Memory Management (1/2)

Contents

- 주소 공간 (Address Space)

  proccess는 Address Space에서 동작한다.

  스케줄링은 proccess단위로,

  excution은 Address Space위에서 작동한다.

  code나 data나 전부 Address Space 위에 있기 때문이다.

- 물리 주소와 가상 주소 (PA and VA)
- 가상 메모리
- Paging
- Translation Look-aside Buffers(TLB)
- 다양한 paging table

서론

메모리 위에 여러개의 프로그램이 올라간다.

여기서의 메모리는 PM였다.

프로그램 1, 2, 3이 있다고 할 때,

2가 내려가고 4가 올라간다고 할 때,

|2|>|4|이면 문제 없이 작동하지만

|2|<|4|이면 못올라가지 않냐? 이럴 때 어떻게 하면 되나?

- 범용 컴퓨터 시스템의 목적
  - CPU의 활용률을 극대화(utilization)
  - 사용자에게 빠른 응답을 제공

    - 보다 많은 프로그램을 메모리에 올려서 실행시킴 (multi-programming)
    - 여러 프로그램을 동시에 실행시키기 위한 스케줄링 기법 등장

- 메모리 관리의 필요성 등장
    - 여러 프로그램이 동시에 메모리에 적재되어 실행되면서, 메모리를 공유 할 필요가 생김

      → 공유? → dynamic link library같은 것
    - 컴퓨터의 메모리는 한정된 자원
        - 실행하는 프로그램이 많아지면 메모리 요구량이 증가

주소공간

232개의 서로 다른 주소→ 2^32개의 서로 다른 주소

- 32bit 주소 버스를 가진 시스템이 주소 1개당 1Byte의 메모리를 접근할 수 있다면, 이 시스템이 address할 수 있는 주소 공간의 크기는 몇 Byte 인가?

  주소 1개 당 1Byte의 메모리

  주소 2^32개 → 2^32Byte의 메모리

  = 2^22KB의 메모리

  = 2^12MB의 메모리

  = 2^2GB의 메모리

  = 4GB의 메모리

  주소 1개당 2Byte의 메모리

  → 이 시스템이 address할 수 있는 주소 공간의 크기 = 8GB

추가 설명

주소 공간은 virtual address의 크기와 같다.

가상 주소 공간은 address space안에서의 주소를 의미한다.

virtual address가 어디까지 커질 수 있는가? 주소 공간의 크기까지 커질 수 있다

주소 공간의 크기는 CPU의 주소 버스(address bus)의 크기에 의존한다.

address line의 갯수에 따라서 달라진다.

예를 들어서 address bus = 32이면, 주소 공간의 크기는 2^32bit = 4GB인 것이다.

2^32 - 1이 가장 큰 주소이다.

virtual address의 크기는 주소 공간의 크기로 제한되고

주소 공간의 크기는 CPU의 주소 버스(address bus)의 크기에 의존한다.

만일 address bus = 16이면, 주소 공간의 크기는 2^16bit이상 쓸 수가 없다.

물리주소와 가상주소

- 물리 주소 (physical address)
  - 컴퓨터의 메인 메모리를 접근할 때 사용되는 주소
  - 기억 장치의 주소 레지스터에 적재되는 주소

    CPU가 메모리에 쓸 때는 물리주소를 사용한다.

PA는 커널이 알고 있다.

사용자들은 전부 VA이다.

- 가상 주소 (logical address or virtual address)
  - 프로세스의 관점에서 사용하는 주소
  - CPU 관점의 주소는 물리 주소도, 가상 주소도 될 수 있음
    - 어느 메모리 모델을 사용하느냐에 따라 달라진다.
  - Logical 이기 때문에 주소공간을 의미 있는 단위로 나누어 사용할 수 있음
    - Intel은 주소공간을 segment단위로 나누어 사용

초창기 컴퓨터의 주소 관리

- 물리주소를 Compile time에 생성
  - 프로세스가 물리메모리에서 실행되는 주소를 compile time에 알아서 절대 코드를 생성한다.

