19

세그멘테이션

지금까지 프로세스 주소 공간 전체를 메모리에 탑재하는 것을 가정해 왔다. 베이스와 바운드 레지스터를 사용하면 운영체제는 프로세스를 물리 메모리의 다른 부분으로 쉽게 재배치할 수 있다. 이러한 형태의 주소 공간에 대해 재미있는 사실을 발견했을 것이다. 스택과 힙 사이에 사용되지 않는 큰 공간이 존재한다.

그림 19.1에서 볼 수 있듯이 스택과 힙 사이의 공간은 사용되지 않더라도 주소 공간을 물리 메모리에 재배치할 때 물리 메모리를 차지한다. 베이스와 바운드 레지스터 방식은 메모리 낭비가 심하다. 또한, 주소 공간이 물리 메모리보다 큰 경우 실행이 매우 어렵다. 이런 측면에서 볼 때 베이스와 바운드 방식은 유연성이 없다.

핵심 질문: 대용량 주소 공간을 어떻게 지원하는가

스택과 힙 사이에 잠재적으로 큰 빈 영역이 존재하는 주소 공간을 어떻게 지원하는가? 우리의 예에서 사용하는 작은 주소 공간에서는 낭비는 그렇게 심각하지 않다. 그러나 크기가 4 GB인 32비트 주소 공간을 상상해 보라; 통상 프로그램은 단지 수 메가바이트만 사용함에도 불구하고 주소 공간 전체가 메모리에 탑재되어야 한다.

19.1 세그멘테이션: 베이스/바운드(base/bound)의 일반화

이 문제를 해결하기 위한 아이디어가 탄생했으니, 바로 **세그멘테이션**(**segmentat-ion**)이다. 상당히 오래된 아이디어로서 적어도 1960년대 초까지 거슬러 올라간다 [Hol61; Gre62]. 아이디어는 간단하다. MMU 안에 오직 하나의 베이스와 바운드 쌍만 존재하는 것이 아니라 주소 공간의 논리적인 **세그멘트**(**segment**) 마다 베이스와 바운드 쌍이 존재한다. 세그멘트는 특정 길이를 가지는 연속적인 주소 공간이다. 우리가 기준으로 삼은 주소 공간에는 코드, 스택, 및 힙의 세 종류의 세그멘트가 있다. 세그멘테이션을 사용하면 운영체제는 각 세그멘트를 물리 메모리의 각기 다른 위치에 배치할 수 있고, 사용되지 않는 가상 주소 공간이 물리 메모리를 차지하는 것을 방지할 수 있다.

예를 들어 보자. 그림 19.1의 주소 공간을 물리 메모리에 배치하려고 한다. 각 세그멘트의 베이스와 바운드 쌍을 이용하여 세그멘트들을 독립적으로 물리 메모리에 배치할

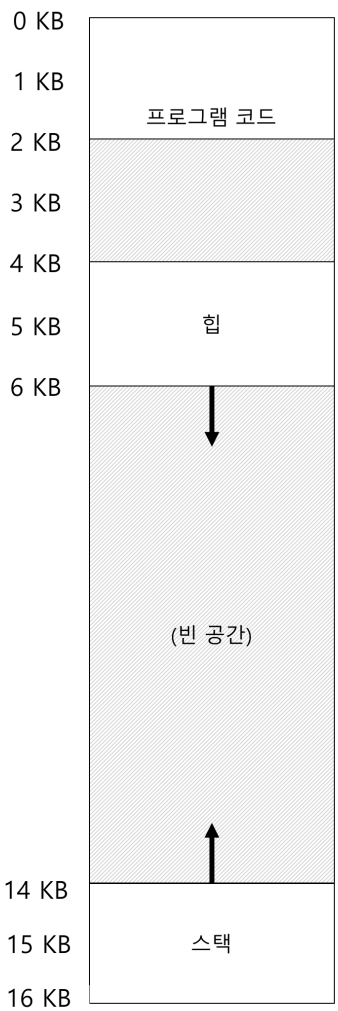
수 있다. 그림 19.2를 보자. 64 KB의 물리 메모리에 3개의 세그멘트와 운영체제용으로 예약된 16 KB 영역이 존재한다.

