

# 주기억장치 관리

# Main Memory Management

양희재 교수 (hjyang@ks.ac.kr) / 경성대학교 컴퓨터공학과

# 메모리 역사

- 메모리 역사

- Core memory
- 진공관 메모리
- 트랜지스터 메모리
- 집적회로 메모리: SRAM, DRAM

- 메모리 용량

↳ 하드웨어 메모리.

- 1970년대: 8-bit PC 64KB
- 1980년: 16-bit IBM-PC 640KB > 1MB > 4MB
- 1990년: 수MB > 수십 MB
- 2000년~: 수백 MB > 수 GB

# 언제나 부족한 메모리

- 프로그램 변천

- 기계어/어셈블리어 작성
- C언어 작성
- 자바, 객체지향형 언어 작성
- 숫자 처리 > 문자 처리 > 멀티미디어 처리 > Big Data

- 메모리 용량 증가 vs 프로그램 크기 증가

- 언제나 부족한 메모리

- 어떻게 메모리를 효과적으로 사용할 수 있을까?

- 메모리 낭비 없애기
- 가상 메모리 (virtual memory)

# 프로그램을 메모리에 올리기

- 메모리 구조
  - 주소(Address) + 데이터(Data)
- 프로그램 개발
  - 원천파일 (Source file): 고수준언어 또는 어셈블리언어
  - 목적파일 (Object file): 컴파일 또는 어셈블 결과
  - 실행파일 (Executable file): 링크 결과
- 컴파일러, 어셈블러, 링커, 로더
  - Compiler, assembler, linker, loader
- 프로그램 실행: code + data + stack

# 프로그램을 메모리에 올리기

- 실행파일을 메모리에 올리기
  - 메모리 몇 번지에?
  - 다중 프로그래밍 환경에서는?
- MMU 사용
  - 재배치 레지스터 (Relocation register), Base, Limit
- 주소 구분
  - 논리주소 (*logical address*) vs 물리주소 (*physical address*)  
: CPU가 보는 주소 →  → : 실제 메모리의 주소



# 메모리 낭비 방지

- Dynamic Loading
- Dynamic Linking
- Swapping

실행파일을 메모리에 옮기는 것.

# 동적 적재 (Dynamic Loading)

- 프로그램 실행에 반드시 필요한 루틴/데이터만 적재
  - 모든 루틴(routine)이 다 사용되는 것은 아니다 (예: 오류처리)
  - 모든 데이터(data)가 다 사용되는 것은 아니다 (예: 배열)
  - 자바: 모든 클래스가 다 사용되는 것은 아니다
  - 실행 시 필요하면 그때 해당 부분을 메모리에 올린다.
    - cf. 정적 적재 (Static loading)

# 동적 연결 (Dynamic Linking)

- 여러 프로그램에 공통 사용되는 라이브러리
  - 공통 라이브러리 루틴(library routine)를 메모리에 중복으로 올리는 것은 낭비
  - 라이브러리 루틴 연결을 실행 시까지 미룬다.
  - 오직 하나의 라이브러리 루틴만 메모리에 적재되고,
  - 다른 애플리케이션 실행 시 이 루틴과 연결(link)된다.
    - cf. 정적 연결 (Static linking)
  - **공유 라이브러리 (shared library)** – Linux 또는,
  - **동적 연결 라이브러리 (Dynamic Linking Library)** - Windows

# Swapping

- 메모리에 적재되어 있으나 현재 사용되지 않고 있는 프로세스 이미지
  - 메모리 활용도 높이기 위해 Backing store (= swap device) 로 돌아내기  
↳ 작은 컴퓨터는 하드 디스크 하나는  
File system과 Backing store로 나누기 사용.  
큰 컴퓨터(서버, 슈퍼컴퓨터 등)의 경우는 하드 디스크를 두개도 함.
  - *swap-out* vs. *swap-in*
  - Relocation register 사용으로 적재 위치는 무관
  - 프로세스 크기가 크면 backing store 입출력에 따른 부담 크다.

