VM for Embedded Systems

범용 gerneral purpose system과 embeded system의 경계가 무너지는 대표적인 사례

원래 embbeded 시스템 디자이너들은 virtual memory 부분에서 신경을 쓰지 않음

메모리라는 리소스(자원 - cpu, 메모리, io devices, 저장장치)에 관한 문제 : 자원을 쓰고 싶은 client가 몇 명이냐?

- Virtual memory (자원 관리에 필연적)
    - 임베디드 시스템에서는 문제(이슈)가 되지 않았음 : 임베디드에서 돌아가는 processor는 하나였기 때문
        - OS 사용 없이 단일 전용 application만 사용되었기 때문
    - 변화된 상황
        - 많은 최신 임베디드 시스템이 OS에 의존
    - 사용했을 때 위험성
        - Real time(데이터 전달량 및 처리량 보다는 시간이 중요함) 특성에 악영향을 끼침

Memory Management

user process가 하나가 아닌 n개일 경우 어떻게 해결할까

목표: 상황에 따라 시시각각 다른 process한테 넘겨주고 할 수 있을까

- 목표
    - 프로그래밍을 위한 편리하게 추상화
    - 서로다른 프로세스 간의 isolation 제공
    - 프로세스 전반에 걸쳐 부족한 물리적 메모리 자원 할당
        - 메모리가 심하게 다투는 경우 특히 중요
- 방법
    - 가상 주소 변환 (Virtual address translation) (모듈)

      각 프로그램은 자기가 4GB를 다 사용할 수 있다고 착각
    - 페이징(paging (자료구조))과 TLB
    - Page table 관리
- 규칙
    - Page 교체 규칙

  과거를 통해 미래를 예측!

Virtual Memory

os는 자원 management 총 책임 -> 메모리 관점에서 어떻게 관리를 할지

cpu를 어떻게 가상화 할지 -> 시간으로 가상화, 지금은 차라리 cpu를 여러개 놓자

- VM이란?
    - OS에서 메모리 관리를 위해 제공하는 기본 추상화
    - 물리적 메모리에 있는 전체 주소 공간을 요구하지 않고도 프로그램을 실행할 수 있도록 지원
        - Called "Demand paging" (요구 페이징 : 필요할 때마다 해당 페이지를 가져오는 개념)

2bit cpu 에서는 32bit addressing, 2^30 -> GB * 2^2 -> 4 = 4GB에 space에 대해서, physical 메모리가 실제로 user processor가 요구할 수 있는 메모리 공간과의 갭이발생 : 다 안쓸거잖아!

각자는 다 내노라고 하지만 방법이 있을 것같은데?

locality 를 잘 분석하면 여러 user process를 만족시킬 수 있음

- Demand paging system

  page: 가상 주소로 표현되는 메모리 블록을 칭함

  frame: page에 대응되는 실제 물리적 메모리 블록

  page falut: 해당 데이터가 실제 메모리에 없을 경우, 하드디스크로 부터 읽어드려야하는 ..

1. 어떤 process가 처음으로 어떤 page 쓰려고 하는 순간
2. 잠깐만 기다려, 이 page를 실제 physial memory에 할당해주고 (page fault, 시간 얼마나 걸릴지 모름 panelty) -> `virtual memory가 실시간성을 심하게 저해하는 요소를 제공할 수 있다.`
3. 아무 일 도 없었던 것처럼

- Observation (패턴을 발견하기 위해 관찰이 엄청나게 중요해요)
  - 많은 프로그램에서 모든 코드 또는 데이터를 사용하지 않음
    - ex) 실행하지 않는 분기, 액세스 하지 않는 변수 등
  - 사용할 때 까지 메모리를 할당할 필요가 없음

Virtual Memory

- 기능
  - (물리적 주소가 process 여러개한테 분배가 돼야하는데 이거를 어떻게 하면 많이 이용한다고 많이 줄지, 조금만 줄지, 등을 paging으로 분배)
  - OS는 VM을 활용하여 process의 실행 시간(run-timge) 에 따라 물리적 메모리의 양을 조정
  - process들 간의 isolation 제공

