[Chapter 9]

소프트웨어융합학과/2020105742/한지훈

1. Memory Management Goals
   A. convenient abstraction, maximize performance: OS 목적과 동일(메모리 관리에 있어서)
   B. isolation between processes: 멀티 프로그래밍으로 넘어가면서 여러 프로세스가 메모리 위에 올라갔을 때 물리적으로 완벽하게 분리 필요(OS 관리 하에)

2. Memory Management by system
   A. Batch programming: Memory Management 필요 없음
      I. 이유: 한 번에 하나의 프로세스만 실행되기 때문에 메모리 관리, isolation 필요 없음
   B. Multiprogramming: 하나의 메모리 위에 여러 개의 프로세스가 있기 때문에 Memory Management 필요해짐
      I. Requirements
      - Protection: 프로세스 간에 주소를 제한하여 구분 지어야 함
      - Fast translation: 메모리에 대한 접근을 빠르게 하여 bottleneck을 없애야 함
      - Fast context switching: Protection과 Fast translation을 만족하기 위해서 필요

3. Virtual Memory(VM)
   A. 핵심 개념: 메모리는 주소체계를 가지는데, 그대로 쓰지 않고 logical/physical memory로 분리해 쓰는 개념
      I. 조건: logical address(CPU에서 알아먹는 주소)와 physical address(실제 memory 접근시 사용하는 주소)간 매핑
      II. 방법: Address binding
   B. 일부 개념: 디스크의 일부를 메모리처럼 사용 > 이론상 무한대의 virtual 메모리 구현

4. Address binding
   A. 정의: 실제 실행 시 메모리로 올라갈 때, 매핑을 어떤 방식으로(어떤 시점에서) 할 것인지
   B. 필요성: logical 메모리와 physical 메모리는 분리되어 있기 때문에 binding이 필요함
   C. Compile time
      I. 동작 방식
         짠 코드에 대해서 컴파일 타임에 binding을 함
         - 아직 실행이 되지 않은 상태이기 때문에 static 방식(정해진 상수값에서부터 쭉 넣는 방식)
         - 이때 상숫값 주소는 기존 logical 주소와 동일한 값을 가짐
      II. 문제: 어떤 디바이스에서 실행해도 고정 주소부터 할당되기 때문에 다른 할당과 겹치면 문제 발생
      III. 사용: embedded 환경
      - 이유: 기능들이 다 사전에 정의되어 있기 때문에 주소도 고정해서 사용하는 것이 편리할 수 있음
   D. Load time
      I. 동작 방식
         OS가 load 시에 적절한 비어 있는 공간의 주소값을 시작 주소값으로 설정함
         - Code 안에서 메모리 주소값 쓰는 부분(int 변수 등) 있으면 logical, physical 주소 차이만큼 더해줌
         - 즉 load 과정에서 code 딴으로 binding 된다고 볼 수 있음
      II. 장점: Compile time 방식보다는 유연하다고 할 수 있음
      III. 단점:
      - offset만큼 더해서 돌려야 하기 때문에 연산 수가 증가해서 load time이 길어짐
      - 특히 실행파일 모든 instruction에 대해 더하고 올려서 내가 실행하지 않을 코드에 대해서도 연산 필요
        - 전체 code에 대해 다 binding을 해야 해서 비효율적임
   E. Execution time

I. 동작 방식

일단 load 과정에서는 별도 연산 없이 그대로 올림

- 변수 등에 대해 연산이 이뤄질 때(실행 시에) offset을 더해줌

II. 장점:

- Load time을 줄일 수 있음
- 실행되는 code에 대해서만 binding하고, 나머지는 하지 않아서 효율적임 > 일반적으로 가장 많이 사용

III. 단점: 메모리 참조시마다 계속 더해주기 때문에 execute시 CPU가 할 일이 증가하여 오버헤드가 커짐

- 해결: MMU(Memory-Management Unit)을 통해 오버헤드를 줄일 수 있음(swap)

5. Logical (Virtual) Address Space

A. 개요: code나 debug시에 우리가 보는 주소는 전부 logical address(상대주소)임

I. 이유: 이미 Virtual memory가 적용되어서 개발자는 physical 주소 알 수 없음

6. Contiguous Allocation

A. 개요: logical 주소를 physical 주소로 할당할 때, 기준 주소부터 쭉 쓰는 방식

I. 동작 방식:

CPU가 MMU에게 메모리 사용요청을 하여 logical 주소를 넘겨줌

- MMU는 logical 주소에 대해 limit register로 protection 걸어줌(가능 범위 내의 주소인지)
- no라면 trap 걸어서 메모리 보호
- yes라면 넘겨받은 logical 주소에 relocation register(기준주소)를 더한 physical 주소로 메모리 접근

