[Chapter 10]

소프트웨어융합학과/2020105742/한지훈

1. Virtual Memory
   A. 개요: 이론적으로 메모리를 무한대로 사용할 수 있는 방식
      I. 실제: 불가능함(physical memory 크기는 정해져 있음)
      II. 방법: storage(디스크) 일부를 메모리처럼 처리하게 하여 가상으로 더 큰 메모리 가진 시스템처럼 운영 가능

2. Demand Paging
   A. 개요:
      I. 'demand' 이유: 필요할 때(요구에 따라) paging을 함
      II. 코드 처음 실행 시 느린 이유: 캐싱 안되서도 있지만, 애초에 느린 경우도 있음
         - 느린 이유: 처음 실행시 logical memory만 만들고, page table entry는 다 invalid임
         - 다 invalid인 이유: 내가 사용할 기능들만 load하기 위해(두번째부터는 좀 빨라짐
         - 실제 실행시 invalid이면 두가지 경우로 나뉨(<B. 동작 방식> 에서 기술)
   B. 동작 방식 – 전 과정 설명할 수 있어야 함
      1. MMU가 CPU로부터 logical 주소를 받아 변환하려 할 때, TLB를 먼저 봄
         - TLB Hit 나면(TLB에 page 정보 있으면) 바로 physical memory access 가능
         - TLB Hit 안나면(TLB에 page 정보 없으면) page table 참조
      - page table 참조시
         - 참조결과 page가 valid이면(physical memory에 올라가 있는 상태) TLB 업데이트 후 1로 돌아감
         - 참조결과 page가 invalid이면 MMU interrupt가 발생하여 OS로 mode change가 일어남
      - OS는 invalid의 이유를 확인함(protection fault/page fault)
         - 이유가 protection fault라면 디스크에도 없어서 m에 올라와 valid로 바뀔 여지가 없음
           -> kill해주면 됨
         - 이유가 page fault라면 page가 디스크에는 있어서 valid로 바뀔 여지가 있음
           -> page를 secondary storage(디스크)에서 physical memory로 올림(Swap In)
           -> Page table을 valid로 바꿔주고, TLB도 업데이트 한 후에 다시 1로 돌아감
           -> 재접근 시 valid로 바뀌어있고, TLB가 업데이트 되어있어서 접근할 필요가 없음
   C. 성능 확인: Locality(캐싱 성능 = Demand Paging 성능)
      I. Temporal locality: 특정 영역 계속 access 됨(반복문 실행 비율 > 80%라서 유효)
      II. Spatial locality: 최초 접근 memory 주변 memory가 계속 access 됨

3. Copy-on-Write
   A. 개요: write가 일어났을 때 새 frame을 할당해줌(리눅스의 개념, page sharing과 비슷)
      I. Right after fork(스케줄링 전): 두 프로세스가 완전 동일하여 다른 frame 할당할 필요 없음
      II. 프로세스 1이 page C modify(write)시: C 내용 달라지기 때문에 그때 되서 다른 frame 할당해줌
         - memory 아낄 수 있음

4. Page Replacement
   A. 필요 이유:
      Demand paging은 code, data 등 다 올릴 때에 비해 physical memory 훨씬 효율적으로 사용 가능
      - But physical memory 한정 > 포화 상태에서 disk에 있는 프로세스를 올려야 할 때(page fault) 문제 발생
      - 하나를 내리고(Swap out) 하나를 올려야 한다(swap in) > Page Replacement: 성능 저하에 큰 영향
      - 누구를 내릴지는 Page Replacement algorithm이 결정

5. Page Replacement algorithm

A. 가장 중요한 고려사항: 누구를 내릴 것인지(best victim)

I. 방법: locality 개념 뒤집기(가장 access 된지 오래된 애는 다시 사용될 확률 아주 낮음)

B. 구현: OS 상에서 구현됨

C. Demand paging 성능: EAT(Effective Access Time)

I. 계산: EAT = (1 – p) x memory access + p x (page fault overhead + swap out + swap in + restart overhead)

- p = page fault 발생확률
- EAT = memory access하는데 걸리는 시간

D. 목표: p 줄이기

I. 이유: page fault가 발생하지 않게 하여 Page Replacement Algorithm이 최소한으로 수행되도록 하기 위해

II. 방법: 프로세스 별 할당 프레임 수 상한 높이기

- 한계: 프로세스 당 프레임 수 4개 이후부터는 가성비 매우 떨어짐
- 이유: locality가 잘 적용되려면 최개 4개 frame 정도가 적절(예외 존재)