컴파일하면 물리주소가 나왔음

- 시작 주소의 위치가 바뀔 경우에는 다시 compile을 해야 한다.
- 예) .COM: DOS binary format

- 다양한 프로그램이 실행됨에 따라 compile time에 물리 주소를 정하기가 어려워짐
  - 1개의 프로그램이 실행될 경우 문제가 없음
  - Multiprogramming의 경우, compile time시에 결정된 주소는 다른 프로그램과 같이 메모리에 적재하기 어려움
    - 논리 주소를 생성하기 시작함

Compile 및Link time에서의주소

- Compile time
  - Compiler가 symbol table을 만들고 주소는 symbol table relative한 주소로 이루어짐.

    ab라는 symbol의 주소를 적어 놓은 것이 symbol table이다.
  - 컴파일된 object 파일은 주소 0부터 시작함 (relocatable)

    → 즉, 어떤 수든지 더해주면 바뀌기가 쉽다

complie 결과로 만들어진 object들은 file relative한 파일들이다.

- Link time
  - object 파일들과 시스템에서 제공하는 library들을 묶어서 symbol table에 의존적이 아닌 주소를 만들어 냄 (address resolution)
  - Link의 결과로 하나의 executable파일이 만들어진 주소는 0 부터 시작함
  - Executable은 하나의 주소 공간으로 0 부터 시작함

linker의 결과로 만들어진 파일들은 0으로 시작한다

교수님 질문

Q. Link를 하게 되면 address resolution을 다 하게 됨

그런데 address resolution이 안되는 것이 있다.

그것이 무엇이겠는가?

A. DLL이야기를 했었다. 예를 들어서 보면 printf의 주소는 excutable 파일에 들어 있지 않다.

이것은 run time의 주소가 이를 해결해야 할 것이다.

물론 dynamic linking이 아닌, static linking을 쓴다면 excutable 파일 내부에서 다 해결 된다.

Load 및 실행시의 주소 결정

- Load time
  - 프로그램의 실행을 위해 loader는 executable을 메모리로 load한다.
  - 주소공간 전체가 메모리에 올라간다면, load시에 물리 주소에 대한 binding이 일어난다.

    linking에서는 VA를 만들어 주었다.
    - 프로그램은 relocatable 주소로 되어 있기 때문에 base register를 통해서 물리 주소로 바꾸어 실행하게 된다.
  - 만일 프로그램의 시작 주소를 바꾸려면, 다시 load를 해야 한다

    사실 요즘은 이런 일이 잘 발생하지는 않는다.

프로세스를 만들어주면서 PA와의 binding이 생긴다는 것이다.

- Execution time
  - 프로세스가 실행될 때 물리 주소가 바뀌는 경우, 물리 주소에 대한 binding은 프로세스가 실행될 때 일어난다.

    프로세스가 실행되는 중간에 주소가 바뀌기도 한다.
    - paging이나 swapping을 통해서 프로세스가 올려지는 메모리의 물리 주소는 바뀔 수 있다.

      paging이나 swapping을 통해서 PA를 내보내는 경우가 있다.

      메모리가 부족할 때 프로세스 일부는 swap space해버리는 것이다.

page out되면 더 이상 PM에 존재하지 않게 된다.

swapping으로 다시 메모리에 올라갔을 때는 이전의 주소를 그대로 쓰리라는 보장이 없기에 이렇게 표현한다.

- 이러한 형태의 주소 결정 방법을 사용하기 위해서 MMU와 같은 특별한 하드웨어가 필요하게 된다.

  MMU를 사용하면 좀더 다이나믹하게 주소가 변경된다.
- 대부분의 general-purpose 운영체제에서는 이 방식을 사용

초창기 컴퓨터들은 complie하면서 PA가 정해졌는데,

요즘은 PA가 변경되는 시점이 점점 더 뒤로 가고 있다.

CPU에서의 주소 변환 방법 - 초기

- CPU에서 physical relative address를 사용하는 경우
  - 프로그램 내 instruction들의 주소를 시작 주소(base address) 부터의 상대적인 offset으로 표현한 방법
  - 시작 주소가 결정되면 시작 주소 + 상대 주소의 합으로 절대 주소를 생 성할 수 있다.