II. 문제: Hole이 발생하고, 이에 따라 External fragmentation이 발생

- 발생 과정:

OS 로드 후 프로세스들의 실행, exit이 계속 일어남

- Contiguous Allocation에서는 위에서부터 순차적으로 할당되기 때문에 사이에 공간 생김(hole)
- Ex) 총 공간 2mb 남아도 연속된 공간 최디 크기가 1mb라서 1.1mb인 프로세스11 실행 불가
- Hole 합산 크기보다 작은 프로그램도 실행 불가능해지며, 가면 갈수록 심해짐
- External fragmentation 발생

- 해결법 1: 3가지 방법
  - 1. First-fit: 프로세스 실행가능한(들어갈 수 있는) 가장 첫 hole에 할당
  - 2. Best-fit: 프로세스 실행가능한(들어갈 수 있는) hole 중에서 가장 작은(가장 비슷한)
  - 3. Worst-fit: 프로세스 실행가능한(들어갈 수 있는) 가장 큰 hole에 할당

    특징: hole 탐색 위해서는 hole 정보 manage 필요한데, hole 정보는 프로세스 실행/종료시 변경됨
  - 관리가 어렵기 때문에 3가지 방법 모두 안 씀
- 해결법 2: compaction
  - 방법: 주기적으로 모든 hole을 하나로 합치기

    특징: I/O 관점에서 data를 계속 옮겨야 하고 PCB 정보 등을 업데이트해야 함
  - Bottleneck 발생
  - External fragmentation을 해결할 수 있지만 side effect인 bottleneck 때문에 안 씀

7. Paging

A. 개요: Contiguous Allocation은 연속공간을 찾아야 하고, 약 4GB의 linear 공간을 다 관리해야 하는 문제

- Logical memory와 Physical memory를 같은 단위로 나눠 사용하여 External fragmentation 방지

B. 용어

I. Logical memory의 한 discrete 단위: Page

II. Physical memory의 한 discrete 단위: Frame

C. 동작 방식

I. Physical memory의 frame들은 서로 섞여있기 때문에 logical에서 physical로의 매핑이 필요함

- OS딴에서 page mapping table로 관리를 하며, page 크기는 보통 4096bytes
- 만약 한 프로세스가 3900byte 차지하면 page가 1개 필요하며, 196byte 남음
- Internal fragmentation 발생(page 여러 개 먹는 상황에서도 마찬가지)

II. 문제: Internal fragmentation 발생

- 발생 과정:

  만약 한 프로세스가 3900byte 차지하면 page가 1개 필요하며, 196byte 남음
  - Internal fragmentation 발생(page 여러 개 먹는 상황에서도 마찬가지)
- 특징:
  - external fragmentation에 비해 덜 심각함(page 안에서 fragmantion은 별 문제 X)
  - 실제로 쓰이는 것은 paging
  - 정도: page size에 따라 달라짐 > 최적: 32bit 컴퓨터에서는 4096byte

8. Address Translation

A. 과정:

- CPU는 logical 주소를 알고 있음 – (p, d): p = page number(몇번 페이지) / d = offset(페이지 안 주소)
  - MMU는 p 먼저 들고 와서 page table에서 frame number(f)을 읽음
  - Physical 주소 – (f, d)로 메모리에 접근

B. Page tables

I. 개요: OS가 관리하며, page number을 frame number로 매핑함

II. 특징: virtual address space에서 각 page당 한 개의 page table entry를 가짐

- 의미: logical memory에서는 memory를 0~max까지 다 쓸 수 있다고 가정
  - Page entry 개수(Page 쪼개는 개수): 약 1m개(4GB/4096bytes
  - 모든 프로세스가 1m개의 page가 있다고 가정
  - 프로세스 마다 page table entry 개수 약 1m개 가짐

9. Paging Example – 조건 주면 계산 가능해야 함

A. 가정:

- Logical address: 32 bits(32bit CPU), 8칸
- Physical address: 20 bits, 5칸 > 용량: 2^20 = 1mb(아주 작음)
- Page size: 4kb(4096byte)

B. 과정

Logical address(32 bits)는 page number과 offset로 이루어짐

- 4096byte = 2^12이라서 page table의 크기는 12bit이고, 이는 곧 offset을 의미함(0~4095)
- 남는 20bit에 page number를 표현
- 2^20개의 page 관리 가능(1m개)
- Page number는 page table의 index 값이며, index 위치의 frame number을 읽음
- 2칸의 frame number과 기존 3칸의 offset가 결합되어 5칸(20bit)의 physical address 생성
- Physical 주소로 메모리 접근