그림에서 볼 수 있듯이, 사용 중인 메모리에만 물리 공간이 할당된다. 사용되지 않은 영역이 많은 대형 주소 공간을(우리는 때때로 **드문드문 사용되는 주소 공간**(**sparse address space**)이라고 부름) 수용할 수 있다.

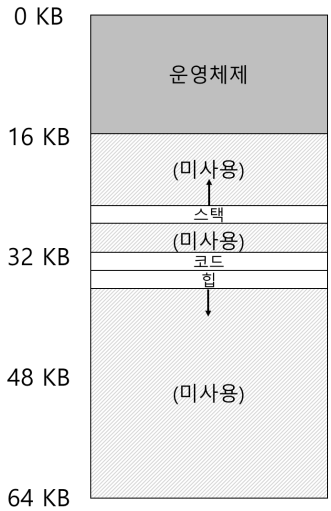
세그멘트 지원을 위한 MMU 하드웨어 구조는 예상한 것과 같다. 이 예제의 경우 3쌍의 베이스와 바운드 레지스터 집합이 필요하다. 그림 19.3은 앞의 예에 해당하는 각 레지스터의 값을 보여준다. 각 바운드 레지스터는 세그멘트의 크기를 저장한다.

그림에서 코드 세그멘트가 물리 주소 32 KB에 배치되고 크기는 2 KB이며, 힙 세그멘트가 34 KB에 배치되고 역시 크기는 2 KB라는 것을 알 수 있다.

그림 19.1의 주소 공간을 사용하여 주소 변환을 해 보자. 가상 주소 100 번지를



〈그림 19.1〉 **주소 공간 (복습)**



〈그림 19.2〉 **물리 메모리에 세그멘트 배치하기**

세그멘트	베이스	크기
코드	32 KB	2 KB
힙	34 KB	2 KB
스택	28 KB	2 KB

〈그림 19.3〉 **세그멘트 레지스터의 값**

여담: 세그멘트 폴트

용어 Segment Fault는 세그멘트 사용 시스템에서 불법적인 주소 접근 시 발생한다. 재미있게도 이 용어는 세그멘트에 대한 지원이 전혀 없는 컴퓨터에서도 여전히 사용된다는 사실이다. 이 경우에는 코드의 오류 원인을 알 수도 없다.

참조한다고 가정하자. 가상 주소 100번지는 코드 세그멘트에 속한다. 참조가 일어나면 하드웨어는 베이스 값에 이 세그멘트의 *오프셋*(이 경우, 100)을 더해 물리 주소는 100+32 KB 또는 32868이 된다. 그 후, 주소가 범위 내에 있는지 검사하고(100은 2 KB보다 작다), 범위 내에 있을 경우, 물리 메모리 주소 32868를 읽는다.

가상 주소 4200의 힙을 살펴보자(다시 그림 19.1을 보시오). 주의할 점이 있다. 가상 주소 4200을 힙의 베이스(34 KB)에 더하면 물리 주소 39016을 얻지만 이 주소는 올바른 물리 주소가 아니다. 먼저 힙 안에서의 *오프셋*, 즉 주소가 참조하는 바이트가 *이 세그멘트* 시작으로부터 몇 번째 바이트인지를 얻어야 한다. 힙은 가상 주소 4 KB(4096)에서 시작하기 때문에 오프셋 4200은 실제로는 4200 빼기 4096, 즉 104가 된다. 이 오프셋(104)을 베이스 레지스터의 물리 주소(34 KB)에 더해 원하는 결과 34920을 얻게

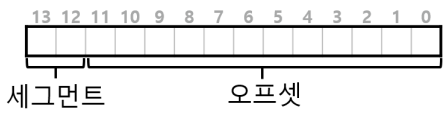
된다.

