8. Memory Management 1

Binding of Instructions and Data to Memory

1. Complile time binding

   multitasking, mutliprogramming 이 아닌 경우에만 사용 가능, 한번 정해지면 바꿀 수 없기 때문에 만약 변경사항이 생기면 recompile 해야한다

2. Load time binding

   프로세스 간에 상대 주소만 저장해서 실제 메모리에 프로세스가 올라 갈 때 주소가 확정됨

   이렇게 되면 좋은 점이 multitasking이 가능해진다

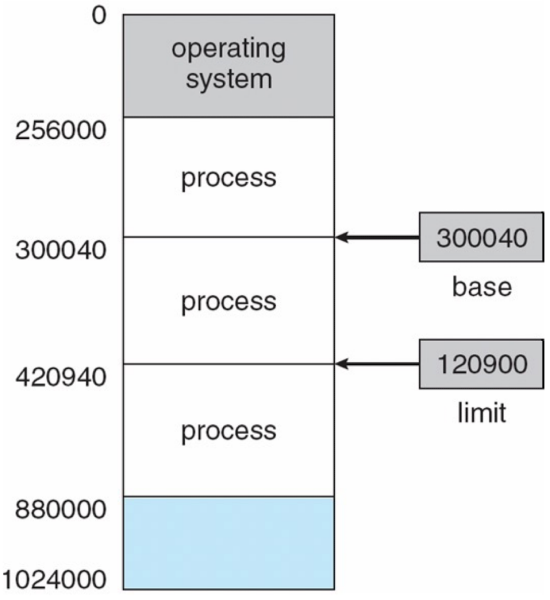
   그러나 한번 memory에 올라가면 실제 주소는 변경할 수 없게 된다

3. Execution time binding

   이거는 프로세스를 논리적 주소로만 기록해두고 실제 메모리에 올라갈 때 자동적으로 실제 물리적 주소로 변환되도록 만든다.

Base and Limit Registers

각각의 프로세스마다 시작주소와 프로세스의 길이를 이용해 주소를 정할 수 있다



이렇게 base register에 프로세스의 시작 주소를 기록하고 limit register에 프로세스의 길이를 저장한다

이 때 base register에 들어가는 시작 주소는 load time에 결정된다

(메모리에 올라가기 전까지는 프로세스의 길이만 저장해놨다가 메모리에 올라가면 주소 확정)

Logical vs Physical Address

1. Logical Address(Virtual address)

   CPU가 사용하는 프로세스의 주소, CPU는 기본적으로 프로세스의 모든 데이터가 메모리에 있다고 생각하고 데이터에 접근하려 함

2. Physical Address

   실제 DRAM에 존재하는 프로세스 주소

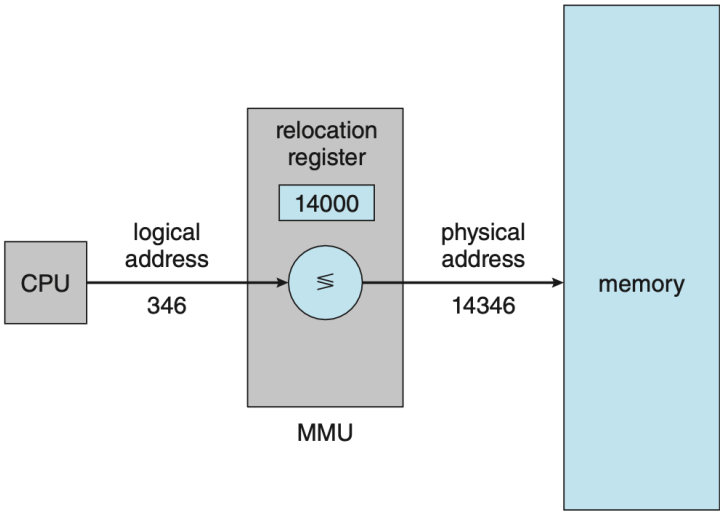
만약 램의 크기가 4GB인 컴퓨터를 쓰는데 프로세스의 크기가 6GB이면?

