21

페이징 : 개요

운영체제는 거의 모든 공간 관리 문제를 해결할 때 두 가지 중 하나를 사용한다. 첫 번째 방법은 우리가 가상 메모리의 **세그멘테이션**에서 보았듯이, *가변 크기의* 조각들로 분할하는 것이다. 불행하게도, 이 해결책은 태생적인 문제를 가지고 있다. 공간을 다양한 크기의 청크로 분할할 때 공간 자체가 **단편화**(**fragmented**) 될 수 있고, 할당은 점점 더 어려워진다.

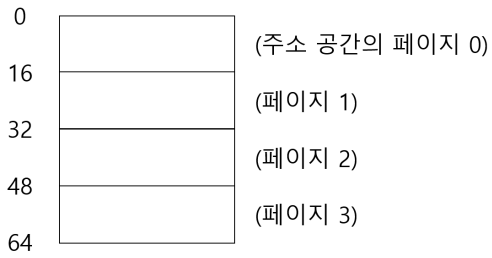
두 번째 방법인 공간을 *동일 크기의* 조각으로 분할하는 것을 고려해 볼 필요가 있다. 가상 메모리에서 이를 **페이징**(**paging**)이라 부르고 이 아이디어의 태생은 초기의 중요한 시스템인 Atlas까지 거슬러 올라간다 [Kil+62; Lav78]. 프로세스의 주소 공간을 몇 개의 가변 크기의 논리 세그멘트 (예, 코드, 힙, 스택)로 나누는 것이 아니라 고정 크기의 단위로 나눈다. 이 각각의 고정 크기 단위를 **페이지**(**page**)라고 부른다. 상응하여 물리 메모리도 **페이지 프레임**(**page frame**)이라고 불리는 고정 크기의 슬롯의 배열이라고 생각한다. 이 프레임 각각은 하나의 가상 메모리 페이지를 저장할 수 있다. 우리의 문제는 다음과 같다.

핵심 질문 : 페이지를 사용하여 어떻게 메모리를 가상화할 수 있을까
세그멘테이션의 문제점을 해결하기 위해 페이지를 사용하여 어떻게 메모리를 가상화할 수 있는가? 기본적인 기법은 무엇인가? 공간과 시간 오버헤드를 최소로 하면서 그 기법을 잘 동작하게 만들기 위한 방법은 무엇인가?

21.1 간단한 예제 및 개요

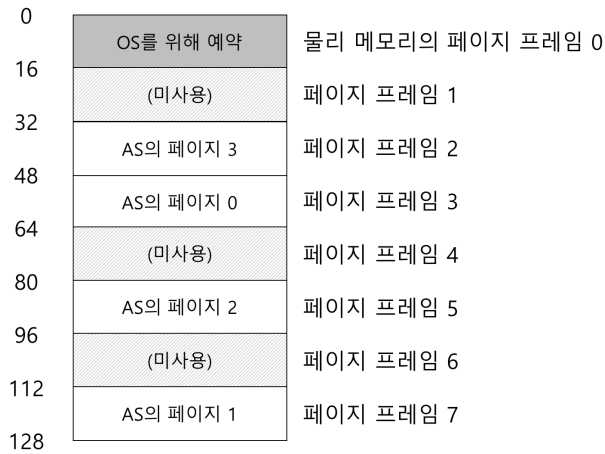
이 방법을 명확히 이해하기 위해 간단한 예를 통해 설명해 보자. 그림 21.1은 총 크기가 64바이트이면서 4개의 16바이트 페이지로 구성된 (가상 페이지 0, 1, 2, 4) 작은 주소 공간의 예를 보여준다. 물론 실제 주소 공간은 훨씬 커서 32비트의 경우 4 GB, 64비트의 경우에는 그보다 훨씬 크다[1].

1) 64비트 주소 공간은 상상하기 어려울 정도로 엄청나게 크다. 비유가 도움이 될 것이다. 32비트 주소 공간을 테니스 코트라고 생각하면 64비트 주소 공간은 유럽만하다(!).



〈그림 21.1〉 **간단한 64바이트 주소 공간**

물리 메모리는 그림 21.2에 나타낸 바와 같이, 고정 크기의 슬롯들로 구성된다. 이 경우 8개의 페이지 프레임, 총 128바이트의 비현실적으로 작은 물리 메모리이다. 그림에서 볼 수 있듯이, 가상 주소 공간의 페이지들은 물리 메모리 전체에 분산 배치되어 있다. 그림은 또한 운영체제가 자기자신을 위해서 물리 메모리의 일부를 사용하는 것도 보여 준다.



〈그림 21.2〉 **128바이트 물리 메모리에 탑재된 64바이트 주소 공간**

앞으로 보겠지만, 페이징은 이전 방식에 비해 많은 장점을 가지고 있다. 아마도 가장 중요한 개선은 *유연성*일 것이다. 페이징을 사용하면 프로세스의 주소 공간 사용 방식과는 상관없이 효율적으로 주소 공간 개념을 지원할 수 있다. 예를 들어, 힙과 스택이 어느 방향으로 커지는가, 어떻게 사용되는가에 대한 가정을 하지 않아도 된다.

