에 없을 때의 소요시간: 10 + 100(테이블 읽기) + 100 ns
  - ✓ EAT = 110r + 210(1−r) = 112 ns
  - ✓ 12%의 속도저하
- ✓ **2단계 페이징**
  - ✓ 캐시에 없을 때의 소요시간: 10 + 200(테이블 읽기) + 100 ns
  - ✓ EAT = 110r + 310(1−r) = 114 ns
  - ✓ 14%의 속도저하
  - ➔ r 이 높으면 다단계 페이징도 문제 없음

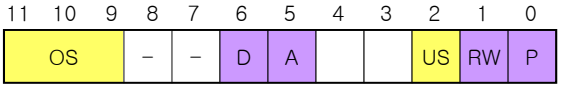
x86의 Segmentation with Paging



x86 MMU의 페이징 부분

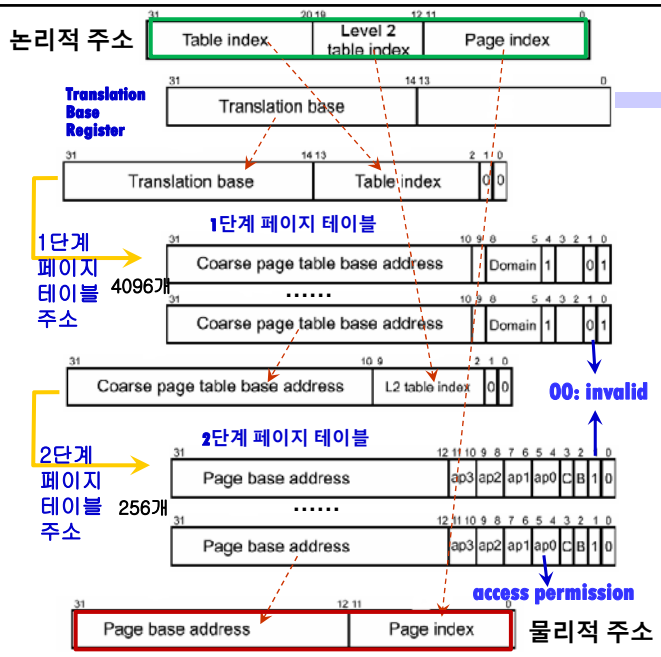


x86의 Protection 항목

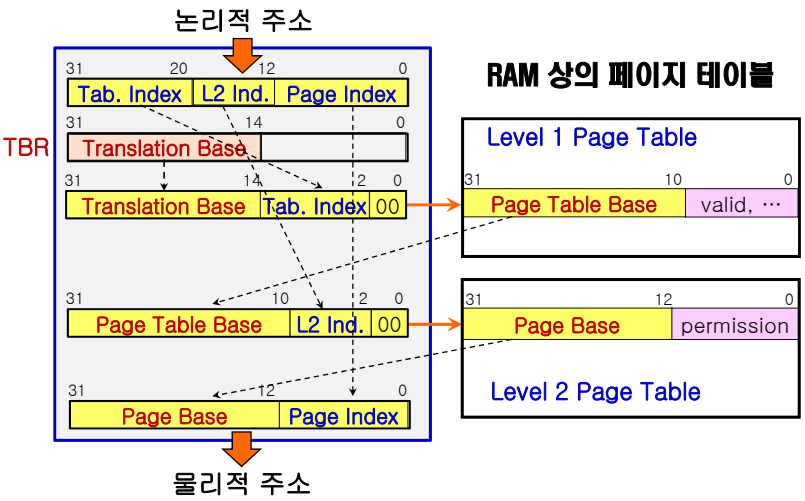


OS: OS에서 필요에 따라 사용 (page aging 등)
D: dirty
A: accessed
US: user-supervisor(1) / supervisor-only(0)
RW: read-write(1) / read-only(0)
P: present(1) / invalid(0)