6. Algorithm 종류

A. FIFO

I. 개요: 가장 이전에 들어온 프로세스를 내리기 – P.26(6번)

II. 한계: "Belady's Anomaly"(단순 FIFO 사용 시 성능 낮음, locality 개념 뒤집었을 때 성능 높음을 증명)

- 성능이 낮음 = page fault가 많이 남

[Optimal]: 구현 불가능(분기에 따라 달라지기 때문에 fault 최소화될 수 있게 예측은 불가능 > 근사

B. LRU(Least Recently Used)

I. 개요: locality 개념 뒤집기(가장 access 된지 오래된 애는 다시 사용될 확률 아주 낮음) – P.29(8번)

II. 구현:

- 시간정보 필요(실수 X, counter – memory access 될때마다 access 프로세스는 0, 나머지는 +1)
  - Demand paging entry page마다 다 counter 정보 필요하고, TLB에도 필요
  - Bit 수 많아지면 성능은 좋아지지만 메모리 효율이 낮아짐
  - 대안으로 stack 사용해도 stack 구현 자체에 연산상 오버헤드 존재
  - 그래도 필요하기 때문에 대략적으로 표현하는 방식 사용(LRU Approximation Algorithms) - 3가지

III. Reference bit:

- 방식: 초기값 전부 0, 접근시 1로, 주기적으로 다 0으로 포기화(일정한 time quantum마다)
  - 0인 프로세스들 중에서 victim 선정
- 문제: refresh 주기가 길면 0 중 누가 가장 오래된 프로세스인지 몰라 random 선택 > 효율 낮음

IV. LRU Clock / Second chance

- 방식: 포인터를 두고 reference bit = 1이면 기회 한번 줌(0으로 바뀌고 넘어감)
  - 순차적으로 0 설정하고, reference bit = 0이면 victim으로 선정
- 문제: random보다 좋지만 포인터/커널 오버헤드(탐색 복잡도 N > victim 선정 시간 커짐) 문제 있음

V. NRU(Not Recently Used) / Enhanced second chance

- 방식: R(Reference bit - read)와 M(modify bit - write) 사용하여 4가지 case로 구분 – P.33
  - Read 하면 R: 0 > 1 / interrupt 하면 R: 1 > 0
  - Write 하면 R: 0 > 1, M: 0 > 1 / interrupt 해도 M은 1 > 0으로 돌아가지 않음, R만 변경
- 우선순위: 후순위 ~ (R=1 M=1) > (R=1 M=0) > (R=0 M=1) > (R=0 M=0) ~ 최우선
- 보통 write 한번이라도 된 프로세스가 locality 특성을 더 많이 가짐

- 특징: 실제로 사용되는 방식
- 장점: 이해와 구현이 간단하며, 준수한 성능을 보여줌

C. LFU(Least Frequently Used)

I. 개요:

시간 정보 뿐만 아니라 단위 시간당 사용된 횟수도 함께 확인하기

➢ 단위 시간당 많이 쓰이지 않은 프로세스를 victim으로 선정

II. 한계:

- LFU 동작하려면 count 정보를 저장해야 해서 size가 커지게 됨, shifting 작업도 오버헤드가 됨

➢ 연산량을 줄이고 공간을 효율적으로 사용하기 위해 approximate하게 됨(1bit > LRU와 같아짐)

7. PTE: page fault 발생시에만 OS가 확인하고 보통은 CPU의 HW가 확인하기 때문에 CPU HW 스펙 dependency 큼

8. Allocation of Frames

A. Allocation algorithms

I. 개요: 프로세스들 간에 frame을 어떤 방식으로 나누어줄 것인가

II. 종류

- Equal allocation: 그냥 프로세스마다 똑같은 개수만큼 할당
- Proportional allocation: 비율에 따라 할당(더 큰 프로세스에게 더 많은 프레임 할당)
- Priority-based allocation: 스케줄링 우선순위 높으면 많이 할당(CPU, page에 많이 access 가정)

III. 실제 사용: Priority-based allocation + Proportional allocation(우선순위 기반, 하지만 크기도 무시할 수 없음)

B. Page replacement

I. 개요: 할당 frame을 다 썼을 때 victim 선정 범위를 어떻게 설정할 것인가

II. 종류

- Global replacement: 전체 frame을 대상으로 victim을 선정
- Local replacement: 돌고 있는 해당 프로세스 안의 frame을 대상으로 victim을 선정