초창기에는 프로그램 전체를 가져와서 relocation register를 더해서 메모리에 올려버린다.

Text는 read-only이고, data는 read-write를 다 했었다.

완전히 다르기에 영역을 다르게 표시해야 했다.

Text는 14000, data는 20000을 더해주면 Text와 data를 다른 영역에서 접근하는 것이 가능해진다.

CPU가 제공하는 메모리 주소를 바로 메모리에 넣지는 않는다.

relocation register에서 주소에 숫자를 더해준다.

relocation register의 값을 언제 바꿔줄까?

→ 프로그램이 1번을 수행을 하다가, 2번을 수행을 하게 될때이다.

→ 그것이 context swicthing을 하는 효과가 난다.

메모리 메니징과 스케줄링을 같이 생각해라

CPU에서의 주소 변환 방법

MMU가 나온 이유?

CPU에서 나오는 주소와 메모리로 접근하는 주소가 같아서는 안된다.

그러면 multiprogramming, time-sharing이 안된다.

그래서 translator가 필요한 것이다.

- Translation의속도가 중요한 요소가 된다
  → 하드웨어를 사용해야 한다.

단순히 값을 더하는 것만으로는 가상메모리를 지원 할 수 없다.

MemoryManagement Unit (MMU)

MMU는 CPU내부에 존재한다.

MMU가 번역을 해서 그 나온 주소를 bus에 태워서 메모리에 보낸다.

가상메모리

- 정의
  - 실제 존재하지는 않지만 사용자에게 메모리로서의 역할을 하는 메모리 (virtual)
- Basic Idea
  - 프로세스가 수행 되기 위해서 프로그램의 모든 부분이 물리 메모리 (physical memory)에 있을 필요는 없다.
    - 현재 실행되고 있는 code/data/stack 부분만이 (전체가 아니라) 물리 메모리에 있으면 프로세스는 실행이 가능

물리 메모리가 1MB인데, 10MB짜리 프로세스를 돌릴 수 있느냐?

→ 가능하지 않다고 생각할 수 있다.

→ 이것을 가능하게 하는 것이 가상메모리이다.

→ 어떻게 이것을 가능하게 하는것인가?

교수님의 질문

가상 메모리가 필요한가? 왜 필요한가?

throughput을 향상하고 멀티프로세싱을 더 많이 하기 위해서이다.

가상 메모리를 사용하는 것이 이제는 당연해졌다.

프로세스를 수행할때, 프로세스가 물리 메모리에 있을 필요가 없다

폰 노이만 구조를 생각하면 program counter와 stack pointer가 움직인다.

프로세스가 수행되는게 뭐냐? program counter가 앞으로, stack pointer가 뒤로 움직이는 것이다.

그렇기에 모든 부분이 전부 물리 메모리에 올라갈 필요가 없다는 것이다.

그 중의 일부만이 물리메모리에 올라와 있다.

각각의 프로세스들이 concurrent하게 수행되는데 문제가 없다.

time sharing은 프로세스가 수행하는 순서가 계속 바뀌는 것이다.

교수님 질문

가상 메모리가 뭐냐?

→ 작은 메모리에 큰 프로그램이 돌아가게 하는 기술이다.

→ 사용자가 실제로 가지고 있지 않지만, 쓸 수 있다고 생각되어지는 메모리

학생 질문

제가 설명을 좀 놓친것 같은데 78MB Executable이 전부 메모리에 올라가지 않아도 되는 이유가 뭔가요?

A. program counter와 stack pointer 주변의 메모리만 있으면 수행을 할 수가 있다.

1mb이하의 메모리만 물리메모리에 올라가 있으면 실행에 무리가 없다.

만일 go-to를 사용해서 그 주소가 물리 메모리에 없는 경우가 발생할 수도 있다.

사용자는 메모리가 부족하다고 생각하지 않고 실행한다.

page table

어떤 프로세스의 어떤 메모리가 물리메모리로 mapping되었다는 것을 page table에 기록해둔다.

MMU가 page table를 가지고 있다.

MMU가 page table를 관리한다.