연속 메모리 할당

Contiguous Memory Allocation

# 연속 메모리 할당

- 다중 프로그래밍 환경

- 부팅 직후 메모리 상태: O/S + big single hole
- 프로세스 생성 & 종료 ↪ scattered holes



- 메모리 단편화 (Memory fragmentation)

- Hole들이 불연속하게 흩어져 있기 때문에 프로세스 적재 불가
- 외부 단편화 (external fragmentation) 발생
- 외부 단편화를 최소화 하려면?

빈공간을 다 허용해 놓는다  
충분하지만 공간이 빙어되어 있거나  
사용할 수 없음

# 연속 메모리 할당

- 연속 메모리 할당 방식

- First-fit (최초 적합); 초기에는 순차적으로 탐색해 들어갈 수 있는 첫 공간에 넣는 것
- Best-fit (최적 적합); 사이즈가 제일 비슷한 곳에 넣는 것
- Worst-fit (최악 적합); 사이즈가 더 큰 것 중에 제일 앙苦难을 가진 곳에 넣는 것

- 예제

- Hole: 100 / 500 / 600 / 300 / 200 KB
- 프로세스: 212 417 112 426 KB

# 연속 메모리 할당

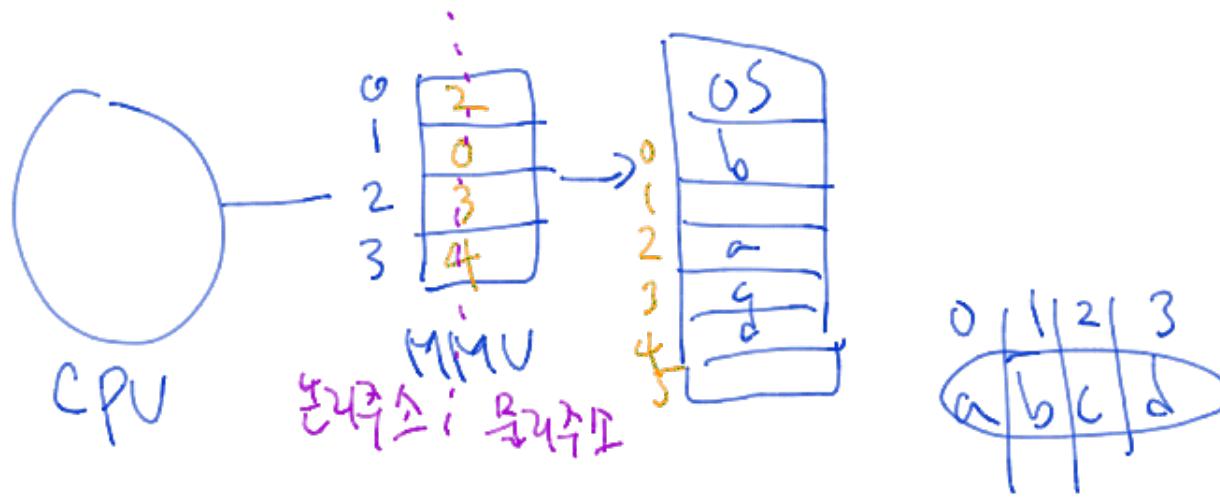
- 할당 방식 성능 비교: 속도 및 메모리 이용률
  - 속도: first-fit
  - 이용률: first-fit, best-fit
- 외부 단편화로 인한 메모리 낭비: 1/3 수준 (사용 불가)
  - Compaction : 최적 알고리즘 없음, 고부담 : 빈 공간들을 하나로 모으는 것
  - 다른 방법은?  
      페이지!

페이지

Paging

# 페이지 (Paging)

- 프로세스를 일정 크기(페이지)로 잘라서 메모리에!
  - 프로세스는 페이지(page)의 집합
  - 메모리는 프레임(frame)의 집합
- 페이지를 프레임에 할당
  - MMU 내의 재배치 레지스터 값을 바꿈으로서
  - CPU 는 프로세스가 연속된 메모리 공간에 위치한다고 착각
  - MMU 는 페이지 테이블 (page table) 이 된다.