만일 힙의 마지막을 벗어난 7KB와 같은 잘못된 주소를 접근하려고 한다면 어떠한 일이 벌어질까? 하드웨어가 그 주소가 범위를 벗어났다는 것을 감지하고 운영체제에 트랩을 발생시킨다. 운영체제는 아마도 문제의 프로세스를 종료시킬 가능성이 크다. 이제 모든 C 프로그래머가 두려워하는 유명한 용어의 기원을 알게 되었다: **세그멘트 위반**(**segment violation**) 또는 **세그멘트 폴트**(**segment fault**).

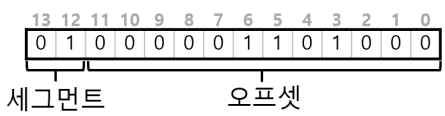
19.2 세그멘트 종류의 파악

하드웨어는 변환을 위해 세그멘트 레지스터를 사용한다. 하드웨어는 가상 주소가 어느 세그멘트를 참조하는지 그리고 그 세그멘트 안에서 오프셋은 얼마인지를 어떻게 알 수 있는가?

한 가지 일반적인 접근법, 가상 주소의 최상위 몇 비트를 기준으로 주소 공간을 여러 세그멘트로 나누는 것이다. 이 기법은 VAX/VMS 시스템에서 사용되었다 [LL82]. 위의 예에서는 3개의 세그멘트가 있다. 주소 공간을 세그멘트로 나누기 위해서는 2비트가 필요하다. 세그멘트를 표시하기 위해 가상 주소 14비트 중 최상위 2비트를 사용하는 경우 가상 주소의 모양은 다음과 같이 보일 것이다.



우리의 예에서 최상위 2비트가 00이면, 하드웨어는 가상 주소가 코드 세그멘트를 가리킨다는 것을 알고, 따라서 코드 세그멘트의 베이스와 바운드 쌍을 사용하여 주소를 정확한 물리 메모리에 재배치한다. 최상위 2비트가 01이면 하드웨어는 주소가 힙 세그멘트라는 것을 인지하여 힙의 베이스와 바운드를 사용한다. 이해를 돕기 위해서 앞에서 사용한 힙에 해당하는 가상 주소(4200)를 변환해 보자. 가상 주소 4200에 해당하는 이진 형식은 다음과 같다.



그림에서 볼 수 있듯이 최상위 2비트(01)는 하드웨어에게 참조하는 세그멘트의 종류를 알려준다. 하위 12비트는 세그멘트 내의 오프셋이다. `0000 0110 1000` 또는 16진수 `0x068` 또는 10진수로 `104`. 하드웨어는 세그멘트 레지스터를 파악하는 데 처음 2비트를 이용하고 세그멘트 오프셋으로 다음 12비트를 취한다. 오프셋에 베이스 레지스터 값을 더하여 하드웨어는 최종 물리 주소를 계산한다. 오프셋은 바운드 검사도 쉽게 만든다. 오프셋이 바운드보다 작은지 여부만 검사하면 된다. 그렇지 않으면 주소가