⇒ 그러면 일단 이걸 다 불러올 수 없으니까 문제돼, 그래서 일단 다 올라왔다고 가정하고 execution time 때마다 실제 DRAM에 원하는 프로세스의 데이터가 없으면 지속적으로 disk에서 꺼내서 DRAM으로 올려준다

Memory-Management는 logical과 physical의 간극을 어떻게 줄이는지가 핵심

Memory-Management Unit(MMU)

execution time 때 virtual address → physical address로 변환해주는 hardware 장치



CPU가 logical address만 들고 있고 이걸 MMU에다가 넣어주면 자동적으로 base address를 더해서 실제 주소를 찾을 수 있게 됨 (MMU가 프로세스의 시작 주소를 기억하고 있어)

Swapping necessitates dynamic relocation

1. MMU의 동작보니까 덧셈 하나 때문에 hardware를 굳이 만들어야 하냐?(연산량이 많아)
2. 만약 memory에 올라간 프로세스의 address가 변할 일이 뭐가 있을까? 애초에 고정되어 있으면 compile time에 주소 확정 지어버리면 되는거 아닌가?

   ⇒ memory에 올라간 process가 교체되는 경우

Swapping: process가 메모리에 있다가 disk로 내려간 다음에 다시 올라오면 기존에 실행되었던 주소랑 현재 주소가 달라짐 (virtual address는 동일)

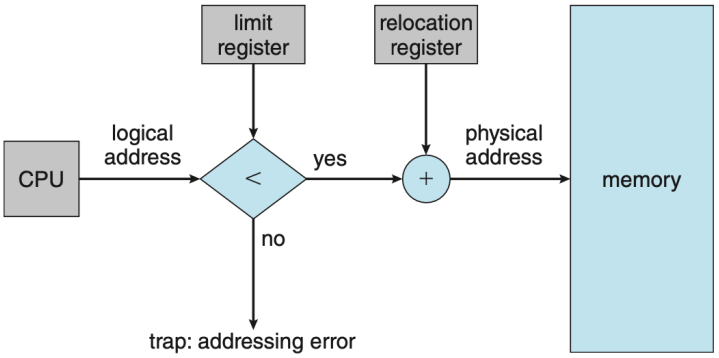
Backing store: process를 어떻게 빠르게 교체할 건지 (I/O 시간을 줄이는 방법)

Contiguous Allocation

이건 뭐 이름부터가 연속적으로 저장하는 걸 말하는거

프로세스를 다닥다닥 붙여서 저장하는 방법

MMU랑 비슷



limit register가 logical address의 한계를 기록

relocation register는 MMU에서 봤던 base register와 같은 역할

결국 logical address를 physical address로 변경

근데 MMU랑 다른 점은 이렇게 함으로써 프로세스가 disk로 내려갔다가 다시 올라와도 주소를 재할당 할 수 있어

근데 이러면 뭐가 문제냐?

⇒ Hole

모든 프로세스의 크기가 동일하진 않기 때문에 원래 100MB 짜리 프로세스가 들어왔다가 80MB 짜리 프로세스가 빈 공간에 들어가버리면 20MB가 남아버려

이러면 메모리 공간에 낭비가 생기기 때문에 문제를 해결해줘야 해

Dynamic Storage-Allocation Problem

hole이 생긴 공간을 어떻게 활용할 건지 3가지 기준으로 나눔

1. First-fit

   프로세스가 다시 올라올 때 들어갈 수 있는 첫번째 hole에 프로세스 등록

2. Best-fit

   가장 알맞은 hole에 넣는거 (가장 오차가 적은 Hole)

3. Worst-fit

   가장 큰 공간에 할당

3가지 다 장단점이 있는데 차례대로 살펴보면

1. First-fit

   장점: 무조건 첫번째 hole이기 때문에 찾는 시간이 단축

   단점: 만약 뒤에서 정말 크기와 완벽하게 일치하는 hole이 있으면 손해를 보는거

2. Best-fit

   장점: hole이 완벽하게 메워지면 당연히 좋아

   단점: 최대한 완벽하게 메우려다 보니 매우 작은 hole이 여러개 존재할 수 있어, 이런 경우 오히려 메모리를 낭비할 수 있게 됨 또한 항상 적합한 크기를 찾아야 하다보니 전체 탐색을 해야함

3. Worst-fit

   장점: 다음에 들어올 process가 적합한 hole을 찾기 유리해

   단점: 무조건 전체 탐색 해야돼서 시간이 오래 걸려, 만약 다음 프로세스 진입이 늦으면 hole이 계속 커

Fragmentation

1. External fragmentation: 실행이 안돼서 남아있는 공간
2. Internal fragmentation: 프로세스한테 할당은 됐는데, 프로세스가 실제로 사용하지 않는 경우

   ex) process한테 1024byte만큼 줬는데 당장 프로세스는 10byte만 쓰고 있는 경우

internal은 그냥 프로세스가 메모리를 적게 할당받아버리면 해결

external은 별도로 작업을 해줘야 함 ⇒ 이게 compaction(빈공간 없도록 쭉 당겨주는 거 생각하면 됨)