III. 실제 사용: Local(Global은 내가 뺐어온 frame이 다음에 올라갔을 때 등의 manage 상의 문제가 생김)

9. Page-Fault Frequency Scheme

A. Page-fault rate: 특정 프로세스에 대해서 들쭉날쭉일 수밖에 없음

I. Ex1: 반복 5만번, page 크기와 같은 크기의 배열 접근 > 같은 위치 계속 접근하여 page fault 적음

II. Ex2: 반복 5만번, page 크기보다 훨씬 큰 배열 접근 > 접근 위치 계속 달라져서 page fault 많음

B. 목표: 최적의 page-fault rate 유지하기 위해 rate가 upper/lower bound 안에서만 움직이게 유도

C. 방법: 프로세스별 할당 frame 수를 상황에 따라 늘렸다 줄였다 하여 조절

I. 실제 rate > upper bound: 그 프로세스에 할당된 frame 대비 locality가 크다고 판단

➢ Frame을 추가로 더 할당해주어 rate 줄여줌

II. 실제 rate < lower bound: 그 프로세스에 frame을 너무 많이 줬다고 판단

➢ Frame을 한 개씩 뺐어 rate 높여줌

10. Thrashing

A. 개요: 프로세스 수가 많아지면 CPU utilization도 비례하여 증가하여 100% 찍을거라 예상

➢ But 실제로 그렇지 않고, 특정 지점에서 CPU utilization은 급격하게 감소

I. 이유: Thrashing(남은 frame 없는 지점에서 프로세스 또 추가했을 때 replacement 수 급증(swap in/out)

II. 발생 조건: 모든 프로세스의 locality 총량이 전체 메모리 크기보다 커지는 지점

11. Working-Set Model

A. 개요: locality의 총량 개념을 수치적으로 구체화한 개념

B. 방식: 시간 delta동안 access된 frame 수를 확인하여 높으면 bad locality, 낮으면 good locality(page fault 적음)

I. 낮은 delta: locality의 특성을 파악하기 어려움

II. 무한대의 delta: 전체 page의 locality를 보는 것이기 때문에 의미가 없음

- Delta값을 적절하게 조절하는 것이 중요

12. Other Considerations

A. Prepaging: 한번 page fault 발생시 frame을 여러 개 올려서 page fault 여러 개 skip하여 효율 증가

B. Page size selection

I. Fragmentation: internal framentation 줄이기 위해서는 page size 작은게 좋음

II. Page table size: entry 개수가 감소하여 Page table size가 줄기 때문에 page size 큰게 좋음

III. I/O overhead: 여러번보다 한번에 많이 읽어야 오버헤드가 줄기 때문에 page size 큰게 좋음

IV. Locality: spatial locality를 확보할 수 있기 때문에 page size 큰게 좋음

- Page 크기는 4 kbyte가 적절

C. TLB Reach

I. 정의: TLB에 들어가 있는 page table의 엔트리로 커버할 수 있는 범위

II. 중요 이유: TLB reach를 높이면 performance가 증가하게 됨

III. 계산법: TLB Reach = (TLB Size) X (Page Size)

IV. 높이기 위한 방법:

- Entry가 가리키고 있는 실제 page/frame 크기 커지면 TLB 하나 엔트리에서 접근가능한 정보량 커짐
- TLB Reach 높이기 위해 TLB size/ Page Size를 가변화하기도 함

D. Program structure

I.
```
for (j = 0; j < 1024; j++)
    for (i = 0; i < 1024; i++)
        A[i][j] = 0;
```
- 최악 page fault 개수 = 1024*1024
- 이유: 원소 하나 접근할 때마다 다른 page에 접근하기 때문에 page fault 계속 발생, 비효율적

II.
```
for (i = 0; i < 1024; i++)
    for (j = 0; j < 1024; j++)
        A[i][j] = 0;
```
- 최악 page fault 개수 = 1024
- 이유: 하나의 page(4kb)안에 data들이 4byte씩 차곡차곡 쌓이기 때문에 locality 우수, 효율적

E. I/O Interlock

I. I/O 장치의 direct memory access(DMA)시 memory가 virtual memory로 관리되고 있음을 모름

- physical memory 쓰기 때문에 OS가 DMA용 버퍼가 쓰이는 physical memory 영역 swap 안되게 잠금