잘못된 것이다. 베이스와 바운드 쌍을 배열 형식으로 저장할 경우(세그멘트당 하나의 항목), 원하는 물리 주소를 얻기 위하여 다음과 같은 작업을 하게 된다.

```
1 // 14 bit VA중의 상위 2 bit를 얻어옴
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // 이제 오프셋을 얻어옴
4 Offset = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

우리는 현재 예를 기준으로 위 코드에서 사용된 상수 값들을 정할 수 있다. `SEG_MASK`는 `0x3000`, `SEG_SHIFT`는 12, 그리고 `OFFSET_MASK`는 0xFFF로 지정된다.

세그멘트 종류를 나타내는 데 최상위 2비트를 사용하고 주소 공간에는 세 개의 세그멘트(코드, 힙, 스택)만 존재하기 때문에 지정 가능한 세그멘트 하나는 미사용으로 남는다. 즉, 전체 주소 공간의 1/4은 사용이 불가능하다. 이 문제를 해결하기 위해 일부 시스템은 코드와 힙을 하나의 세그멘트에 저장하고 세그멘트 선택을 위해 1비트만 사용한다 [LL82].

특정 주소의 세그멘트를 하드웨어적으로 파악하는 다른 방법들이 있다. **묵시적(implicit)** 접근 방식에서는 주소가 어떻게 형성되었나를 관찰하여 세그멘트를 결정한다. 예를 들어, 주소가 프로그램 카운터에서 생성되었다면(즉, 명령어 반입) 주소는 코드 세그멘트 내에 있을 것이다. 주소가 스택 또는 베이스 포인터에 기반을 둔다면 주소는 스택 세그멘트 내에 있다. 다른 주소는 모두 힙에 있어야 한다.

19.3 스택

지금까지 주소 공간의 중요한 구성 요소인 스택을 다루지 않았다. 앞의 그림에서 스택은 물리 주소 28 KB에 배치되어 있지만 한 가지 아주 중요한 차이가 있다. 다른 세그멘트들과는 *반대 방향으로 확장된다*는 것이다. 물리 메모리 28 KB에서 시작해서 26 KB까지 차지한다, 가상 주소 16 KB에서 14 KB에 해당한다. 다른 방식의 변환이 필요하다.

첫 번째, 간단한 하드웨어가 추가로 필요하다. 베이스와 바운드 값뿐 아니라 하드웨어는 세그멘트가 어느 방향으로 확장하는지도 알아야 한다. 예를 들어, 하나의 비트를 사용하여 주소가 커지는 쪽(양의 방향)으로 확장하면 1, 작아지는 쪽(음의 방향)으로 확장하면 0으로 설정할 수 있다. 그림 19.4는 이 사실을 반영하여 하드웨어가 관리해야 하는 정보를 표현하고 있다.

하드웨어는 세그멘트가 반대 방향으로 늘어날 수 있다는 것을 알고 있기 때문에, 그런 가상 주소에 대해서는 다른 방식으로 변환한다. 이 과정을 이해하기 위하여 스택에 해당하는 가상 주소를 예로 들어 변환해 보자.

이 예에서 가상 주소 15 KB에 접근하려고 한다고 가정하자. 이 주소는 물리 주소 27 KB에 매핑되어야 한다. 이 가상 주소를 이진 형태로 바꾸면 `11 1100 0000 0000` (16진수 `0x3C00`)이 된다. 하드웨어는 상위 2비트 (11)를 사용하여 세그멘트를 지정한다. 이를 고려하면 3 KB의 오프셋이 남는다. 올바른 음수 오프셋을 얻기 위해서 3 KB

세그멘트	베이스	크기	순방향으로 증가?
코드	32 KB	2 KB	1
힙	34 KB	2 KB	1
스택	28 KB	2 KB	0

〈그림 19.4〉 **세그멘트 레지스터 (반대 방향 확장 지원 기능 포함)**

에서 세그멘트 최대 크기를 빼야 한다. 이 예에서는 세그멘트의 최대 크기가 4 KB이고 따라서 올바른 오프셋은 3 KB에서 4 KB를 뺀 -1 KB이다. 이 음수 오프셋(-1 KB)을 베이스(28 KB)에 더하면 올바른 물리 주소 27 KB를 얻게 된다. 바운드 검사는 음수 오프셋의 절댓값이 세그멘트의 크기보다 작다는 것을 확인하여 계산할 수 있다.

19.4 공유 지원