학생질문 :

code/data/stack의 전체가 아니라, 일부만 올라가도 되는데, 그럼 그 일부가 뭔가?

A.

교수님 가상메모리가 프로세스 간의 concurrent를 보장하지만 필요한 부분마다 메모리에 올린다면 속도는 더 느릴 것 같습니다. 실제로도 그런가요?

A. 더 느려진다. 그래서 MMU를 사용하는 것이다.

MMU를 통해서 느려지지 않게 만드려고 한다.

가상메모리 – main memory와 secondary storage를이용한 구현

물리 메모리가 만약 부족해지면 어떻게 하는가?

→ Secondary Storage로 보낸다. (swap space)

그러다가 필요해지면 swap space에서 메인메모리로 꺼내온다.

학생질문

PU에서 하드웨어적으로 MMU를 거쳐 physical address에 접근한다면, MMU를 거치지 않고 직접 physical memory에 접근할 수 있는 방법도 있나요?

A. 옛날에는 yes, 지금은 no

지금은 반드시 MMU를 거친다.

가상메모리– logical to physical address mapping table

VA 3, 8이 PA 6을 공유한다.

dll이 이런 방식으로 작동한다.

이 역시 가상메모리가 하는 일이다.

가상메모리가 하는 일

: 1. 프로그램의 크기에 관계없이 메모리를 물리적으로 쓸 수 있게 해준다.

: 2. 주소 공간들이 물리 메모리를 공유할 수 있게 해준다.

ex) IPC의 shared memory (shmat(2)

shared memory를 프로세스에 붙인다는 것의 구현을 이러한 방식으로 한다.

ex) dll도 마찮가지이다.

Paging

- 주소 공간을 동일한 크기인 page로 나누어 관리
  - 보통 1page의 크기는 4KB로 나누어 사용

    → 실험애 봤더니 4KB가 제일 낫더라

  - 프레임 (Frame)
    - 물리 메모리를 고정된 크기로 나누었을 때, 하나의 블록
  - 페이지 (Page)
    - 가상 메모리를 고정된 크기로 나누었을 때, 하나의 블록
  - 각각의 프레임 크기와 페이지 크기는 같다.

- 페이지가 하나의 프레임을 할당 받으면, 물리 메모리에 위치하게 된다.
  - 프레임을 할당 받지 못한 페이지들은 외부 저장장치 (Backing store)에 저장된다.

    할당을 못 받은 애들은 disk 어딘가에 존재한다.
    - Backing store도 페이지, 프레임과 같은 크기로 나누어져 있다.

Paging –page 번호와offset

- CPU가 관리하는 모든 주소는 두 부분으로 나뉜다.
  - 페이지 번호 (Page number)
    - 각 프로세스가 가진 페이지 각각에 부여된 번호
    - 예) 1번 프로세스는 0부터 63번까지의 페이지를 가지고 있음
  - 페이지 주소 (Page offset)
    - 각 페이지의 내부 주소를 가리킴
      - 예) 1번 프로세스 12번 페이지의 34번째 데이터

4kb는 4098이다.
여기에 접근을 어떻게 하나?

프로세스가 받은 페이지에 번호를 부여한다.
페이지에 0번지, 1번지, 2번지가 있는 것이다.
그 다음에 페이지 안에서의 번호인 offset을 준다.
offset은 0~4097까지이다.

어떤 것은 가변 페이지 크기를 지원하는데, 이는 VA에 대한 것이다.

학생질문
페이지와 프레임의 크기가 다를 수도 있나요?
yes, 그런데 그렇게 안하는 것이 좋다.

프로그램이 큰 경우에는 페이지를 여러개를 할당하는 방식으로 작동한다.

vm문제도 시험에 한 문제 이상 나온다.

4KB의 page라면, offset은 얼마인가?
→ 0~2^12-1이다.

그렇다면 offset을 나타내는데 필요한 bit의 갯수는 얼마인가?

12bits이다.

중간점검퀴즈

- 128MB의 물리 메모리를 4KB 단위로 페이징 하려고 하면, 몇 개의 frame이필요한가?

  128MB/4KB = 32k이다.