또 다른 장점은 페이징이 제공하는 빈 공간 관리의 *단순함*이다. 예를 들어, 운영체제가 우리의 작은 64바이트 주소 공간을 8페이지 물리 메모리에 배치하기를 원한다고 할 때, 운영체제는 비어 있는 네 개의 페이지만 찾으면 된다. 아마 이를 위해 운영체제는 모든 비어 있는 페이지의 **빈 공간 리스트**를 유지하고 리스트의 첫 네 개 페이지를 선택할 것이다. 이 예에서, 운영체제는 주소 공간의 페이지 0을 물리 프레임 3에, 가상 페이지 1을 물리 프레임 7, 페이지 2를 프레임 5, 그리고 페이지 3을 프레임 2에 배치하였다.

페이지 프레임 1, 4, 6은 현재 비어 있다.

주소 공간의 각 가상 페이지에 대한 물리 메모리 위치 기록을 위하여 운영체제는 *프로세스 마다* **페이지 테이블**(**page table**)이라는 자료 구조를 유지한다. 페이지 테이블의 주요 역할은 주소 공간의 가상 페이지 **주소 변환**(**address translation**) 정보를 저장하는 것이다. 각 페이지가 저장된 물리 메모리의 위치가 어디인지 알려준다. 위의 간단한 예제의 경우(그림 21.2) 페이지 테이블은 다음과 같은 4개의 항목을 가지게 될 것이다(가상 페이지 0 → 물리 프레임 3), (VP 1 → PF 7), (VP 2 → PF 5) 및 (VP 3 → PF 2).

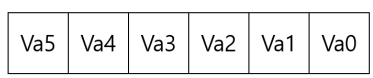
페이지 테이블은 프로세스마다 존재한다는 사실을 숙지해야 한다(우리가 논의하는 대부분의 페이지 테이블 구조는 *프로세스 마다* 존재하는 구조이다. **역 페이지 테이블**(**inverted page table**)이라는 예외적인 기법도 있다). 위의 예에서 다른 프로세스를 실행해야 한다면 운영체제는 이 프로세스를 위한 다른 페이지 테이블이 필요하다. 새 프로세스의 가상 페이지는 *다른* 물리 페이지에 존재하기 때문이다(공유 중인 페이지가 없다면).

이제 주소 변환 준비가 되었다. 작은 주소 공간(64바이트)을 가진 프로세스가 다음 메모리 접근을 수행한다고 가정하자.

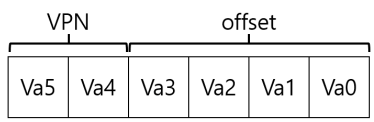
```
movl <virtual address>, %eax
```

구체적으로 주소 `<virtual address>`의 데이터를 `eax` 레지스터에 탑재하는 데에 집중하자(그 전에 일어나는 명령어 반입 단계는 무시하자).

프로세스가 생성한 가상 주소의 **변환**을 위해 먼저 가상 주소를 **가상 페이지 번호**(**virtual page number**, **VPN**)와 페이지 내의 **오프셋** 2개의 구성 요소로 분할한다. 이 예에서는 가상 주소 공간의 크기가 64바이트이기 때문에 가상 주소는 6비트가 필요하다($2^6 = 64$). 가상 주소를 개념적으로 다음 그림과 같이 생각할 수 있다.



이 그림에서 Va5는 가상 주소의 최상위 비트이며, Va0은 최하위 비트를 나타낸다. 우리는 페이지 크기(16바이트)를 알고 있기 때문에, 다음과 같이 가상 주소를 나눌 수 있다:

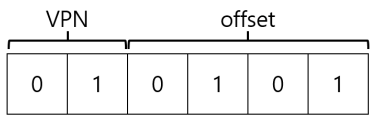


페이지 크기는 64바이트의 주소 공간에서 16바이트이다. 따라서 4페이지를 선택할 수 있어야 하고 따라서 주소의 최상위 2비트가 그 역할을 한다. 우리는 2비트 가상 페이지 번호 (VPN)를 가지게 된다. 나머지 비트는 페이지 내에서 우리가 원하는 바이트의 위치를 나타낸다. 이것을 오프셋이라고 부른다.