특히 db 같은 거 보면 file이 계속 커지기만 하고 줄어들지 않아서 compaction 작업을 주기적으로 해줘야 하는데 이게 overhead가 매우 큰 연산(db는 못 쓰게 막아두고 처리하면 된다치지만, 운영체제는 그렇게 해버리면 사용자가 뭘 못하게 되니까)

그래서 이걸 최대한 안 쓰는게 좋아

Paging

일단 저 Fragmentation 문제를 해결하려고 나온 개념

근데 paging은 external fragmentation 문제는 해결 하지만 internal fragmentation 문제는 해결 못해

paging은 logical address가 한번에 다 올라올 필요가 있냐? 에서 나온 아이디어

즉, CPU 입장에서 모든 프로세스가 동시에 다 올라왔다고 가정하고 실제 접근해서 데이터가 없을 때마다 disk에서 데이터 불러서 메모리로 올려

그래서 프로세스(logical memory)를 잘게 쪼개

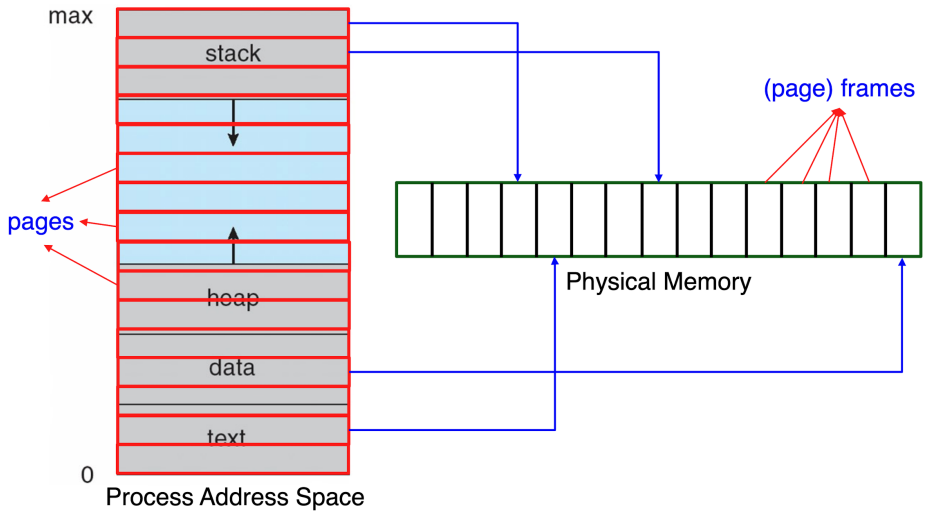
그리고 똑같이 physical memory도 잘게 쪼개

⇒ 이 때 쪼갤 때 physical memory의 단위랑 logiccal memory의 단위가 같아야 돼

physical memory에 딱 맞게 logical memory를 끼워 넣어

대신 이거할 때 page table이 있어야 하는데 logical address를 physical address로 변경 가능

CPU가 만약 현재 physical memory에 없는 데이터를 접근하려 하면 해당 페이지와 physical memory에 올라와있는 데이터를 교환



이렇게 잘 끼워넣을 수 있어요

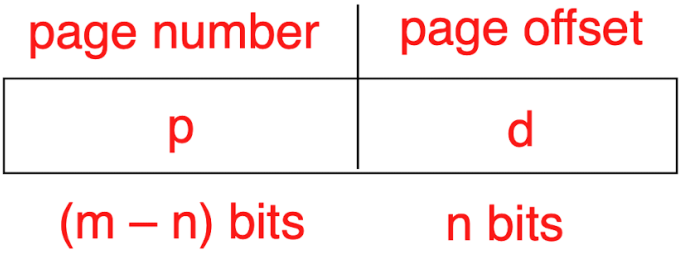
대신에 프로세스의 마지막 page는 웬만하면 external fragmentation이 존재