- 4GB의 logical address를 페이징 하려고 하면, 총 몇 개의 page가 필요한가? (페이지 크기는 4KB)

  100만 개의 page가 필요

- page의 크기가 4KB일 때, 한 페이지의 메모리를 access하기위한 주소 bit는 몇 비트인가?

  주소 1개당 1Byte의 메모리를 access가능 하다는 가정이 필요

  page의 크기가 4KB

  메모리 access의 단위가 Byte이다.

  page의 주소의 갯수는 4KB = 2^12bit이다.

  그래서 우리에게 필요한 주소의 bit 수는 12bit이다.

  4KB * 1Byte = 32KB = 2^()

  12bit

Page table

- 페이지 테이블 (Page Table)
  - 각 프로세스의 페이지 정보를 저장함
    - 프로세스마다 하나의 페이지 테이블을 가짐

      → 프로세스마다 하나의 페이지 테이블을 가진다.
  - 인덱스 : 페이지 번호
  - 내용 : 해당 페이지에 할당된 물리 메모리(프레임)의 시작 주소

        - 이 시작 주소와 페이지 주소를 결합하여 원하는 데이터가 있는 물리 메모리 주소를 알 수 있다.

- 페이지 테이블의 구현
    - 페이지 테이블은 물리 메모리에 위치함

      → 페이지 테이블은 persistent하지 않다.
    - 페이지 테이블 기준 레지스터 (PTBR: Page-table base register)가 물리 메모리 내의 페이지 테이블을 가리킴

      → CPU안에 page table을 가리키는 register가 있는데 context switching할 때 이것을 변경함
    - 페이지 테이블 길이 레지스터 (PTLR: Page-table length register)가 페 이지 테이블의 사이즈를 나타냄

page table은 kernel data structure이다.

교수님 질문

VA를 PA로 맵핑을 하는데 프로세스마다 page table이 있을 필요성이 무엇인가?

프로세스끼리 서로의 메모리 공간을 침범하면 안되기 때문이다.

프로세스가 서로 침범을 안하게 되는 이유가 무엇인가?

서로 page table을 달리 가지고 있기 때문이다.

각각의 page table을 가지고 있다는 것은 각각의 접근하는 메모리를 달리할 수 있다는 것이다.

학생질문

프로세스 간에 서로 접근을 못하게 하는 것이 page table을 통해서 가능해 진다.

즉, 프로세스라는 abstraction이 page table로 구현된다.

하나의 page table에서 공유되는 것은 무엇인가?

shamat(2) 하는 것이다.

이것은 kernel이 허용하는지의 여부에 따라 성공과 실패가 결정된다.

context swiching을 할때는 page table을 바꿔줘야 함

그렇지 않으면 엉뚱한 물리메모리에 접근할 수 있다.

다른 page table을 가져야 다른 물리메모리에 접근가능

Page table 을이용한LA –PA mapping

page table을 통해 VA가 PA로 맵핑된다.

Page table – page table을 이용한 주소 변환

CPU에서 주소가 나오면 2개로 쪼갠다.

offset과 page 번호

교수님 질문

32bit 시스템에서 4KB의 page크기이면, P의 bit수는 몇개인가?

20bit이다.

20bit는 page table로 들어가고, 나머지 12bit는 offset으로 간다.

이슈가 있다.

page table에 f가 없을 수도 있고

page table에 접근하고, 또 물리 메모리에 접근해야 한다.

그래서 성능이 떨어진다.

Page Table Entry (PTE)

- PTE란?
  - 페이지 테이블에 있는 하나의 record – page 에 해당

    PTE - page는 1대1 대응이다.
- PTE의 각 필드 내용
  - Page base address

    - 해당 페이지에 할당된 프레임의 시작 주소 → f

      이 f에 d를 더하면 물리메모리에 접근 가능
    - 이를 통해 물리 메모리에 접근할 수 있음
- Flag bits : 매우매우 중요
    - Accessed bit: 페이지에 대한 접근이 있었는지
    - Dirty bit: 페이지 내용의 변경이 있었는지
    - Present bit: 현재 페이지에 할당된 프레임이 있는지

      → f=123|0이면 page 123에 할당된 frame은 없다는 말이 된다.
    - Read/Write bit: 읽기/쓰기에 대한 권한 표시

      → trap을 공부 했었다. 그때를 생각해 봐라

      → *p = 0을 접근하려고 하면 trap이 발생한다.