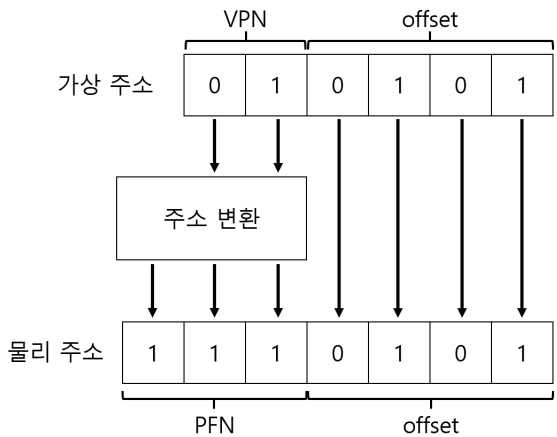
프로세스가 가상 주소를 생성하면 운영체제와 하드웨어가 의미있는 물리 주소로 변환한다. 예를 들어, 위 탑재 명령어의 가상 주소가 21이라고 하자.

```
movl 21, %eax
```

"21"을 이진 형식으로 변환하면 "010101"을 얻고, 이 가상 주소를 검사하고 가상 페이지 번호와 오프셋으로 나눈다.



따라서 가상 주소 "21"은 가상 페이지 "01"(또는 1)의 5번째("0101"번째) 바이트이다. 이 가상 페이지 번호를 가지고 페이지 테이블의 인덱스로 사용하여 가상 페이지 1이 어느 물리 프레임에 저장되어 있는지 찾을 수 있다. 위의 페이지 테이블에서 **물리 프레임 번호**(**physical frame number**, **PFN**) 혹은 **물리 페이지 번호**(**physical page number**, **PPN**)는 7(이진수 111)이다. VPN을 PFN으로 교체하여 가상 주소를 변환할 수 있다. 그런 후에 물리 메모리에 탑재 명령어를 실행한다(그림 21.3).



〈그림 21.3〉 **주소 변환 과정**

오프셋은 동일하다(즉, 변환되지 않는다)는 것에 주의하라. 오프셋은 페이지 *내에서의* 우리가 원하는 위치를 알려 주기 때문이다. 최종적으로 계산된 물리 주소는 1110101(십진수 117)이며, 이곳이 탑재할 데이터가 저장된 정확한 위치이다(그림 21.2).

이 개요를 염두에 두고, 이제 페이징에 관해 몇 가지 기본적인 질문을 해 보자(그리고 대답도 할 수 있길 바라면서). 페이지 테이블은 어디에 저장되는가 페이지 테이블의 내용은 무엇이며 테이블의 크기는 얼마인가? 페이징은 시스템을 (너무) 느리게 만들지는

않는가? 이들과 다른 질문에 대해서도 이제부터 부분적이라도 대답할 것이다. 계속 읽기 바란다.

21.2 페이지 테이블은 어디에 저장되는가

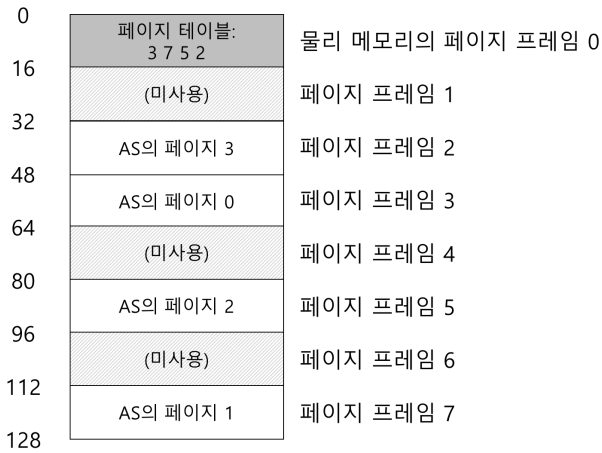
페이지 테이블은 매우 커질 수 있다. 우리가 이전에 논의했던 작은 세그멘트 테이블이나 베이스-바운드 쌍에 비해서 말이다. 예를 들어, 4 KB 크기의 페이지를 가지는 전형적인 32비트 주소 공간을 상상해 보자. 이 가상 주소는 20비트 VPN과 12비트 오프셋으로 구성된다(1 KB 페이지 크기를 위해 10비트가 필요하며, 4 KB 페이지를 위해서는 두 비트만 추가하면 된다).

20비트 VPN은 운영체제가 각 프로세스를 위해 관리해야 하는 변환의 개수가 2^{20}이라는 것을 의미한다(어림잡아 백만이다). 물리 주소로의 변환 정보와 다른 필요한 정보를 저장하기 위하여 **페이지 테이블 항목**(**page table entry**, **PTE**) 마다 4바이트가 필요하다고 가정하면, 각 페이지 테이블을 저장하기 위하여 4 MB의 메모리가 필요하게 된다. 이것은 꽤 커다란 크기이다. 프로세스 100개가 실행 중이라고 가정하자. 주소 변환을 위해서 운영체제가 400 MB의 메모리를 필요로 하는 것을 의미한다. 컴퓨터가 Giga byte 단위의 메모리를 가지고 있는 현재라도 변환을 위해서 이런 큰 청크를 사용한다는 것은 매우 비정상적이다. 64비트 주소 공간을 위한 페이지 테이블의 크기가 얼마나 클지에 대해서는 생각하기조차 싫다.