    (거기는 니 영역이 아닌데? OS와 ARM CPU 하드웨어가 같이 관리, 어떤 process가 합법적으로 접근할 수 있는 영역이 아닌지를 점검을 하고, 허가받지 않은 영역에 대한 액세스를 발생시키면 abort)
  - 각 process는 개인적인 분리된 주소 공간이 있음 (위랑 똑같은 얘기)
  - 한 process는 다른 process의 메모리 주소를 접근할 수 없음
- 구현
  - HW가 지원하는 것 : address tanslation (주소 변환)

    MMU(Memory management unit) : 프로세스가 발생시키는 virtual memory를 physical memory로 변경, TCB(translation lookaside buffer)(MMU에 포함된 캐시같은 구조)

    exception이 처리되는 과정, 누구의 책임이죠? 초반에 몇가지는 하드웨어가 자동으로 처리. exception이 발생 하자마자, stack에 대피 시키는 과정은 소프트웨어 ! 여기서도 마찬가지

    어떤 OS를 설치할 때 반드시 확인해야하는것 : MMU를 지원하는가? 아니면 그것은 반쪽짜리 OS, OS는 메모리 가상화가 필수적 완전 중요한 요소 최고장점
  - OS 가 지원하는 것 : Virtual Address - to - Physical Address

    Page fault hander, page table (SW의 책임) 왜 HW의 책임이 아닌가 ? 아주 빨리 실행되면서 비교적 간단한일 - 하드웨어가 책임, SW같은 영역은 디게 복잡하지만 빈번x,

-> page fault handler가 하는일: fault를 만든 프로세스의 수행을 중단시켜야함, MMU가 해당 데이터가 속한 하드 디스크 영역을 고정크기 (page)혹은 가변 크기 (segment)의 블록 단위로 메모리로 읽어 들임

```
- 그 프로세스의 context들을 전부 save, 그 프로세스를 block 상태로 만들어야함
```

- FTL(Flash Translation Layer)

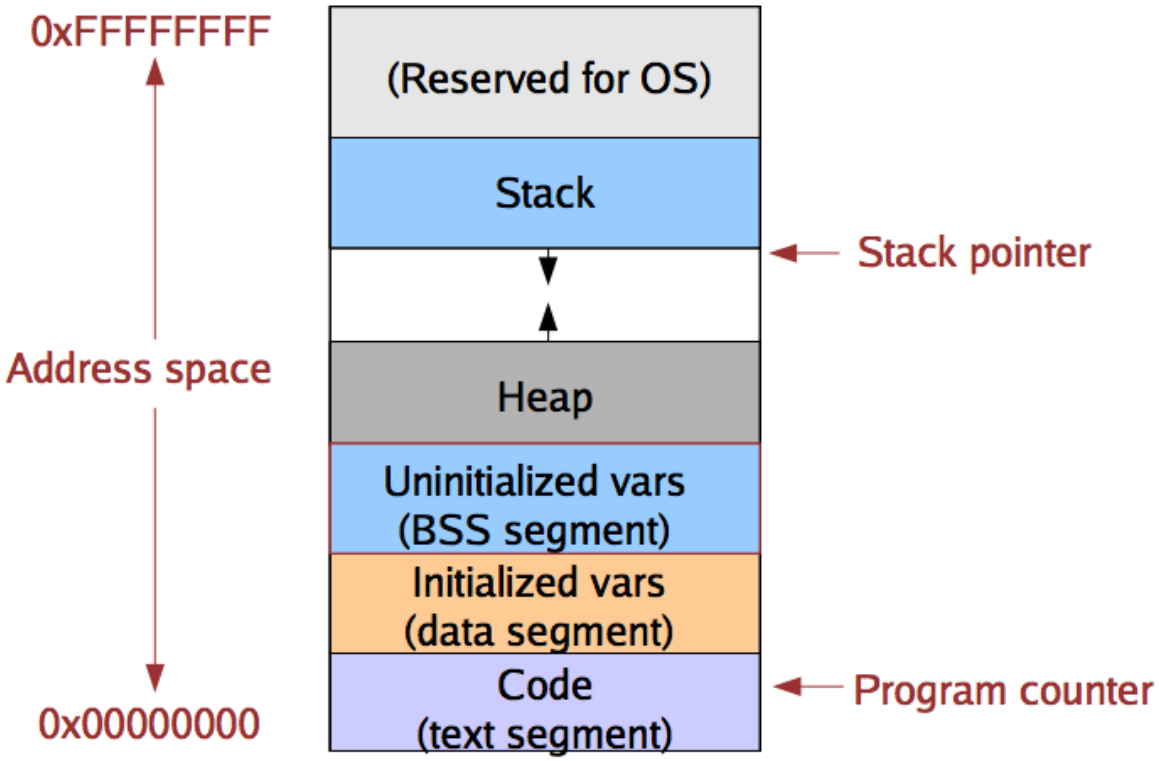
  플래시 메모리와 파일시스템 사이에 위치하면서, 플래시 메모리를 디스크처럼 사용할 수 있게 해주는 사상(Mapping) 기술 = 섹터 기반의 시스템에서 블록/페이지 기반의 SSD를 사용하기 위해서는 섹터로 변환시켜주는 기술 = FTL을 통해 섹터로 변환 = SSD를 SDD처럼 사용