I. 만약 physical address도 32bit하면 2^32 = 4GB > 32bit 시스템에서는 최대 4GB 인식됨

10. Free Frame list

A. 정의: 현재 비어있는 frame들의 list로, OS가 관리

B. 동작 방식

page들을 frame으로 매핑할 때, memory allocation 정책에 따라 free frame list에서 필요한 수의 frame 할당

- 그에 따른 page table 생성

11. Page Table 구현

A. 개요: OS가 관리하며, 실행 중인 프로세스의 수만큼 존재

B. 구성

I. PTBR(Page-table base register): page table의 base address를 가지고 있음

II. PTLR (Page-table length register): page table의 크기를 가지고 있음(page table 크기는 1M 고정 X, 최적화)

C. 특징:

Page table은 memory에 있기 때문에 memory에 access 필요

- 실제 data에 접근하기 위해서 memory에 또 access 필요
- Paging은 Contiguous Allocation과 달리 memory에 2번 접근 필요(two memory accesses)
- 성능이 감소하기 때문에 해결이 필요함(TLB)

12. TLB(Translation Look-aside Buffer, Associative Memory)

A. 정의: 캐싱을 효율적으로 사용하여 page table 정보 일부를 캐시로 빼놓고 빠르게 access

B. 위치: MMU 안에 있음

C. 동작 방식:

MMU가 CPU로부터 logical 주소를 받아 변환하려 할 때, TLB를 먼저 봄

- 만약 TLB에 page 정보 있으면 바로 access
- TLB에 정보 없으면 메모리로 가서 정보를 TLB에 업데이트하고 TLB로 다시 돌아가서 access
- 1번 혹은 2번의 메모리 접근을 하면서 계속 반복

I. 성능 개선: 꽤 있음

- 이유: linear/binary search로 한번에 비교 처리 불가한 일반 memory와 달리, TLB(associative memory)는 인덱스 넣으면 frame이 바로 튀어나옴
  - Hit 발생했을 때 memory 접근 1번보단 살짝 느리지만, 2번보단 훨씬 빠름

D. Locality: 캐싱이 유효한 이유

I. Temporal locality: 가장 최근에 사용한 data가 가장 가까운 미래에 사용될 확률이 높음

II. Spatial locality: 지금 접근 중인 영역의 근처에 있는 data들이 가장 가까운 미래에 사용될 확률이 높음

- 주변 영역도 뭉텅이로 가져와 별도 disk access 없이 바로 접근 가능

E. Hit rate

I. 정의: 들어오는 page entry 중에서 TLB에 page 정보가 있어 바로 access하는 것의 비율

II. 특징: 보통 99% 이상이라서 사실상 memory 1번 접근하는 것과 속도 비슷함

III. 놓치는 경우:

- HW: PCB 안의 page table을 HW로 바로 접근(mode change X)
- OS: OS의 요청으로 접근(유저 -> 커널로 mode change가 일어나 오버헤드가 커서 잘 안씀)

F. Context switch 시

I. 과정:

- Page table에는 Valid bit(유효한 entry인지 여부)가 있음
  - Context switch가 일어나면 MMU는 기존 프로세스의 Valid bit를 전부 invalid로 변환
  - 프로세스가 처음 올라가면 memory 2번 접근 필요(cold)