      → 왜 그랬을까

      *p = 0은 0번째 page table의 제일 첫번째 PTE의 read bit는 0이다.

      그래서 더이상의 excution이 불가능 해서 trap이 발생한다.

      MMU가 PTE에 접근을 해서 봤는데, 예가 읽으면 안되는 값을 읽으려는 것을 보고 trap handler가 signal을 호출하고, signal이 해당 프로세스에게 terminate하라고 한다.

      (이는 거부 할 수 없는 signal이다.)
- 그 외 구성에 따라 여러 가지 내용들이 포함될 수 있다.

Translation Look-aside Buffers (TLB)

- 페이징 방법에서는 데이터로의 접근이 항상 두 번의 메모리 접근을 거쳐야 함
    - 페이지 테이블에 한 번, 물리 메모리 내의 데이터에 한 번
    - 메모리 접근 속도를 크게 떨어뜨림

컴퓨터공학의 두가지 대 주제가 있다.

1. caching
2. indirect 계층

TLB는 caching이다.

주소 caching이다.

- 해결 방법 – TLB in MMU
  - 페이지 테이블을 이용해 변환된 주소를 TLB에 저장해둠
  - 다음 접근 시에는 TLB에 저장된 값을 이용하여 빠르게 변환된 주소를 얻을 수 있음
    - TLB는 레지스터이기 때문에 빠른 수행이 가능함
  - TLB hit ratio
    - TLB 내에서 원하는 주소를 찾을 수 있는 확률
    - 높을 수록 메모리 접근 속도를 향상시킬 수 있음

TLB는 PTE의 cache이다.

TLB 도식

TLB에 동시에 많이 넣어버린다. (SAC)

그래서 TLB hit이 되면 바로 f에 넣고 pagetable lookup을 하지 않는다.

그래서 속도가 빨르다.

좋은 질문

TLB proccessing을 어떻게 하는가?

TLB은 하드웨어 자체에 들어있다.

이것을 각각의 프로세스가 가질 수 없다.

방법이 뭔가?

flush할 수 있게 한다.

context가 바뀌는 것도 의미가 없다.

TLB를 어떻게 저장하나?

초창기에는 MMU에서 하나만을 저장하였다.

나중에 가서는 점점 그 갯수가 늘어났다.

교수님 tlb를 flush 시킨다는게 초기화 시킨다는 말씀이신가요?

그래야한다. 그래야 새로 스위칭 된 프로세스가 캐슁된다.

Multilevel Page Table

- 다단계 페이지 테이블의 필요성
  - 시스템의 발전에 따라 가상 주소 공간도 매우 큰 용량을 요구하게 됨
  - 그로 인해 페이지 테이블의 크기가 커지고 그 차지하는 공간에 의해 페이징이 잘 이루어질 수 없다.
  - 예) 32비트 가상 주소 공간을 가지는 시스템
    - 페이지 크기 : 4Kbyte (2^12) 일 경우,
    - 페이지 테이블 크기 : 1M (2^20) * 4 byte = 4Mbyte (프로세스 1개당임)
      - 페이지 테이블의 값 하나는 보통 4 byte를 차지한다.
  - 페이지 테이블 자체도 페이징된 공간에 저장됨

프로세스를 100개를 돌리면 400MB

프로세스를 1000개를 돌리면 4GB가 된다.

굉장히 sparse하다는 것이다.

→ 성기다는 뜻이다.

내 프로그램은 300MB인데, 이것을 mapping을 하기 위한 PT의 크기는 4MB로 늘 고정된다.

→ 배보다 배꼽이 더 클 수 있다.

2 level Page Table 도식

교수님 level 2 page table에서 level 1 PT의 entry가 4byte일 필요가 있는지 궁금합니다. 하나의 level 1 PT가 가리키는 level 2 PT 의 총 갯수는 2^10 개 이므로 한 entry에 10bits + flag bits면 충분하지 않나요?