세그멘테이션 기법이 발전함에 따라 시스템 설계자들은 간단한 하드웨어 지원으로 새로운 종류의 효율성을 성취할 수 있다는 것을 깨달았다. 구체적으로, 메모리를 절약하기 위해 때로는 주소 공간들 간에 특정 메모리 세그멘트를 **공유**하는 것이 유용하다. 특히, **코드 공유**가 일반적이며, 현재 시스템에서도 광범위하게 사용 중이다.

공유를 지원하기 위해, 하드웨어에 **protection bit**의 추가가 필요하다. 세그멘트마다 protection bit를 추가하여 세그멘트를 읽거나 쓸 수 있는지 혹은 세그멘트의 코드를 실행시킬 수 있는지를 나타낸다. 코드 세그멘트를 읽기 전용으로 설정하면 주소 공간의 독립성을 유지하면서도, 여러 프로세스가 주소 공간의 일부를 공유할 수 있다. 각 프로세스는 여전히 자신의 전용 메모리를 사용하고 있다고 생각하지만 운영체제는 이 변경이 불가능하도록 설정된 메모리 영역을 비밀리에 공유시켜 그러한 환상을 유지토록 한다.

하드웨어(및 운영체제)가 유지하는 부가 정보의 예가 그림 19.5에 나와 있다. 코드 세그멘트는 읽기 및 실행으로 설정되어 같은 물리 세그멘트가 여러 가상 주소 공간에 매핑될 수 있다.

세그멘트	베이스	크기	양수 방향으로 증가?	보호
코드	32 KB	2 KB	1	읽기-실행
힙	34 KB	2 KB	1	읽기-쓰기
스택	28 KB	2 KB	0	읽기-쓰기

〈그림 19.5〉 **세그멘트 레지스터 값 (보호 정보 포함)**

protection bit를 사용하면 앞서 언급한 하드웨어 알고리즘이 수정되어야 한다. 가상 주소가 범위 내에 있는지 확인하는 것 이외에 특정 액세스가 허용되는지를 확인해야

한다. 사용자 프로세스가 읽기 전용 페이지에 쓰기를 시도하는 경우 또는 실행 불가 페이지에서 실행하려고 하면 하드웨어는 예외를 발생시켜서 운영체제가 위반 프로세스를 처리할 수 있게 해야 한다.

19.5 소단위 대 대단위 세그멘테이션

우리 예제의 대부분은 지금까지 소수의 세그멘트(즉, 코드, 스택, 힙)만을 지원하는 시스템에만 주로 초점을 맞추었다. 우리는 이 세그멘테이션을 **대단위(coarse-grained)**라고 생각할 수 있다. 주소 공간을 비교적 큰 단위의 공간으로 분할하기 때문이다. 일부 초기 시스템(예를 들어, Multics [CV65; DD68])은 주소 공간을 작은 크기의 공간으로 잘게 나누는 것이 허용되었기 때문에, **소단위(fine-grained)** 세그멘테이션이라고 부른다.

많은 수의 세그멘트를 지원하기 위해서는 여러 세그멘트의 정보를 메모리에 저장할 수 있는 세그멘트 테이블 같은 하드웨어가 필요하다. **세그멘트 테이블**을 이용하면 매우 많은 세그멘트를 손쉽게 생성하고 융통성 있게 세그멘트를 사용할 수 있다. 예를 들어, Burroughs B5000과 같은 초창기 시스템은 수천 개의 세그멘트를 지원하였고, 컴파일러가 코드나 데이터를 여러 세그멘트로 분할할 경우 운영체제와 하드웨어가 이를 지원할 수 있게 하였다 [RK68]. 당시의 생각은 소단위 세그멘트로 관리하는 것이 운영체제가 사용 중인 세그멘트와 미사용인 세그멘트를 구분하여 메인 메모리를 더 효율적으로 활용할 수 있다는 것이었다.

19.6 운영체제의 지원

이제 세그멘트의 기본 동작은 이해하고 있어야 한다! 시스템이 각 주소 공간 구성 요소를 별도로 물리 메모리에 재배치하기 때문에 전체 주소 공간이 하나의 베이스-바운드 쌍을 가지는 간단한 방식에 비해 물리 메모리를 엄청나게 절약할 수 있다. 구체적으로는 스택과 힙 사이의 사용하지 않는 공간에 물리 메모리를 할당할 필요가 없기 때문에 같은 크기의 물리 메모리에 더 많은 주소 공간을 탑재할 수 있다.