# 주소 변환 (Address Translation)

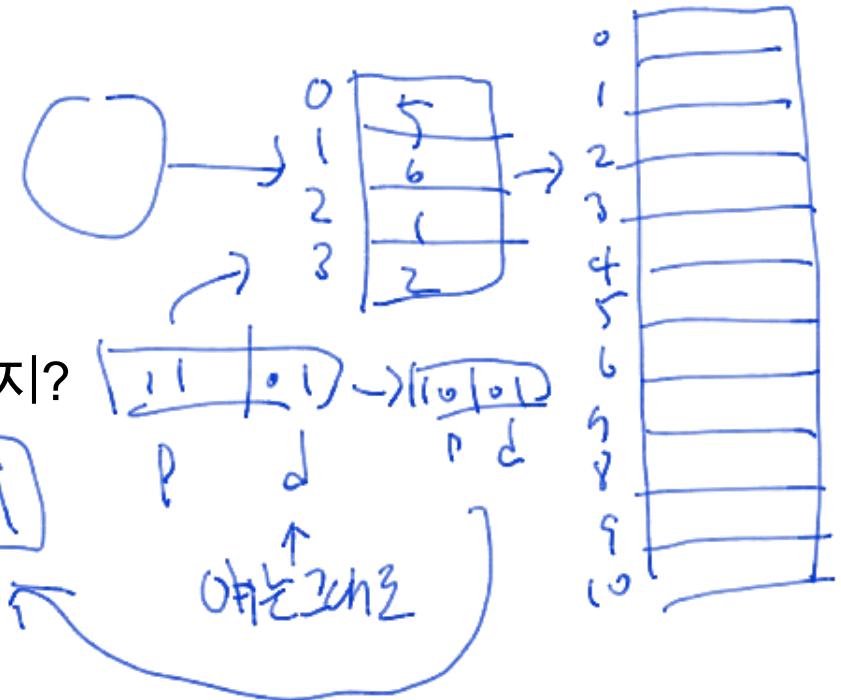
- 논리주소 (Logical address)
  - CPU 가 내는 주소는 2진수로 표현 (전체  $m$  비트)
  - 하위  $n$  비트는 오프셋(offset) 또는 변위(displacement)
  - 상위  $m-n$  비트는 페이지 번호
- 주소변환: 논리주소 → 물리주소 (Physical address)
  - 페이지 번호( $p$ )는 페이지 테이블 인덱스 값
  - $p$ 에 해당되는 테이블 내용이 프레임 번호( $\text{f}$ )
  - 변위( $d$ )는 변하지 않음
  - 변환 그림 참조

# 주소 변환 (Address Translation)

- 예제

- Page size = 4 bytes
- Page Table: 5 6 1 2
- 논리주소 13 번지는 물리주소 몇 번지?

$2^n \rightarrow$  Displacement



- 예제

- Page Size = 1KB
- Page Table: 1 2 5 4 8 3 0 6

$$2^10 \quad 3 = 110_{(2)}$$

논리주소 3000번지는 물리주소 몇 번지?

- 물리주소 0x1A53 번지는 논리주소 몇 번지?

$$101110111000 \rightarrow 101110111000$$

- 숙제!

$$\begin{array}{r} 21300 \\ \downarrow 1500 \\ 21750 \\ \downarrow 375 \\ 21375 \end{array} \rightarrow \begin{array}{r} 2187 \\ \downarrow 93 \\ 2146 \\ \downarrow 23 \\ 2121 \end{array} \rightarrow \begin{array}{r} 1211 \\ \downarrow 45 \\ 122 \\ \downarrow 10 \\ 10 \end{array}$$

$$\begin{array}{r} 00011010001010011 \\ \downarrow \\ 11100101010011 \end{array} \rightarrow 1E53$$

# 내부단편화, 페이지 테이블

- 내부 단편화 (Internal Fragmentation)

- 프로세스 크기가 페이지 크기의 배수가 아니라면,
- 마지막 페이지는 한 프레임을 다 채울 수 없다.
- 남은 공간 = 메모리 낭비

- 페이지 테이블 만들기

- CPU 레지스터로 ~~빠른지만 많이 넣을 수 없어.~~  $\rightarrow$  가변식으로 적용 X
- 메모리로 ~~많이 넣을 수 있지만 느리고~~  $\rightarrow$  신속하고 사용 (SRAM으로 만든 cache와 연결하는 용도)  
 $\rightarrow$  TLB (Translation Look-aside Buffer)로
- 테이블 엔트리 개수 vs 변환 속도
- 연습: TLB 사용 시 유효 메모리 접근 시간

$$T_m = 100\text{ns}, T_b = 20\text{ns}, \text{hit ratio} = 80\%$$

$$T = h(T_b + T_h) + (1-h)(T_b + T_m + T_o)$$

# 보호와 공유

- 보호 (Protection): 해킹 등 방지

- 모든 주소는 페이지 테이블을 경유하므로,
- 페이지 테이블 엔트리마다 r, w, x 비트 두어  
→ read, write, execute
- 해당 페이지에 대한 접근 제어 가능