마지막 page가 아니면? 없지

마지막 page의 fragmentation 크기를 작게 하는 게 중요한데 이렇게 하려면 page 크기를 적절하게 잘 선택해야 됩니다

Address Translation Scheme in Paging

logical address를 physical address로 변경하는 방법



page number: 프로세스에서 몇번째 page인지

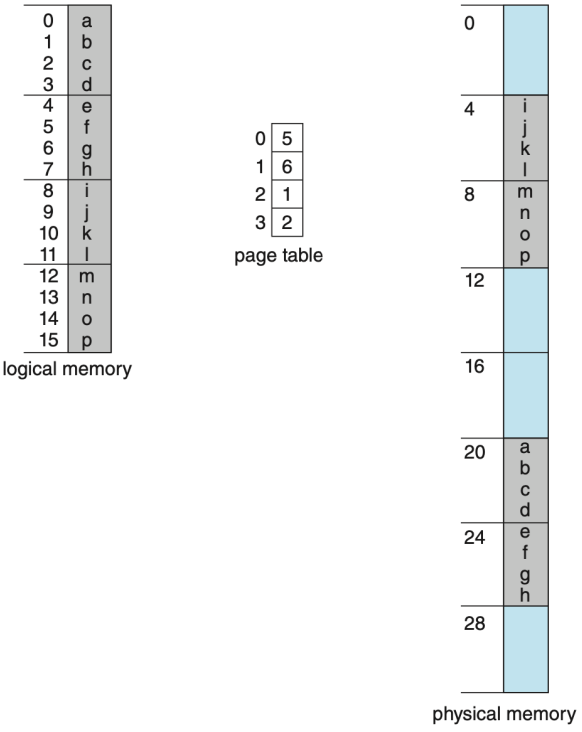
page offset: 그 page 안에서 주소가 어떻게 되는지

만약 page 크기를 4096byte로 잡았으면 page 크기로 12bit 필요

32bit 운영체제에서

page number: 20bit 사용

page offset: 12bit 사용



전체 주소가 0~15기 때문에 16만큼 필요 ⇒ 4bit

여기서 하나의 페이지의 크기가 4기 때문에 2bit 이용

남은 2bit로 page 개수 표현 가능

page number: 2bit

offset: 2bit

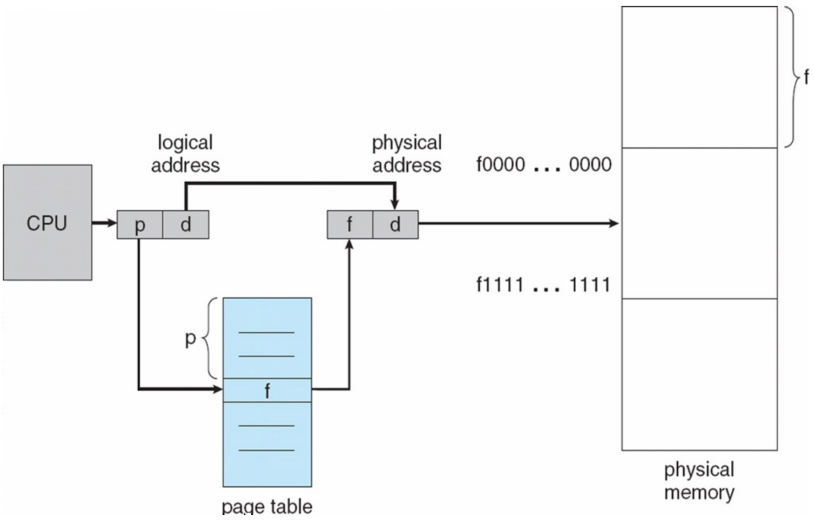
---

page table을 통해서 logical page number를 frame number와 mapping

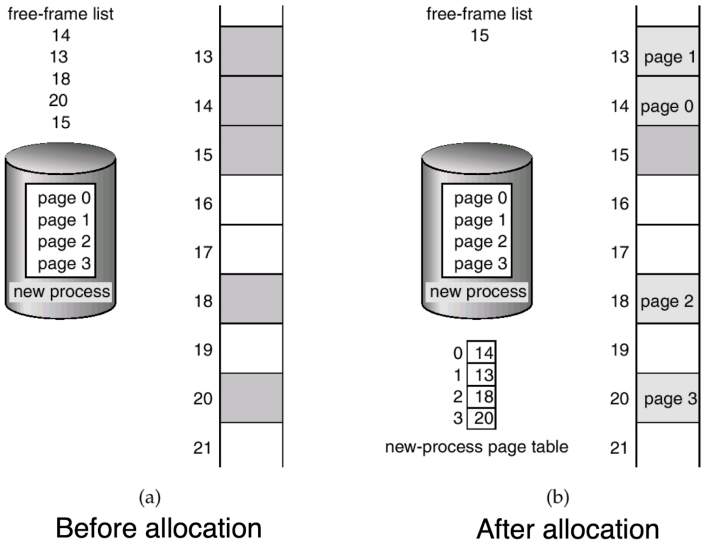
0번째 logical page → 5번 frame (실제 주소는 20)

1번째 logical page → 6번 frame (실제 주소는 24)

...



logical address를 page table을 통해서 몇번째 frame에 존재하는지 찾아야 한다.