세그멘테이션은 새로운 많은 문제를 제기한다. 해결해야 할 새로운 운영체제 문제를 우선 설명한다. 처음 문제는 오래된 문제이다. 문맥 교환 시 운영체제는 어떤 일을 해야 하는가? 아마 이미 상상하고 있으리라 믿는다. 세그멘트 레지스터의 저장과 복원이다. 각 프로세스는 자신의 가상 주소 공간을 가지며, 운영체제는 프로세스가 다시 실행하기 전에 레지스터들을 올바르게 설정해야 한다.

두 번째, 더욱 중요한 문제는 미사용 중인 물리 메모리 공간의 관리이다. 새로운 주소 공간이 생성되면 운영체제는 이 공간의 세그멘트를 위한 비어있는 물리 메모리 영역을 찾을 수 있어야 한다. 이전에 우리는 각 주소 공간의 크기가 동일하다고 가정했다. 물리 메모리는 프로세스가 탑재될 슬롯의 집합이라고 생각될 수 있었다. 지금은 프로세스가 많은 세그멘트를 가질 수 있고, 각 세그멘트는 크기가 다를 수 있다.

일반적으로 생길 수 있는 문제는 물리 메모리가 빠르게 작은 크기의 빈 공간들로 채워진다는 것이다. 이 작은 빈 공간들은 새로이 생겨나는 세그멘트에 할당하기도 힘들

거니와 기존 세그멘트를 확장하는 데에도 도움이 되지 않는다. 우리는 이 문제를 **외부 단편화**(**external fragmentation**)라고 부른다 [Ran69]. 그림 19.6(왼쪽)을 보시오.

〈그림 19.6〉 **압축 전과 압축 후의 메모리 상태**

이 예에서 새로운 프로세스가 생성되어 20 KB를 할당하려고 한다고 가정하자. 위 예에서 24 KB의 빈 공간이 존재하기는 하지만 하나의 연속된 공간이 아니라 세 개의 청크(chunk)로 나누어져 있다. 운영체제는 20 KB의 요청을 충족시킬 수 없다.

이 문제의 해결책 중 한 가지는 기존의 세그멘트를 정리하여 물리 메모리를 **압축**(**compact**)하는 것이다. 예를 들어, 운영체제는 현재 실행 중인 프로세스를 중단하고, 그들의 데이터를 하나의 연속된 공간에 복사하고, 세그멘트 레지스터가 새로운 물리 메모리 위치를 가리키게 하여, 자신이 작업할 큰 빈 공간을 확보할 수 있다. 이렇게 함으로써 운영체제는 새로운 할당 요청을 충족시킬 수 있다. 하지만 세그멘트 복사는 메모리에 부하가 큰 연산이고 일반적으로 상당량의 프로세서 시간을 사용하기 때문에 압축은 비용이 많이 든다. 압축 작업 후의 물리 메모리가 그림 19.6(오른쪽)에 나와 있다.

간단한 방법은 빈 공간 리스트를 관리하는 알고리즘을 사용하는 것이다. 빈 공간 관리 알고리즘은 할당 가능한 메모리 영역들을 리스트 형태로 유지한다. **최적 적합**(**best-fit**), **최악 적합**(**worst-fit**), **최초 적합**(**first-fit**) 및 **버디 알고리즘**(**buddy algorithm**)과 같은 고전적인 알고리즘을 포함하여 말 그대로 수백 개의 방식이 존재한다. 이 중에서 최적 적합은 빈 공간 리스트에서 요청된 크기와 가장 비슷한 크기의 공간을 할당한다 [Knu68]. 이러한 알고리즘에 대해 더 알고 싶다면 Wilson 등의 논문이 좋은 출발점이 될 것이다 [RWi+95]. 아니면 추후 이 책에서 기본적인 내용이 나올 때까지 기다려도 무방하다. 알고리즘이 아무리 정교하게 동작한다고 해도 외부 단편화는 여전히 존재한다. 좋은 알고리즘은 외부 단편화를 가능한 줄이는 것이 목표이다.

팁: 1000개의 해결책이 존재한다면, 최선의 해결책은 존재하지 않는다

외부 단편화를 최소화하기 위한 매우 다양한 알고리즘이 존재한다는 사실은 중요한 의미를 가진다: 외부 단편화를 해결하는 유일한 "최선"책은 존재하지 않는다. 우리는 합리적인 선택을 하고 그 해결책이 충분히 좋기를 희망할 뿐이다. 유일한 진정한 해결책은 가변 길이 할당을 허용하지 않음으로써 아예 문제가 생기지 않게 하는 것이다. 이후의 장에서 이 해결책을 살펴볼 것이다.

19.7 요약