- 공유 (Sharing): 메모리 낭비 방지

- 같은 프로그램을 쓰는 복수 개의 프로세스가 있다면,
- Code + data + stack에서 code는 공유 가능 (단, non-self-modifying code = reentrant code = pure code 인 경우)  
재진입 코드
- 프로세스의 페이지 테이블 코드 영역이 같은 곳을 가리키게

세그멘테이션

Segmentation

# 세그멘테이션 (Segmentation)

- 프로세스를 논리적 내용(=세그먼트)으로 잘라서 메모리에 배치!
  - 프로세스는 세그먼트(*segment*)의 집합
  - 세그먼트의 크기는 일반적으로 같지 않다.
- 세그먼트를 메모리에 할당
  - MMU 내의 재배치 레지스터 값을 바꿈으로서
  - CPU는 프로세스가 연속된 메모리 공간에 위치한다고 착각
  - MMU는 세그먼트 테이블 (segment table)이 된다.



설명  
4~6

# 주소 변환 (Address Translation)

- 논리주소 (Logical address)
  - CPU 가 내는 주소는 segment 번호( $s$ ) + 변위( $d$ )
- 주소변환: 논리주소 → 물리주소 (Physical address)
  - 세그먼트 테이블 내용: base + limit
  - 세그먼트 번호( $s$ )는 세그먼트 테이블 인덱스 값
  - $s$ 에 해당되는 테이블 내용으로 시작 위치 및 한계값 파악
  - 한계(limit)를 넘어서면 segment violation 예외 상황 처리  
*인터럽트 발생!*
  - 물리주소 =  $\text{base}[s] + d$
  - 변환 그림 참조

# 주소 변환 (Address Translation)

- 예제

Limit	Base
1000	1400
400	6300
400	4300
1100	3200
1000	4700

- 논리주소 (2,100) 는 물리주소 무엇인가?  $4400$
- 논리주소 (1, 500) 은 물리주소? ~~6300~~ 인터럽트 발생

# 보호와 공유

- 보호 (Protection): 해킹 등 방지

- 모든 주소는 세그먼트 테이블을 경유하므로,
- 세그먼트 테이블 엔트리마다 r, w, x 비트 두어
- 해당 세그먼트에 대한 접근 제어 가능
- **페이지보다 우월!** *페이지를 주소위로 자르기 때문에 다른 프로그램으로 교체해야 하는 부분이  
같이 포함될 수 있다. 그로 인해 접근*

- 공유 (Sharing): 메모리 낭비 방지

- 같은 프로그램을 쓰는 복수 개의 프로세스가 있다면,
- Code + data + stack에서 code는 공유 가능 (단, non-self-modifying code = reentrant code = pure code인 경우)
- 프로세스의 세그먼트 테이블 코드 영역이 같은 곳을 가리키게
- **페이지보다 우월!** *Segmentation을 사용하면 code 부분만 분할할 수 있다.*

# 외부단편화

- 외부 단편화 (External Fragmentation)

- 세그멘트 크기는 고정이 아니라 가변적
- 크기가 다른 각 세그멘트를 메모리에 두려면 = 동적 메모리 할당
- First-, best-, worst-fit, compaction 등 문제

- 세그멘테이션 + 페이징

- 세그멘테이션은 보호와 공유면에서 효과적
- 페이징은 외부 단편화 문제를 해결
- 따라서 세그멘트를 페이징하자! ↗ Paged segmentation
- 예: Intel 80x86

세그멘트로 분할 후 페이징 하자.  
→ 하지만 CPU→Mem 으로 가는 Segmentation Table.  
Page table 두번 거쳐나가는 속도를 느낄 때.