  (용량에 따라서 block을 잡고 logical한 주소만 쓰고 싶음, mapping을 하드디스크 안에 controller가 해주고있음 (translate))

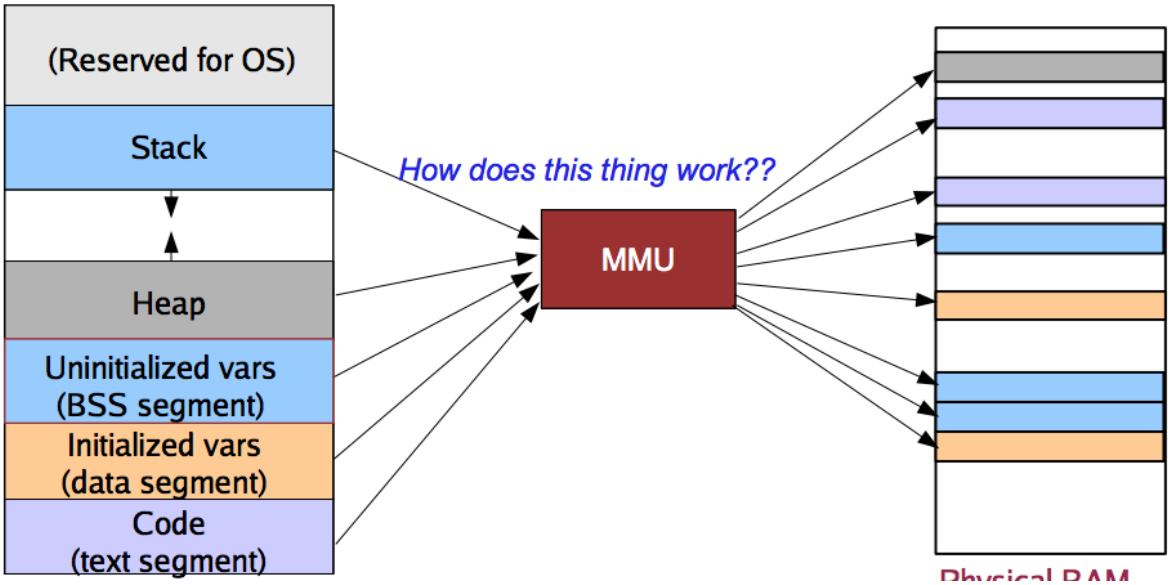
Virtual Address (VA)

- VA : 프로세스가 자체 메모리에 엑세스 하는데 사용하는 메모리 주소
- VA != PA (물리적 주소)
- PA = MMU(VA)
- VA-PA 매핑
- OS에 의해 결정됨