페이지 테이블이 매우 크기 때문에 현재 실행 중인 프로세스의 페이지 테이블을 저장할 수 있는 회로를 MMU 안에 유지하지 않을 것이다. 대신 각 프로세스의 페이지 테이블을 *메모리에* 저장한다. 당분간 페이지 테이블은 운영체제가 관리하는 물리 메모리에 상주한다고 가정하자. 나중에 운영체제 메모리 자체의 많은 부분이 가상화될 수 있다는 것을 알게 될 것이다. 페이지 테이블은 운영체제 가상 메모리에 저장할 수 있고 심지어 디스크에 스왑될 수 있다. 그러나 현재로선 매우 혼란스럽기 때문에 일단 무시한다. 그림 21.4에 운영체제 메모리 영역(PFN 0)에 페이지 테이블이 존재한다. 그림에 작은 변환 정보들이 보이는가?

21.3 페이지 테이블에는 실제 무엇이 있는가

페이지 테이블 구성에 대해 살펴보자. 페이지 테이블은 가상 주소(또는 실제로는 가상 페이지 번호)를 물리 주소(물리 프레임 번호)로 매핑(mapping)하는 데 사용되는 자료 구조이다. 임의의 자료 구조도 사용 가능하다. 가장 간단한 형태는 **선형 페이지 테이블**(**linear page table**)이다. 단순한 배열이다. 운영체제는 원하는 물리 프레임 번호(PFN)를 찾기 위하여 가상 페이지 번호(VPN)로 배열의 항목에 접근하고 그 항목의 페이지 테이블 항목(PTE)을 검색한다. 지금은 이 간단한 선형 구조를 사용한다고 가정한다. 이후의 장에서 페이징과 관련된 문제를 해결하기 위한 고급 자료 구조를 사용할 것이다.



〈그림 21.4〉 **예 : 커널 물리 메모리에 존재하는 페이지 테이블**

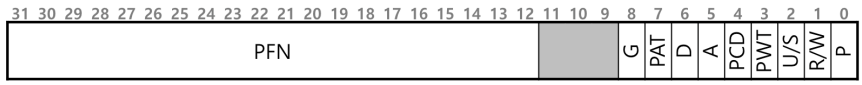
각 PTE에는 심도있는 이해가 필요한 비트들이 존재한다. **Valid bit**는 특정 변환의 유효 여부를 나타내기 위하여 포함된다. 예를 들어, 프로그램이 실행을 시작할 때 코드와 힙이 주소 공간의 한쪽에 있고 반대쪽은 스택이 차지하고 있을 것이다. 그 사이의 모든 미사용 공간은 **무효**(**invalid**)로 표시되고, 프로세스가 그런 메모리를 접근하려고 하면 운영체제에 트랩을 발생시킨다. 운영체제는 그 프로세스를 종료시킬 확률이 높다. Valid bit는 할당되지 않은 주소 공간을 표현하기 위해 반드시 필요하다. 주소 공간의 미사용 페이지를 모두 표시함으로써 이러한 페이지들에게 물리 프레임을 할당할 필요를 없애 대량의 메모리를 절약한다.

페이지가 읽을 수 있는지, 쓸 수 있는지, 또는 실행될 수 있는지를 표시하는 **protection bit**가 있다. Protection bit가 허용하지 않는 방식으로 페이지에 접근하려고 하면 운영체제에 트랩을 생성한다.

중요하지만 당장은 다루지 않는 몇 가지 다른 비트가 있다. **Present bit**는 이 페이지가 물리 메모리에 있는지 혹은 디스크에 있는지(즉, **스왑 아웃**되었는지) 가리킨다. 이런 기본적인 기법들은 나중에 물리 메모리보다 더 큰 주소 공간을 지원하기 위하여 주소 공간의 일부를 **스왑**하는 방법에 대해 배울 때 좀 더 자세히 이해할 것이다. 스와핑은 운영체제가 드물게 사용되는 페이지를 디스크로 이동시켜 물리 메모리를 비울 수 있게 한다. **dirty bit** 또한 일반적인데, 메모리에 반입된 후 페이지가 변경되었는지 여부를 나타낸다.

reference bit(또는 **accessed bit**)는 때때로 페이지가 접근되었는지를 추적하기 위해 사용된다. 또한, 어떤 페이지가 인기가 있는지 결정하여 메모리에 유지되어야 하는 페이지를 결정하는 데에도 유용하다. 이 정보는 페이지 교체에 매우 중요하다. **페이지 교체**에 대해서는 이후의 장에서 매우 자세하게 다룬다.