L1 PT, L2 PT 모두 물리 메모리에 존재한다.

물리 메모리에 접근하기 위해서 32 bit가 필요하다.

이 32를 20/12로 나눈다.

20은 page번호, 12는 offset이다.

페이지 테이블의 주소를 어떻게 아는가?

→ context swicthing을 할 때 L1 PT의 첫번째 주소를 바꿔준다.

PCB에 물리 주소를 딱 적는다.

Inverted Page Table 도식

프로세스마다 PT를 할당하는 것이 아니고, 시스템이 가지고 있는 물리메모리 전체에 대해서 PT를 할당하고 거기에 접근을 하는 것이다.

Demand Paging

- Demand paging이란 프로세스의 실행을 위한 모든 page를 메모리에 올리지 않고, 필요한 page의 요청이 발생할 때(프로세스가 요청할 때) 메모리에 올리는 paging 기법

  → 프로세스가 필요로 하는 page를 올려 놓겠다는 것이 Demand Paging이다.
- Paging 서비스를 통해서 한 프로세스에 필요한 page를 memory와 secondary storage 간에 이동시킴

  → 프로세스가 원하는 PT를 그때그때 잘 저장소로부터 옮겨 놓는 것이다.
- Valid and invalid page

  가상 공간에서 Valid page와 invalid page의 차이가 나야 한다.

  Valid page는 PTE가 있다.

  Valid page는 물리메모리에,

  invalid page는 아직 저장소에 있다.

  교수님 질문

  stack은 어디에 있나?

  stack은 저장소에 저장되고 있나?

  프로그램에는 T/D만 있다. Proccess는 T/D/S가 있다.

  swap area는 stack을 저장하기 위한 곳이다.

- Demand paging의 장점
  - 실행을 위한 물리 메모리 구성의 시간이 줄어든다.
  - 프로세스의 전체 이미지를 메모리에 올리지 않기 때문에 실제 필요한 전체 물리 메모리의 양을 줄일 수 있다.

Demand Paging (cont.)

- Demand paging의 단점
  - 참조하려는 page가 valid한 경우
    - 참조하고자 하는 page가 실제 물리 메모리에 있기 때문에 정상적인 참조가 일어남
  - 참조하려는 page가 invalid한 경우
    - 참조하고자 하는 page가 실제 물리 메모리에 없으므로 이에 대한 처리가 필요
    - Page fault 발생

      저장공간에서 page를 불러서 물리 메모리에 옮기는 작업을 page fault라고 한다.

프로세스를 처음 시작하면 어떻게 되는가?

프로그램을 storage에서 가져와서 loader가 메모리에 올려 놓고, RTS이 excution을 시작한다.

그 후 제일 먼저 발생하는 것이 Page fault이다.

처음 시작할 때는 물리메모리에 page가 올라와 있지 않으므로, Page fault가 많이 발생

preloading이라는 개념도 있다.

먼저 물리메모리에 text나 data를 미리 올려두는 것이다.

Page fault가 발생하면 kernel memory allocator에게서 page를 받아온다.

- Demand paging을 하기 위해서 page table에 어떠한 정보가 더 필요할까?

Demand paging 기법을사용한 page table

가상 공간에서 Valid page와 invalid page의 차이가 나야 한다.

page table의 Valid page는 PTE가 있다. (4, 6, 9)

logical memory하고 page table하고는 linear하게 mapping이 된다.

Page Fault (cont.)

1. 어떤 페이지를 reference한다.
2. 그런데 invalid한 page였다. 그러면 trap이 발생한다.
3. 그 trap을 처리하는데, 그 방식은 disk에 page가 없는가를 찾는 것이다.
4. page를 찾았으면 메모리로 가져와야 하는데, 그러려면 물리메모리의 할당을 받아야 함

   비어있는 메모리를 할당받음
5. 그 페이지 테이블의 번호를 page table에가져다 놓는다. 그리고 invalid를 valid로 바꾼다.
6. trap이 끝났음으로 재 시작한다. (rtt) 멈춰있던 프로세스를 제시작한다.

stack은 어떨까?