Virtual Address (VA)

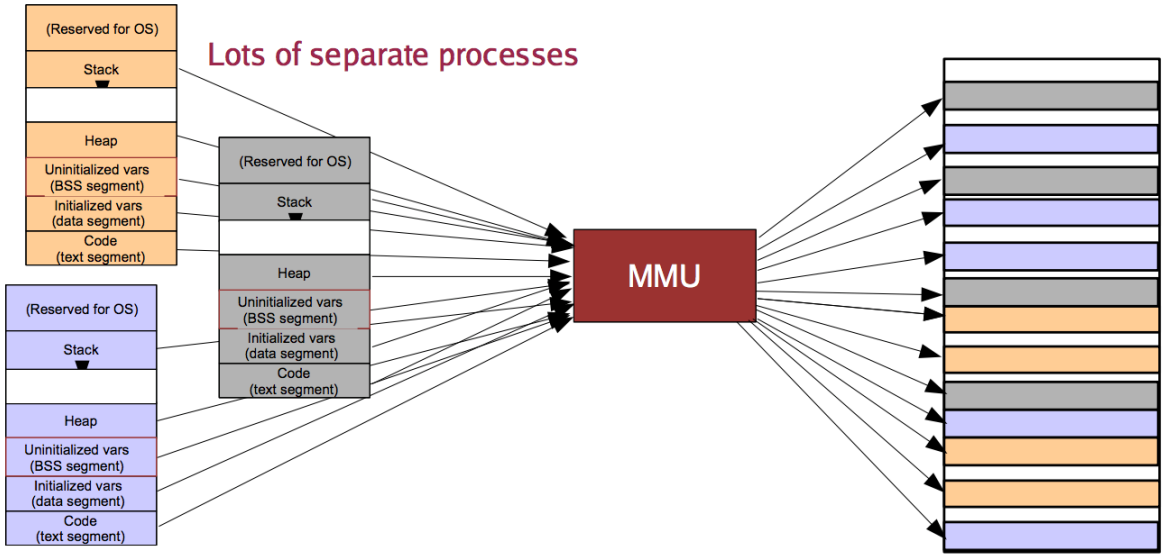


Virtual Address (VA)



무조건 MMU가 시키는 대로 메모리 재배정을 받아야함, SSD에서도 똑같은 상황 발생

Application Perspective



Virtual Address

- isolation, protection

  (한 user 프로세스가 다른 user 프로세스의 physical 메모리 공간을 터치할 수 있는 방법이 원천적으로 차단

  예외상황 : process 사이에 통신하는 방법 - ~메모리 영역으로 사용 -> 이쪽 process가 사용하는 다른 process가 사용하는 virtual 메모리영역하고 주소가 바뀔 필요는 없는데 제한적으로 허용. physical memory가 겹침 -> 이쪽 프로세스가 저쪽프로세스의 내용을 읽을 수 있고,, isolation 상황 o 이 항상 맞진않다.)

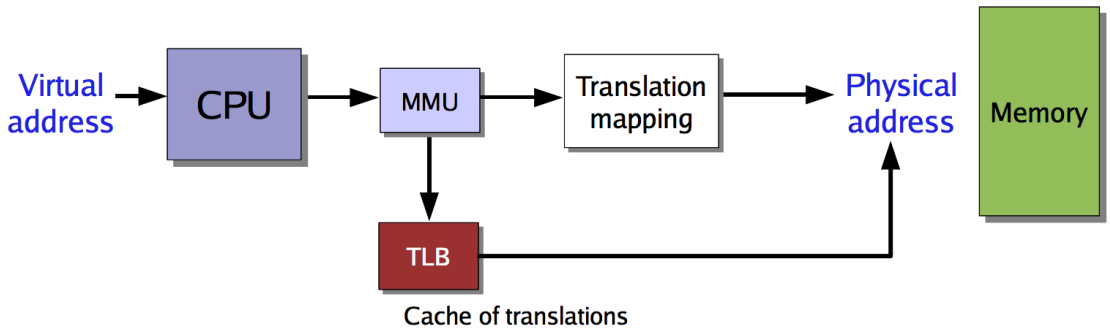
  - 한 프로세스의 VA는 다른 프로세스의 가상 주소와 다른 물리적 메모리를 나타냄
    - ex : 메모리 공유 영역 btw processes
- Relocation