그림 21.5는 x86 아키텍처의 페이지 테이블 항목을 보여 준다 [Int09]. 이 항목은 다음과 같은 비트들을 가진다. Present bit(P), 이 페이지에 쓰기가 허용되는지 결정하



〈그림 21.5〉 **x86 페이지 테이블 항목(PTE)**

는 읽기/쓰기 비트(R/W), 사용자 모드 프로세스가 페이지에 액세스 할 수 있는지를 결정하는 사용자/슈퍼바이저 비트(U/S), 이 페이지에 대한 하드웨어 캐시의 동작을 결정하는 몇몇 비트(PWT, PCD, PAT 및 G), reference bit(A)와 dirty bit(D), 그리고 마지막으로 페이지 프레임 번호(PFN) 자체.

x86 페이징 지원에 대한 자세한 내용은 Intel Architecture Manuals [Int09]를 참조하기 바란다. 한 가지 미리 숙지할 사항이 있다. 매뉴얼을 읽는 것은 (매우 유익하고 운영체제에서 페이지 테이블을 사용하는 코드를 작성하는 사람에게는 반드시 필요한 작업이지만) 처음에는 쉽지 않다. 인내와 열정이 있어야 한다.

21.4 페이징: 너무 느림

페이지 테이블의 크기가 메모리 상에서 매우 크게 증가할 수 있다. 페이지 테이블로 인해 처리 속도가 저하될 수 있다. 간단한 명령어를 예로 들어 보자.

```
movl 21, %eax
```

주소 21에 대한 참조만 고려하고 명령어 반입에 대해서는 고려하지 않기로 하자. 이 예에서 하드웨어가 주소 **변환**을 담당한다고 가정한다. 원하는 데이터를 가져 오기 위해, 먼저 시스템은 가상 주소(21)를 정확한 물리적 주소(117)로 변환해야 한다. 주소 117에서 데이터를 반입하기 전에 시스템은 프로세스의 페이지 테이블에서 적절한 페이지 테이블 항목을 가져와야 하고, 변환을 수행한 후, 물리 메모리에서 데이터를 탑재한다.

이렇게 하기 위해서, 하드웨어는 현재 실행 중인 프로세스의 페이지 테이블의 위치를 알아야 한다. 당분간 하나의 **페이지 테이블 베이스 레지스터(page table base register)**가 페이지 테이블의 시작 주소(물리 주소)를 저장한다고 가정한다. 원하는 PTE의 위치를 찾기 위해 하드웨어는 다음과 같은 연산을 수행한다.

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
PTEAddr = PageTableBaseRegister + (VPN * sizeof(PTE))
```

이 예제에서 VPN_MASK는 `0x30`(16진수 `30` 또는 이진수 `110000`)으로 설정되고, 전체 가상 주소에서 VPN 비트만 골라낸다. SHIFT는 4로 설정되고(오프셋 비트 수), 올바른 정수 가상 페이지 번호를 형성하기 위해 VPN 비트를 오른쪽으로 이동시킨다. 가상 주소 21(`010101`)을 마스킹 하면 `010000`이 되고; 쉬프트는 01 또는 우리가 원하는 가상 페이지 1로 변환한다. 우리는 이 값을 페이지 테이블 베이스 레지스터가 가리키는 PTE 배열에 대한 인덱스로 사용한다.

이 물리 주소가 알려지면 하드웨어는 메모리에서 PTE를 반입할 수 있고, PFN을 추출하고, 가상 주소의 오프셋과 연결하여 원하는 물리 주소를 만든다. 구체적으로, PFN

을 SHIFT 만큼 왼쪽으로 쉬프트 하고 오프셋과 논리적 OR 연산을 하여 최종 주소를 형성한다고 생각할 수 있다.

```
offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

마지막으로, 하드웨어는 메모리에서 원하는 데이터를 가져와서 eax 레지스터에 넣을 수 있다. 이제 프로그램은 메모리로부터 값을 성공적으로 탑재하였다!

정리를 위하여, 각 메모리 참조 시 일어나는 세부 동작을 살펴보자. 그림 21.6은 기본적인 방식을 보여준다. 모든 메모리 참조에 대해(명령어 반입이건, `load` 또는 `store` 명령어의 실행이든) 먼저 페이지 테이블에서 변환 정보를 반입해야 하기 때문에 반드시 한 번의 추가적인 메모리 참조가 필요하다. 엄청난 양의 작업이다. 메모리 참조는 비용이 비싸고 이 경우에 프로세스는 2배 이상 느려진다.

```
1  // 가상 주소에서 VPN 추출
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3  // 페이지 테이블 항목(PTE)의 주소 형성
4  PTEAddr = PTBR + (VPN * sizeof(PTE))
5  // PTE 반입
6  PTE = AccessMemory(PTEAddr)
7  // 프로세스가 페이지를 접근할 수 있는지 확인
8  if (PTE.Valid == False)
9      RaiseException(SEGMENTATION_FAULT)
10 else if (CanAccess(PTE.ProtectBits) == False)
11     RaiseException(PROTECTION_FAULT)
12 else
13     // 접근 가능하면 물리 주소 만들고 값 가져오기
14     offset = VirtualAddress & OFFSET_MASK
15     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
16     Register = AccessMemory(PhysAddr)
```

〈그림 21.6〉 **페이징을 사용한 메모리 접근**

이제 우리가 반드시 해결해야 할 *두 개의* 진짜 문제를 알게 되었다. 하드웨어와 소프트웨어의 신중한 설계 없이는 페이지 테이블로 인해 시스템이 매우 느려질 수 있으며 너무 많은 메모리를 차지한다. 페이징이 메모리 가상화에 필요한 훌륭한 해결책처럼 보이지만 먼저 이 두 가지 중요한 문제가 해결되어야 한다.