제일 먼저 발생하는 것은 text에 대한 Fault이다.

이는 프로그램에서 text를 그냥 가져오면 되는 것이다.

그런데 stack은 가져올데가 없다. (프로그램에 없다.)

→ 그냥 메모리에 할당하면 끝난다.

→ disk를 갈 필요가 없다. (가봤자 없다.)

→ 비어있는데로 할당하면 된다.

만일 물리메모리가 부족하면 page out을 시켜서 메모리를 확보해야 한다.

그런데 stack page를 page out시켜야 하는 경우에 대해서 stack은 딱히 돌아갈 곳이 없다.

그런 stack을 위해 존재하는 것이 바로 swap sapce이다.

Page Fault는 성능에 중요한 역할을 한다.

frame을 늘려줘야 page fault가 감소한다.

Page Fault가 발생하는 것은 당연하지만 발생을 안하게 만드는 것이 좋다.

Page Fault가 0으로 만드는 것이 좋다.

Page Fault는 부팅할 때 어쩔 수 없이 발생하는 것이다.

그러나 어느 시점이 지난 후에는 Page Fault가 발생하지 않도록 만드는 것이 좋다.

잠깐 잠깐 Page Fault가 발생하는 것이 정상이다.

지속적으로 Page Fault가 발생하는 것은 비정상이다.

steady state에 들어가면 Page Fault가 발생하지 말아야 한다.

Page Fault -locality

까만 지역은 메모리의 특정부분을 자주 접근했다는 것이다. (정상)

go-to를 쓰면 왔다갔다 하면 locality가 떨어진다.

locality가 떨어지면 cache utility가 떨어지고, 이는 성능에 치명적이다.

locality가 올라가는 것이 좋다.

locality가 크면, page fault가 발생할 확률이 적다.

VOD는 locality가 없다.

Working set

- 정의
  - 어떤 시간 window 동안에 접근한 page들의 집합
  - 시간마다 working set이 변함
    - ... 25737456745680120123456
  - “적당한” window를 잡으면 working set은 locality를 나타냄

→ working set은 주어진 시간 내에서 프로세스가 접근 한 page들의 집합임

|locality| = N

|working set| = N

프로세스 p에 대해서 이 프로세스에 대해서 working set만큼의 page를 할당해 주면 page fault의 숫자를 최소화 할 수 있다.

프로세스 p를 1000개를 지원해야 하는데, 만약 working set이 100 page이면

여기에 필요한 메모리는?

1000 * 100 * 4KB이다.

시간의 크기에 따라서 working set이 커지기도 하고, 작아지기도 한다.

1000개의 프로세스 → 이것을 처리하는데 필요한 물리메모리의 양을 계산하는데 working set을 사용한다.

우리는 working set을 통용적으로 locality를 가정하고 말함.

큰 working set을 프로세스가 필요한 working set이라고 한다.

프로세스가 1000개가 있으면 그 working set * 1000하면 나온다.

이것은 provisioning이야기이다.

provisioning이 충분히 잘되었다면 pf는 최소화되어야 한다.

pf

1. 물리메모리를 더 꽂는다.
2. 실행되는 프로세스를 줄인다.

그게 안되면 Thrashing이 발생한다.

- Thrashing
  - 프로세스의 실행시간 중, page fault를 처리하는 시간이 execution 시간보다 긴 상황

    메모리가 작으면 이런 현상이 나타난다.

→ CPU는 바쁜데, 처리량은 떨어지는 것

사실 운영체제는 최소한으로 도는 것이 제일 좋다.

vmstat(8)

user time, system time이 나온다.

Thrashing을 하고 있다면 system time이 굉장히 높다

ex) 디도스공격

학생 질문 : ddos 공격이 page fault를 이용한 공격인가요??

→ ddos 공격 중 어떤 것은 SYN 패킷만 계속 보내는 것이다.

→ 패킷이 들어오면 interrupt가 발생. 그런데 이것이 초당 1000만개가 들어오는 것이다.

1 마이크로초가 걸린다고하면 10초가 걸린다. 이러면 원래 처리하려는 것을 하나도 못한다.