  (코드의 위치가 유지되지 않고 바뀐다. 위치를 정확하게 알아야할 프로그램: complier가 relocaable 코드를 만들도록 뒷단에서 노력, 절대주소에 의존적이지 않은 구조들이 담겨 있다.)

  - 프로그램은 실행될 때 어떤 Physical address를 쓸건지 알 필요가 없음
  - 컴파일러가 다시 연결 가능한 코드 생성

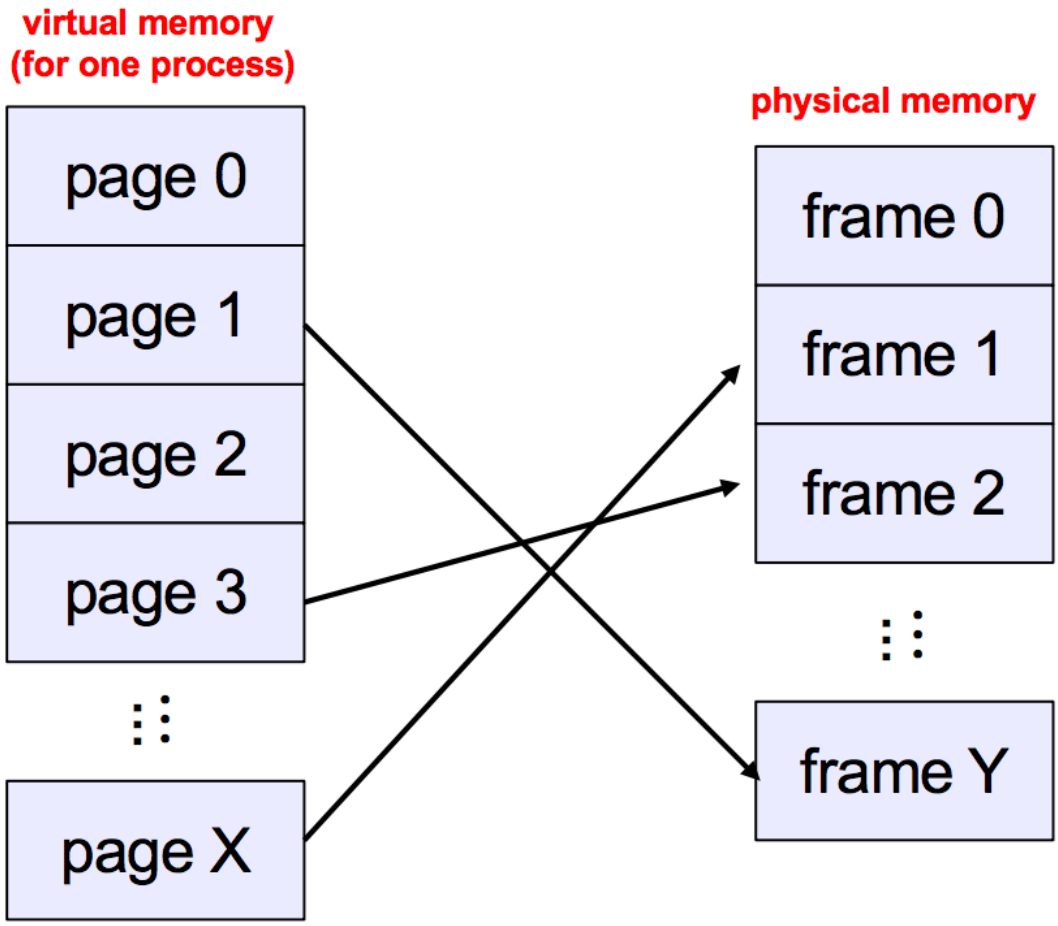
MMU and TLB

- MMU (memory management unit)
  - VA에서 PA로 변환시키는 하드웨에
- TLB (translation lookaside buffer)
  - MMU를 위한 Cache