21.5 메모리 트레이스

마치기 전에, 간단한 메모리 액세스 예를 통해 페이징을 사용했을 때 발생하는 모든 메모리 접근을 살펴보자. 우리가 관심있는 코드는(파일 `array.c`의 C 언어로 된) 다음과 같다.

```
1 int array[1000];
2 ...
3 for (i = 0; i < 1000; i++)
4     array[i] = 0;
```

우리는 `array.c`를 컴파일하고 다음과 같이 실행한다.

여담: 자료 구조—페이지 테이블

현대 운영체제의 메모리 관리 서브시스템에서 가장 중요한 자료 구조 중 하나는 **페이지 테이블**이다. 일반적으로, 페이지 테이블은 **가상-물리 주소 변환**(**virtual-to-physical address translation**)을 저장하여 주소 공간의 각 페이지의 물리 메모리 위치를 알 수 있게 한다. 각 주소 공간은 이런 변환을 필요로 하기 때문에 페이지 테이블은 프로세스마다 하나씩 존재한다. 페이지 테이블의 정확한 구조는 하드웨어(예전 시스템)에 의해 결정되거나 융통성 있게 운영체제에 의해 관리된다(현대 시스템).

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

물론, 이 코드(단순히 배열을 초기화한다)가 어떤 메모리 접근을 생성할지에대해서 진짜로 이해하려면 우리는 몇 가지 사실을 더 알아야 한다(또는 가정해야 한다). 먼저, 루프 안에서 배열을 초기화하기 위해 어떤 어셈블리 명령어를 사용하는지 보기 위하여 결과 이진 파일을 **디스어셈블**(**disassemble**) 해야만 한다(Linux에서는 `objdump`를, Mac에서는 `otool`을 사용하여). 여기 결과 어셈블리 코드가 있다.

```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

x86을 조금 알고 있다면, 위 코드는 이해하기 아주 쉽다[2]. 첫 번째 명령어는 값 `0($0x0)`을 가상 메모리 주소로 옮긴다. 0 값이 저장될 가상 메모리 주소는 `%edi`의 값을 `%eax`의 4배에다 더해서 계산된다. `%edi`는 배열의 시작 주소를 저장하고, `%eax`는 배열 인덱스(`i`)를 저장한다. 배열이 정수 배열이고 정수의 크기는 4바이트이기 때문에 4를 곱한다.

두 번째 명령어는 `%eax`에 저장된 배열 인덱스를 증가시키고, 세 번째 명령어는 `%eax`의 값과 16진수 `0x03e8` 또는 십진수 `1000`을 비교한다. 비교 결과 아직 두 값이 같지 않다면(`jne` 명령어가 검사하는 것), 네 번째 명령어는 루프의 상단으로 다시 분기한다.

이 명령어 시퀀스가 (가상 및 물리 수준 모두에서) 어떤 메모리 접근을 생성하는지 이해하기 위해서, 코드와 배열의 가상 메모리의 주소와 페이지 테이블의 위치에 대해서 몇 가지 가정을 해야 한다.

이 예에서 크기가 64 KB인(비현실적으로 적은) 가상 주소 공간을 가정한다. 또 페이지 크기는 1 KB로 가정한다.

우리가 알아야 할 것은 페이지 테이블의 내용과 위치이다. 선형(배열 기반) 페이지 테이블이고 물리 주소 1 KB (1024)에 위치한다고 가정하자.

이 예에서 신경 써야 할 몇 개의 페이지 테이블의 페이지가 있다. 첫째, 코드가 상주하는 가상 페이지가 있다. 페이지 크기가 1 KB이기 때문에, 가상 주소 1,024는

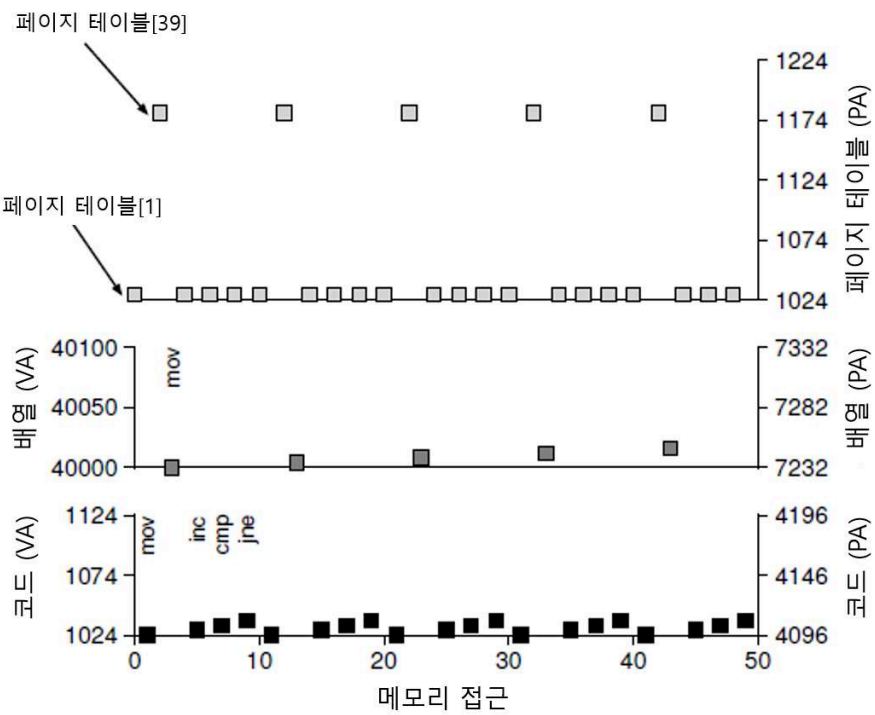
2) 여기서 약간의 부정행위를 하고 있다. 간단하게 하기 위해 각 명령어의 길이는 4바이트라고 가정한다. 실제로는 x86은 가변 길이 명령어를 사용한다.

가상 주소 공간의 두 번째 페이지(VPN=1, VPN=0이 첫 번째 페이지이기 때문에)에 상주한다. 이 가상 페이지가 물리 프레임(4)에 매핑된다고 가정하자(VPN 1 → PFN 4).

다음으로 배열 자체가 있다. 크기는 4000바이트(1000개의 정수)이고, 가상 주소 40000에서 44000까지(마지막 바이트 포함하지 않음) 존재한다고 가정한다. 이 범위에 해당하는 가상 페이지는 VPN=39 ... VPN=42이다. 우리는 이 페이지에 대한 매핑이 필요하다. 이 가상-대-물리 주소 매핑이 다음과 같다고 가정하자: (VPN 39 → PFN7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10).