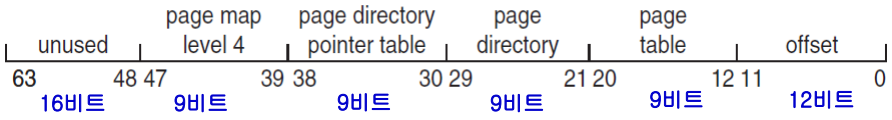
ARM의 2단계 페이징



ARM MMU의 2단계 페이징

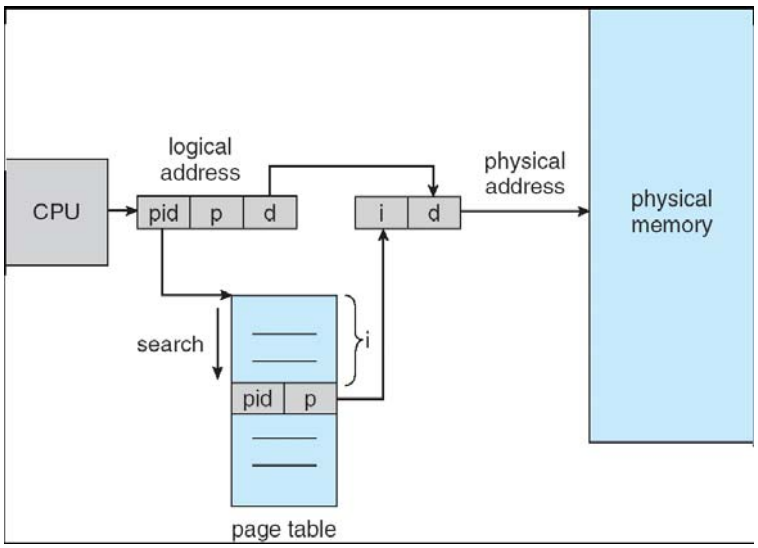


x86-64 프로세서의 4단계 페이징



- ✓ **Linux에서 사용하는 명칭**
  - ✓ **1**단계: **pgd (page global directory)**
  - ✓ **2**단계: **pud (page upper directory)**
  - ✓ **3**단계: **pmd (page middle directory)**
  - ✓ **4**단계: **pte (page table entry)**
  - ✓ **(**참고**) 2**단계 페이징에서는 **pud, pmd**를 생략함

역 페이지 테이블

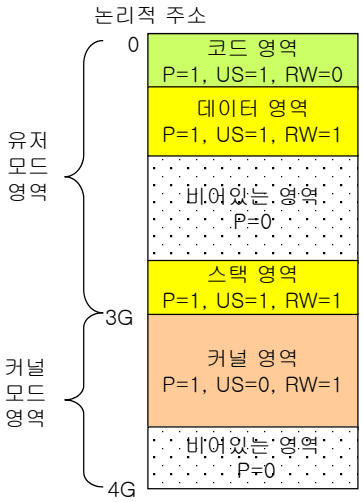


프로세스 별 페이지 테이블

- ✓ **문맥교환 과정의 페이지 테이블 변경**
  - ✓ 프로세스 별로 페이지 테이블 보유
  - ✓ 문맥교환 과정에서 페이지 테이블 주소 변경
  - ✓ (예) **`MMU->PTBR = next->Process->PageTable;`**
  - ✓ MMU 내부의 주소변환 캐시 내용 삭제
- ✓ **커널 모드 페이지 테이블**
  - ✓ 커널 모드에서는 모든 프로세스가 동일한 주소 공간 접근
  - ✓ 논리적 주소공간을 유저모드 용 영역과 커널모드 용 영역으로 분리
  - ✓ 커널모드 용 영역의 페이지 테이블은 모든 프로세스가 공유 가능

