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
  - 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)
- 주소 구분
  - *논리주소 (logical address)* vs *물리주소 (physical address)*

메모리 낭비 방지

- Dynamic Loading
- Dynamic Linking
- Swapping

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 몰아내기
  - *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 반복 ☞ scattered holes
- 메모리 단편화 (Memory fragmentation)
  - Hole 들이 불연속하게 흩어져 있기 때문에 프로세스 적재 불가
  - 외부 단편화 (external fragmentation) 발생
  - 외부 단편화를 최소화 하려면?

연속 메모리 할당

- 연속 메모리 할당 방식
  - First-fit (최초 적합)
  - Best-fit (최적 적합)
  - Worst-fit (최악 적합)
- 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
  - *다른 방법은?*

페이징

Paging

페이징 (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(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?
- 예제
  - Page Size = 1KB
  - Page Table: 1 2 5 4 8 3 0 6
  - 논리주소 3000번지는 물리주소 몇 번지?
  - 물리주소 0x1A53 번지는 논리주소 몇 번지?
- 숙제!

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
  - 메모리로
  - TLB (Translation Look-aside Buffer) 로
  - *테이블 엔트리 개수 vs 변환 속도*
  - 연습: TLB 사용 시 유효 메모리 접근 시간

    Tm = 100ns, Tb = 20ns, hit ratio = 80%

보호와 공유

- 보호 (Protection): 해킹 등 방지
  - 모든 주소는 페이지 테이블을 경유하므로,
  - 페이지 테이블 엔트리마다 r, w, x 비트 두어
  - 해당 페이지에 대한 접근 제어 가능

- 공유 (Sharing): 메모리 낭비 방지
  - 같은 프로그램을 쓰는 복수 개의 프로세스가 있다면,
  - Code + data + stack 에서 code 는 공유 가능 (단, non-self-modifying code = reentrant code = pure code 인 경우)
  - 프로세스의 페이지 테이블 코드 영역이 같은 곳을 가리키게

세그멘테이션

Segmentation

세그멘테이션 (Segmentation)

- 프로세스를 논리적 내용(=세그멘트)으로 잘라서 메모리에 배치!
  - 프로세스는 세그멘트*(segment)*의 집합
  - 세그멘트의 크기는 일반적으로 같지 않다.
- 세그멘트를 메모리에 할당
  - MMU 내의 재배치 레지스터 값을 바꿈으로서
  - CPU 는 프로세스가 연속된 메모리 공간에 위치한다고 착각
  - MMU 는 *세그멘트 테이블 (segment table)* 이 된다.

주소 변환 (Address Translation)

- 논리주소 (Logical address)
  - CPU 가 내는 주소는 segment 번호(s) + 변위(d)
- 주소변환: 논리주소 → 물리주소 (Physical address)
  - 세그멘트 테이블 내용: base + limit
  - 세그멘트 번호(s)는 세그멘트 테이블 인덱스 값
  - s 에 해당되는 테이블 내용으로 시작 위치 및 한계값 파악
  - 한계(limit)를 넘어서면 segment violation 예외 상황 처리
  - 물리주소 = 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?
  - 논리주소 (1, 500) 은 물리주소?

보호와 공유

- 보호 (Protection): 해킹 등 방지
  - 모든 주소는 세그멘트 테이블을 경유하므로,
  - 세그멘트 테이블 엔트리마다 r, w, x 비트 두어
  - 해당 세그멘트에 대한 접근 제어 가능
  - *페이징보다 우월!*

- 공유 (Sharing): 메모리 낭비 방지
  - 같은 프로그램을 쓰는 복수 개의 프로세스가 있다면,
  - Code + data + stack 에서 code 는 공유 가능 (단, non-self-modifying code = reentrant code = pure code 인 경우)
  - 프로세스의 세그멘트 테이블 코드 영역이 같은 곳을 가리키게
  - *페이징보다 우월!*

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*