세그멘테이션은 많은 문제를 해결하며, 메모리 가상화를 효과적으로 실현할 수 있다. 단순한 동적 재배치를 넘어 세그멘테이션은 주소 공간 상의 논리 세그멘트 사이의 큰 공간에 대한 낭비를 피함으로써 드문드문 사용되는 주소 공간(sparse address space)을 지원할 수 있다. 세그멘테이션에 필요한 산술 연산은 쉽고 하드웨어 구현에 적합하기 때문에 속도가 빠르다. 변환 오버헤드도 최소이다. 코드 공유의 장점도 부가적으로 발생한다. 코드가 별도의 세그멘트에 존재한다면 그러한 코드는 실행 중인 여러 프로그램 사이에서 공유될 수 있다.

그러나 살펴본 것처럼 세그멘트의 크기가 일정하지 않기 때문에 몇 가지 문제가 발생한다. 논한 바와 같이 첫째는 외부 단편화이다. 세그멘트는 가변 크기이기 때문에 빈 공간들의 크기 역시 모두 다르며 메모리 할당 요청을 충족시키는 것이 어려울 수 있다. 비교적 정교한 알고리즘 [RWi+95]을 사용하거나 주기적으로 메모리를 압축할 수는 있지만, 외부 단편화 문제는 가변 길이 할당의 태생적인 문제이기 때문에 회피하기 어렵다.

두 번째, 더욱 중요한 문제는 세그멘테이션이 아직 일반적인 드문드문 사용되는 주소 공간을 지원할 만큼 충분히 유연하지 못하다는 것이다. 예를 들어, 크기가 크지만 드문드문 사용되는 힙이 하나의 논리적인 세그멘트에 배정되어 있다고 할 때 이 힙에 접근하기 위해서는 힙 전체가 여전히 물리 메모리에 존재해야 한다. 다시 말해서, 주소 공간이 사용되는 모델과 이를 지원하기 위한 세그멘테이션의 설계 방법이 정확히 일치하지 않는다면, 세그멘테이션은 제대로 동작하지 않는다. 우리는 무언가 새로운 해결책을 찾아야 한다. 찾을 준비 되었나?

참고 문헌

[CV65] **"Introduction and Overview of the Multics System"**
F.J. Corbato and V.A. Vyssotsky
Fall Joint Computer Conference, 1965
Fall Joint Computer Conference에서 발표된 Multics에 관한 다섯 논문 중 하나. 그날 그 방의 보이지 않는 관찰자였으면!

[DD68] **"Virtual Memory, Processes, and Sharing in Multics"**
Robert C. Daley and Jack B. Dennis
Communications of the ACM, Volume 11, Issue 5, May 1968
Multics에서 동적 링킹을 수행하는 방법에 관한 초기 논문. 동적 링킹은 시대를 앞선 아이디어였다. 동적 링킹은 20년이 지난 X-windows 라이브러리가 요구하면서 다시 등장하였다. 어떤 이는 이 커다란 X11 라이브러리가 UNIX의 초기 버전에서 동적 링킹 지원을 제거한 것에 대한 MIT의 복수다라고 말한다.

[Gre62] **"Fact Segmentation"**
M.N. Greenfield
Proceedings of the SJCC, Volume 21, May 1962
세그멘테이션에 관한 또 다른 초기 논문. 너무 일찍 출간되었기 때문에 다른 연구에 대한 참고 문헌이 없다.

[Hol61] **"Program Organization and Record Keeping for Dynamic Storage"**
A.W. Holt
Communications of the ACM, Volume 4, Issue 10, October 1961
세그멘테이션과 그의 사용에 관한 믿을 수 없을 정도로 초기 논문이면서 읽기 어려운 논문.

[09] **"Intel 64 and IA-32 Architectures Software Developer's Manuals"**
Intel, 2009
URL: `http://www.intel.com/products/processor/manuals`