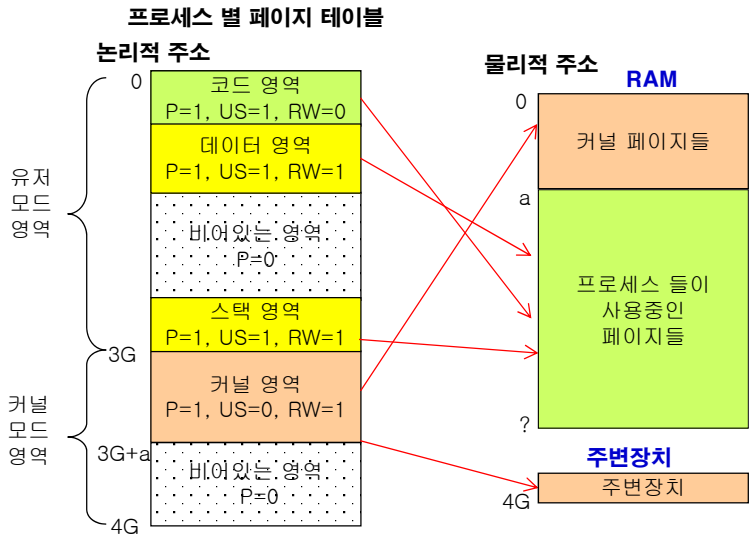
리눅스의 논리적 주소공간 분리

프로세스 별 페이지 테이블

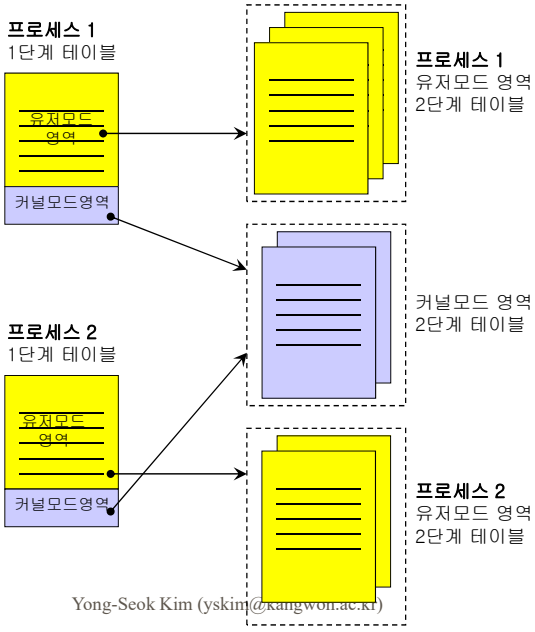


- ✓ **유저모드 영역: 3GB**
- ✓ **커널모드 영역: 1GB**
- ✓ **x86의 Protection**
  - ✓US: user(1)/supervisor(0)
  - ✓RW: read-write(1)/read(0)
  - ✓P: present(1)/invalid(0)