- 2번째 이후부터는 99% hit rate 덕분에 거의 1번 접근하는 시간과 비슷

II. 성능 개선: 처음 올라갔을 때 좀 더 빠르게 동작하기 위해 HW에 page table 일부 넣어서 빠른 접근(CR3)

G. 꽉 찼을 때

I. 동작: 덮어쓰기 필요 > 정책에 따라 동작(LRU, least recently used): access 오래된 애부터 덮어쓰기

- 특징: temporal locality와 반대 개념

13. EAT(Effective Access Time)

A. 공식: EAT = $(1 + \varepsilon)\ \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$

I. ε = TLB 접근 시간

II. α = hit ratio

III. 1, 2: memory access 1, 2회 접근 시간

IV. (1 + e) a: hit가 났을 때 상황 > memory access 1회 시간(1) + TLB 접근 시간(ε), hit 발생 확률(α)

V. $(2 + \varepsilon)(1 - \alpha)$: hit 나지 않았을 때 상황 > memory access 2회 시간(2) + TLB 접근 시간(ε), hit 발생 X 확률(1-α)

B. 계산 예시(조건 주면 계산할 수 있어야 함)

I. α = 99%, ε = 20ns, memory access 시간 = 100ns

- EAT = (100+20)x0.99 + (200+20)x0.01 = 121ns

14. Page Table Entries (PTEs)

A. 정의: page table의 한 행

B. Valid bit (V): 1bit, PTE(page table entry)가 유효한지 여부 확인

I. 사용 이유: protection(page entry가 invalid하면(사용하지 않는 entry이면) fault 등의 error 발생)

C. Reference bit (R): 1bit, 해당 frame에 access(read/write)한 적 있는지 여부

D. Modify bit (M): 1bit, 해당 frame에 write한 적 있는지 여부

E. Protection bits (Prot): 2bit, 상태값(Read, Write, Execute) 등으로 권한 관리(ex read only, …)

F. Frame number (FN): 20bit, Physical memory의 frame 위치를 저장

G. 총 용량: 25bit이지만 32bit system이기 때문에 7bit 채워서 그냥 32bit로 사용함

15. Page Table Structure

A. 개요:

I. 프로세스 1000개 돌아가면 page table 용량이 4GB가 되어 낭비가 큼

II. Valid 상태의 프로세스 수는 생각보다 적음

- 오버헤드를 줄이는 방법을 찾아야 함

B. Hierarchical paging

I. 동작 방식: page table을 outer과 inner 두개로 나눠서 관리

- Outer page table: 1024의 길이를 가지며, 각각의 포인터는 inner page table 1024개 entry를 관리
  - 포인터가 가리키는 1024개 entry중 1개라도 valid면 포인터도 valid, 없으면 invalid + 가리키지 않음
- Inner page table: 기존 1M개 table을 1024개씩 묶음으로 나눔

II. 최악의 경우: outer table(4kb) 다 valid라면 inner table 전체(4mb) 다 써서 기존에 비해 4kb 오버헤드 발생

III. 최선의 경우: outer table(4kb) 1개만 valid라면 inner table 1개(4kb)만 써서 전체 8 kb만 있으면 됨

- 유효한 이유: 프로세스 1개가 page table 꽉 채워 쓰는 경우 거의 없음 > spatial overhead 줄일 수 있음

C. Hashed page table

I. 동작 방식: page table 자체를 hashing을 사용하여 줄임

- 1m개의 entry를 1024개로 줄일 수 있지만, 해쉬값 동일한 2개 이상 valid entry 있으면 살려야 함
  - Linked list로 관리

II. 단점:

- Hash 자체 연산이 복잡해지면 HW상 구현이 어려움
- Linked list는 virtual memory 도입에 부적절함 > 잘 쓰지 않음

D. Inverted page table

I. 동작 방식: OS가 관리하는 page table은 1개만(frame table) 가지며, 엔트리 수 = physical memory 엔트리 수

II. 단점: virtual memory 도입에 부적절함 > 잘 쓰지 않음

16. Shared Pages

A. 개요: page의 또다른 장점으로, code segment 등 중에서 read only(수정 불가)인 영역들은 physical memory로부터 공유하여 사용하여 memory를 아낄 수 있음

I. 활용: library 등도 공유하여 사용할 수 있음

17. Paging 종합정리

A. 장점

I. physical memory의 할당이 쉬움(free list로부터 할당 가능)

II. external fragmentation이 없음(Contiguous Allocation시 발생하는 hole이 없음)

III. page의 protection 가능(page table의 valid/invalid bit를 사용하면 됨)

IV. Shared Pages 가능(read only인 영역들을 공유하여 memory 절약)

B. 단점

I. internal fragmentation 존재(장점 대비 감수 가능하며, 사실상 해결 못하는 유일한 단점)

II. memory에 2번 접근 필요(page table 확인 1번, 실제 data 확인 1번) > 해결: TLB(AM) – hit ratio > 99%

III. page table 크기 커질 수 있음 > 해결: Hierarchical page table

18. Segmentation

A. 개요: 1개의 page에 code segment와 상수 영역 등이 혼합되어 있으면 문제가 생기기 때문에 분리해줘야 함

I. 특징: contiguous와 동작 방식 비슷함

II. 장점: segment 보호/공유 용이하고, internal fragmentation 발생하지 않음

III. 단점: external fragmentation(hole) 발생 > 해결: paging 개념과 segmentation을 결합

B. Segmentation with Paging

I. Hybrid approach: segmentation 하고 나서 paging(internal 생기지만 external 제거 가능)

II. Multiple page size: page 크기를 가변적으로 사용하여 혼합 안생기게 하기

III. 동작 방식:

- Logical address(segment 기반)에서 offset를 계산하여 linear address 위치 확인
  - Linear address(page 기반)에서 Hierarchical 방식으로 physical address에 접근