Paging

- 가상 메모리는 page라는 고정 사이즈로 분할되어있음
- 물리적 주소는 page frame으로 분할되어있음



Page Tables

(주소변환 정보를 누군가는 가지고 있어야함, 참조하는 놈은 MMU(HW) 실제 physical meomory와 매핑해줘야함, HW와 SW가 같이 협력) 4GB / 4KB -> 4MB(백만) 2^20

매핑 서비스 제공 입장에서는 하위 몇 bit 까지는 dontcare : base + offset

4KB chunck 관리하는데 translation 대상 :

user process가 2개 있다고 해봅시다. page table 몇개있어야하죠?

virtual address space(고유, private)가 2개라서 2개

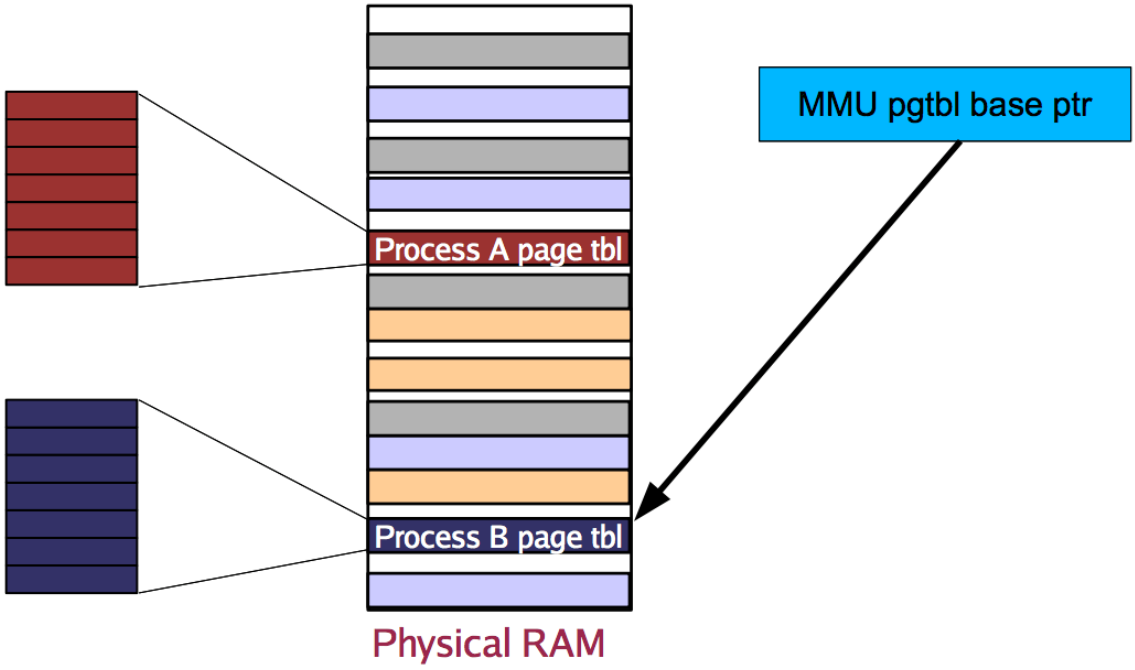
process가 100개이면? page table 사이즈는? 100 -> page table을 main memory에 들어감 (page table은 cpu가 실행될 때 계속 참조하는데 SSD, HDD에 저장되면 너무 느림)

page table이라는 것을 MMU한테 알려줘야함 : cpu안에 있는 special한 레지스터 존재, os가 이 시작 주소를 저장

최초에는 페이지

다 가져올 수 밖에 없지만, 그 다음 부터는 시간지역성, 공간 지역성으로 가져옴

- virtual-to-physical 주소 매핑 정보를 저장
- 위치 : main memory
- 접근 방법
  - MMU는 page table base register라는 special register를 가지고 있다.
  - page table base register는 현재 동작하고 있는 프로세스의 page table 물리적 메모리 주소의 맨 위를 가리킴