이제 프로그램의 메모리 참조를 살펴볼 준비가 되었다. 프로그램이 실행되면, 각 명령어의 반입 시에 메모리가 두 번 참조한다: 명령어의 위치를 파악하기 위해 페이지 테이블 접근 한 번 그리고 명령어 자체에 한 번. 게다가 `mov` 명령어는 메모리 참조를 한 번 한다. 이 명령도 먼저 페이지 테이블 접근 한 번(배열의 가상 주소를 올바른 물리 메모리 주소로 변환하기 위하여), 다음 배열 자체를 접근하기 위하여 한 번의 참조가 필요하다.

처음 다섯 번의 루프 반복에 대해 전체 과정이 그림 21.7에 나와 있다. 가장 아래쪽 그래프가 명령어 메모리 참조다(가상 주소가 왼쪽, 물리 주소가 오른쪽). 중앙의 그래프는 배열에 대한 접근이다(가상 주소가 왼쪽, 물리 주소가 오른쪽). 마지막으로, 맨 위의



〈그림 21.7〉 **가상(및 물리) 메모리 추적**

그래프는 페이지 테이블 메모리 접근이다(물리 주소만, 이 예에서 페이지 테이블은 물리적 메모리만 존재하기 때문에). 전체 트레이스에서, x축은 루프가 처음 다섯 번 반복되는 과정에서의 메모리 접근을 보인다. 루프당 10번의 메모리 접근이 존재한다. 네 번의 명령어 반입, 한 번의 메모리 갱신, 그리고 이러한 네 번의 반입과 한 번의 명시적인 갱신을 위한 주소 변환을 위한 총 다섯 번의 페이지 테이블 접근 등이다.

이 그래프를 이해할 수 있는지 확인하라. 구체적으로 처음 다섯 번의 루프를 넘어서 계속 실행되면 무엇이 바뀌는가? 어떤 새로운 메모리가 접근되는가? 어떻게 그것을 알아낼 수 있을까?

이것은 가장 간단한 예(몇 줄 안되는 C 코드)이지만, 실제 응용 프로그램의 메모리 동작이 얼마나 복잡한 일인지 알았을 것이다. 걱정하지 마라. 앞으로 더 복잡해진다. 우리가 앞으로 소개할 기법은 기존 기법들을 더 복잡하게 할 것이다. 미안![3]

21.6 요약

메모리 가상화에 대한 해결책으로 **페이징**을 소개하였다. 페이징은 이전 방식에 비해 많은 장점을 가지고 있다(세그멘테이션 등과 같은). 먼저 페이징은 (설계에 의해) 메모리를 고정 크기의 단위로 나눈다. 둘째, 가상 주소 공간의 드문 사용을 허용한다.

페이징을 제대로 구현하지 못하면 컴퓨터가 매우 느려지고(페이지 테이블 접근을 위한 많은 추가적 접근) 뿐만 아니라 메모리 낭비(유용한 응용 데이터 대신 페이지 테이블로 가득 참)까지 초래한다. 제대로 동작하는 페이징 시스템 고안을 위해서는 더 많은 노력을 해야 한다. 다음 두 장은 어떻게 할 수 있는지 보여줄 것이다.

3) 우리가 진짜로 미안한 건 아니다. 그러나 말이 된다면 미안하지 않은 것에 대해 미안하다.

참고 문헌

[Int09] **"Intel 64 and IA-32 Architectures Software Developer's Manuals"**
Intel
URL: `http://www.intel.com/products/processor/manuals`
특히 "Volume 3A: System Programming Guide Part 1"과 "Volume 3B: System Programming Guide Part 2"에 관심을 두라.

[Kil+62] **"One-level Storage System"**
T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner
IRE Trans. EC-11, 2 (1962), pp. 223-235
(Reprinted in Bell and Newell, "Computer Structures: Readings and Examples" McGraw-Hill,New York, 1971).
Atlas는 메모리를 고정 크기의 페이지로 나누는 아이디어에 관해 선도적이었고 많은 의미에서 현대 컴퓨터 시스템에서 볼 수 있는 메모리 관리 형식의 초창기 형태였다.

[Lav78] **"The Manchester Mark I and atlas: a historical perspective"**