현재 physical memory에서 비어있는 frame이 색칠된 영역이고

여기에 순차적으로 logical memory에 있는 page들이 딱 맞게 들어와

(free-frame list 순서대로)

여기까지 page table을 활용해서 frame 위치를 어떻게 찾는지 알겠어

근데 문제가 있어

page table은 기본적으로 메모리 내부에 저장되기 때문에 frame 찾는데 2번의 메모리 접근이 필요

(page table 접근 + 실제 frame 위치 접근)

⇒ 이거 때문에 속도가 느려

그래서 하드웨어의 도움을 받아서 문제를 해결하고자 함

⇒ sol) Translation Look-aside Buffer (TLB)

아예 캐시로 만들어버려서 page table에서 entry 찾는 시간(page table look up time)을 줄일 수 있게 됨

얘도 캐시기 때문에 context-switch 발생 때마다 한번씩 flush 됨

Associative Memory (TLB)

아니 그러면 결국 캐시도 메모리인데 큰 차이 없는거 아니냐? 라는 근본적인 질문을 할 수 있음

TLB에서 가장 큰 장점은 parallel search가 가능해

즉 우리가 일반적으로 생각하는 테이블을 순차적으로 탐색하는 것이 아니라, 한번에 똬라락 하면 한번에 찾아져

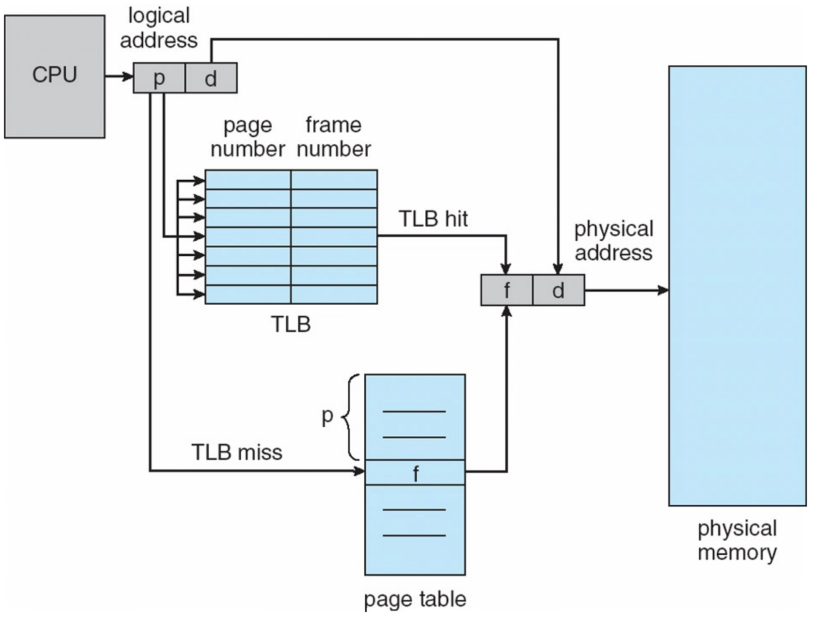
그래서 빠름

대신에 이거도 완전 무결하게 장점만 있는 건 아님

만약 TLB를 뒤졌는데 내가 원하는 페이지에 대한 정보가 없으면 결국 memory에 접근해야 해

이러면 원래 memory access 한번으로 끝날 거 TLB lookup time + memory access 로 더 오래 걸려

즉 한줄로 정리하면 이게 실질적으로 이득인지 살펴봐야 해



Effective Access Time

그래서 위에서 언급한 TLB miss 일 때 시간 비교

$$TLB\ Lookup\ Time\ = \alpha \qquad Memory\ Access\ Time = \beta$$

$$Hit\ ratio = \epsilon$$

여기서 중요한 점은 $\beta >>>>>> \alpha$ 다

그래서 최종적으로 EAT를 계산해보면

$$EAT = (\alpha+\beta)\epsilon + (\alpha+2\beta)(1-\epsilon)$$

$$= \alpha + (2-\epsilon)\beta$$

만약 TLB가 없다면 그냥 2β 나옴

(page table look up time + memory access time)

$\alpha + (2-\epsilon)\beta$ vs 2β

Hit ratio가 0보다 크고 1보다 작고 β가 α 보다 커서 결국 TLB를 쓰는게 더 이득

Memory Protection

page table에서 쓰고 있는지 쓰지 않는지 구별하기 위해 bit를 활용