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

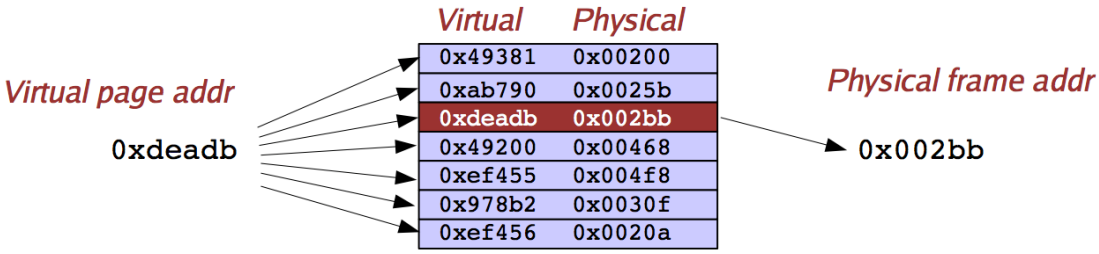
CPU가 내는 주소는 13이다. 이진수로 나타내면 1101(2) 인데 앞 두숫자 11이 p 이고 01이 d라고 볼 수 있다. (p는 페이지 사이즈 크기만큼 크기를 할당. 페이지 크기가 4이므로 두자리가 된다.) 페이지 테이블의 3번째 인덱스에 저장된 2가 f가 되고 d는 그대로 온다. 즉 10 01이 피지컬 어드레스가 된다. 즉 물리주소는 9번지가 된다. 9번지는 두번째 프레임에서 01만큼 떨어져있다고 할 수 있다.

예제
Page Size = 1KB

```
Page Table: 1 2 5 4 8 3 0 6
논리주소 3000번지는 물리주소 몇 번지?

1kb는 10bit. 3000은 101110111000(2) 이다. 페이지 사이즈는 2^10 이므로 10bit로 표현 가능하다. 즉 d는 뒤의 10자리(1110111000(2)) 이고 p는 앞에 두개(10(2))이다. 즉 2번째 인덱스인 101(2) 그리고 d를 붙인 1110111000 (2) 가 물리주소이다.

물리주소 0x1A53 번지는 논리주소 몇 번지?

1 1010 0101 0011 인데 뒤의 10개가 d 나머지 앞에 110이 프레임 넘버. 6은 페이지 테이블의 7번지에 저장되어 있으므로
111과 10 0101 0011 을 붙인것이 논리주소이다.
```

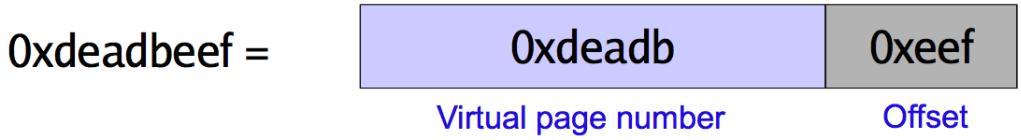
The TLB

- MMU overhead
  - 각 메모리 엑세스는 페이지 테이블에 대한 추가 메모리 엑세스 필요 (100% 넘어가면)
- Solution : TLB (Translation Lookaside buffer)
  - CPU랑 바로 연결된 매우 빠른 chache (하지만 작다)
  - 가장 최근에 v-to-p 주소 변환한 것 저장
  - TLB miss는 메인메모리에 page 엑세스를 필요로 함