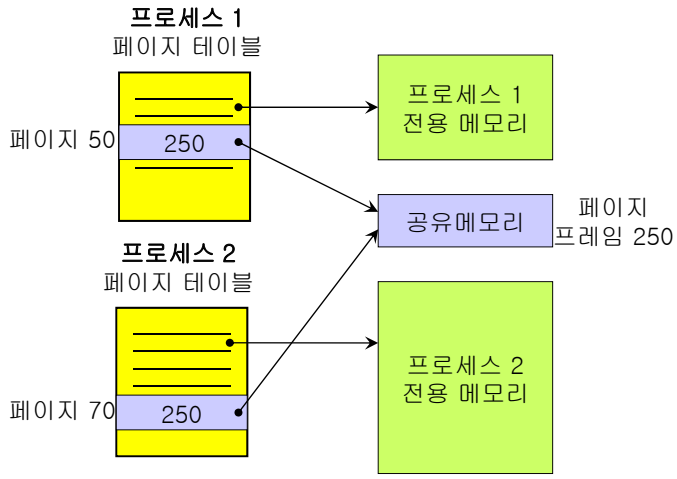
리눅스의 논리적 주소 매핑



커널모드 페이지 테이블의 공유

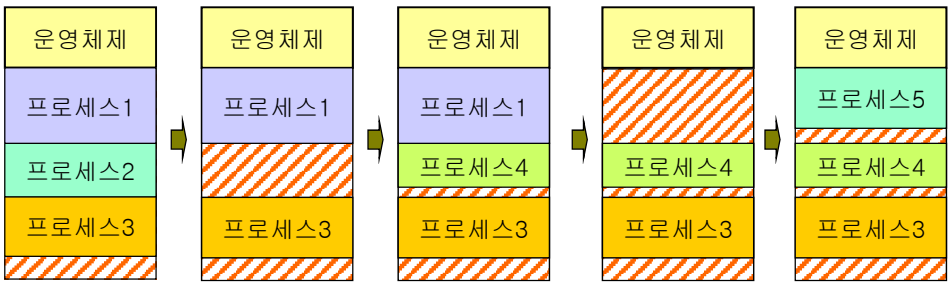


프로세스간의 공유 메모리



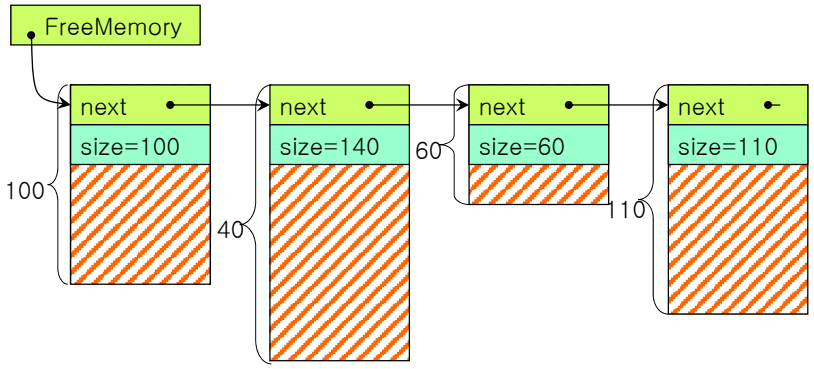
연속된 메모리 영역의 할당

- ✓ **메모리 할당 정책들**
  - ✓ 최초적격 (First-Fit), 최상적격 (Best-Fit), 최악적격 (Worst-Fit)
  - ✓ 일반적으로 First-Fit을 많이 사용
- ✓ **메모리 할당과 해제를 반복하면서 외부 단편화 발생**



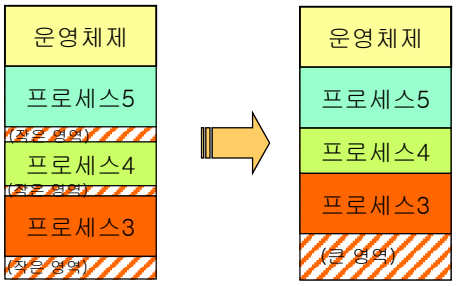
빈 메모리 영역 관리

- ✓ 모든 메모리는 특정 프로세스에 등록되거나 빈 메모리 목록에 등록
- ✓ 프로세스 제어블록에는 프로세스 별로 할당된 메모리 정보 기록

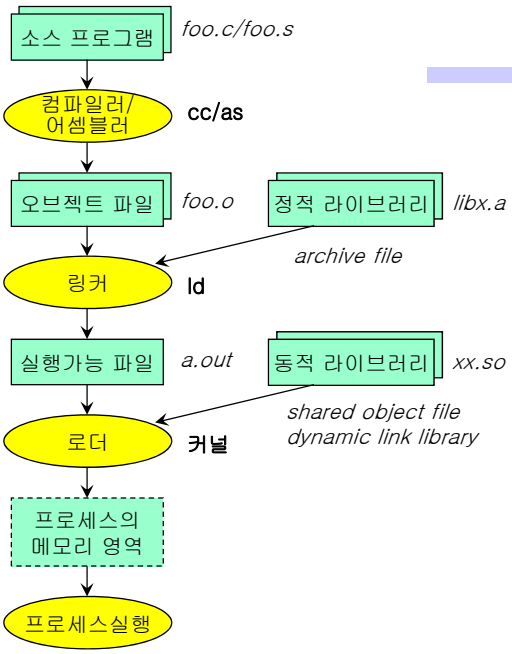


메모리 압축 (compaction)

- ✓ 외부 단편화로 인한 작은 조각들을 모아서 큰 영역으로
- ✓ 일명 garbage collection



프로그램의 컴파일 및 실행



동적 링킹

- ✓ **동적 링킹 (Dynamic Linking)**
  - ✓ 실행프로그램에는 필요한 모듈들에 대한 정보만 기록
  - ✓ 실행해야 하는 시점에 필요한 모듈들을 찾아서 메모리에 적재
  - ✓ 적재시점 동적 링킹: 프로세스가 생성되는 시점에 모든 필요한 모듈들일 찾아서 결합하여 메모리에 적재
  - ✓ 실행시점 동적 링킹: 모듈들이 빠진 상태로 실행을 시작하고 모듈별로 실제 사용되는 시점에 파일로부터 찾아서 적재
- ✓ **동적 링킹의 장점**
  - ✓ 모듈별 버전 업그레이드가 간편
  - ✓ 실제 필요한 모듈만 적재하므로 메모리 사용량 절약 (실행시점 동적 링킹)
- ✓ **공유 라이브러리**
  - ✓ 동적링킹을 기반으로 구현
  - ✓ 필요한 모듈이 이미 메모리에 적재되어 있으면 공유하여 사용

  ➔ 메모리 사용량 절약