S.H. Lavington
Communications of the ACM archive Volume 21, Issue 1 (January 1978), pp. 4-12 Special issue on computer architecture
이 논문은 몇 가지 중요한 컴퓨터 시스템 개발의 역사의 일부를 회고한다. 미국에서는 때때로 잊어버리지만 이 새로운 아이디어의 대부분은 해외에서 왔다.

숙제

이 과제에서는 `paging-linear-translate.py`라는 간단한 프로그램을 사용하게 될 것이다. 이 프로그램을 이용하여 선형 페이지 테이블을 이용한 단순한 가상-대-물리 주소 변환이 어떻게 작동하는지를 이해하고 있는지확인할 수 있다. 자세한 사항은 README를 참조하시오.

문제

1. 변환을 하기 전에 매개변수가 주어졌을 때 선형 페이지 테이블이 크기를 변경하는 방법을 연구하기 위해 시뮬레이터를 사용해 보자. 매개변수가 바뀜에 따라 선형 페이지 테이블의 크기를 계산하시오. 추천하는 입력은 아래와 같다. `-v` 플래그를 사용하여 얼마나 많은 페이지 테이블 항목이 채워지는지 알 수 있다.

   먼저, 주소 공간이 커짐에 따라 선형 페이지 테이블 크기의 변화를 이해하기 위하여:

   ```
   paging-linear-translate.py -P 1K -a 1m -p 512m -v -n 0
   paging-linear-translate.py -P 1K -a 2m -p 512m -v -n 0
   paging-linear-translate.py -P 1K -a 4m -p 512m -v -n 0
   ```

   그런 다음 페이지 크기가 커짐에 따라 선형 페이지 테이블 크기의 변화를 이해하기 위하여:

   ```
   paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
   paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
   paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
   ```

   어느 것 하나라도 실행하기 전에 예상 경향에 대해 생각해 보라. 주소 공간이 커지면 페이지 테이블 크기는 어떻게 변하는가? 페이지 크기가 커지는 경우는? 일반적으로 큰 페이지 크기를 사용하면 안 되는 이유는 무엇인가?

2. 자 이제 변환을 해 보자. 몇 가지 작은 예부터 시작해서 `-u` 플래그를 사용하여 주소 공간에 할당된 페이지 개수를 변경해 보라. 예를 들면:

   ```
   paging-linear-translate.py -P 1k -a 16k -p 32K -v -u 0
   paging-linear-translate.py -P 1k -a 16k -p 32K -v -u 25
   paging-linear-translate.py -P 1k -a 16k -p 32K -v -u 50
   paging-linear-translate.py -P 1k -a 16k -p 32K -v -u 75
   paging-linear-translate.py -P 1k -a 16k -p 32K -v -u 100
   ```

   각 주소 공간에 할당된 페이지의 비율을 높인다면 어떤 일이 벌어지는가?

3. 그러면 몇 가지 무작위 배정을 시도하자. 몇 가지 다른 (그리고 때로는 정말 어이 없는) 주소 공간 매개변수를 사용해 보라:

   ```
   paging-linear-translate.py -P 8  -a 32   -p 1024 -v -s 1
   paging-linear-translate.py -P 8K -a 32k  -p 1m   -v -s 2
   paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
   ```

이 조합 중 어느 조합이 비현실적인가? 이유는?

4. 다른 문제를 시험해 보기 위해 프로그램을 사용하라. 프로그램이 더 이상 작동하지 않는 제약을 찾을 수 있는가? 예를 들어, 만약 주소 공간의 크기가 물리 메모리보다 *크다면* 어떤 일이 벌어지는가?