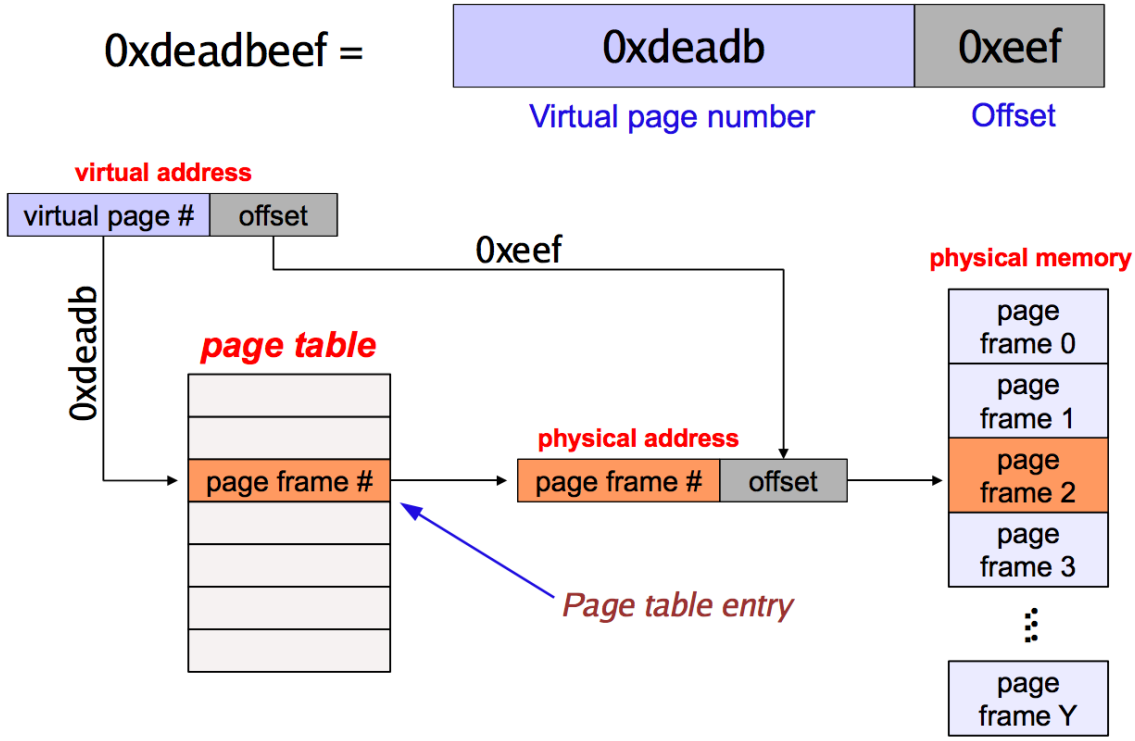


Virtual Address Translation

- MMU가 수행하는 것
  - VA(virtual address) = Virtual page number and offset
  - page table이 제공하는 virtual page와 physical frame을 매핑



Virtual Address Translation

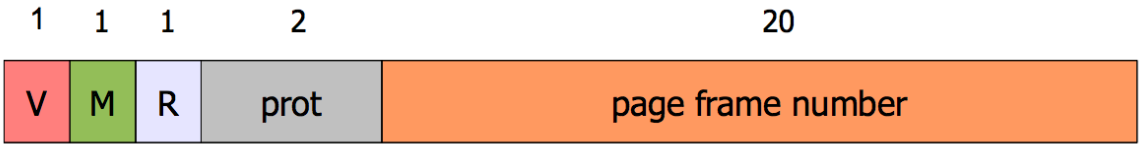


index : virtual page number

MMU가 먼저 TLB에 없는 경우에 이걸 함

Page Table Entries (PTEs)

가상의 page table entry 를 가질 수 도있다. 정도만



- Valid bit (V) : 해당 페이지가 메모리에 있는지 여부
- Modify bit (M) : 페이지가 "dirty"한지 여부 (변경 여부)
- Reference bit(F) : 페이지를 엑세스 했는지 여부
- Protection bit : 페이지를 읽을 수 있는지, 쓸 수 있는지, 실행 가능한지 지정
- Page frame number : 메인 메모리에 있는 물리적 위치
- PFN + OFFSET = physical memory address