세그멘테이션에 관한 부분을 읽으려고 시도해 보라 (볼륨 3a의 3장). 머리를 손상시킬 것이다, 적어도 아주 조금이라도.

[Knu68] **"The Art of Computer Programming: Volume I"**
Donald Knuth
Addison-Wesley, 1968
Knuth는 그의 초창기 책 "The Art of Computer Programming"으로 뿐 아니라 그의 조판 시스템 TeX로 유명하다. TeX는 오늘까지 전문가에 의해 사용되는 강력한 조판 도구이다. 그리고 사실 바로 이 책을 조판한 도구이다. 그의 알고리즘에 관한 시리즈는 현재 컴퓨팅 시스템의 근간을 이루는 많은 알고리즘의 훌륭한 초창기 참고 문헌이다.

[Lam83] **"Hints for Computer Systems Design"**
Butler Lampson
ACM Operating Systems Review, 15:5, October 1983
컴퓨터 시스템의 설계에 관한 Lampson의 그 유명한 조언. 언젠가 한 번은 읽어야 할 논문이자 아마 여러 번 읽어야 할 논문일 것이다.

[LL82] **"Virtual Memory Management in the VAX/VMS Operating System"**
Henry M. Levy and Peter H. Lipman
IEEE Computer, Volume 15, Number 3 (March 1982)
설계에 매우 많은 상식이 담긴 고전적인 메모리 관리 시스템. 나중에 자세히 공부할 것이다.

[RK68] **"Dynamic Storage Allocation Systems"**
B. Randell and C.J. Kuehner
Communications of the ACM Volume 11(5), pages 297-306, May 1968
페이징과 세그멘테이션의 차이에 관한 좋은 개괄. 다양한 컴퓨터에 관한 약간의 역사적 논의도 포함하고 있다.

[Ran69] **"A note on storage fragmentation and program segmentation"**
Brian Randell
Communications of the ACM Volume 12(7), pages 365-372, July 1969
단편화를 논의한 초창기 논문 중의 하나.

[RWi+95] **"Dynamic Storage Allocation: A Survey and Critical Review"**
Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles
In International Workshop on Memory Management Scotland, United Kingdom, September 1995
메모리 할당기에 관한 훌륭한 요약 논문.

[Smo04] **"System Call Support"**
Mark Smotherman
URL: `http://people.cs.clemson.edu/%CB%9Cmark/syscall.html`
시스템 콜 지원에 관한 깔끔한 역사. Smotherman은 인터럽트와 컴퓨팅 역사의 다른 재미있는 측면과 같은 주제에 관한 초기 역사를 수집하였다. 자세한 사항에 관해서는 웹 페이지를 참조하라.

숙제

이 프로그램은 세그멘테이션 시스템에서 주소 변환이 어떤 식으로 일어나는지 보여준다. 자세한 내용은 README를 참조하시오.

문제

1. 우선 몇 주소를 변환하기 위해 작은 주소 공간을 사용하여 보자. 여기에 여러 가지 랜덤 시드와 함께 간단한 매개변수의 집합이 있다. 이때 주소를 변환할 수 있을까?

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2. 이제 우리가 구축한 이 작은 주소 공간을 이해하고 있는지 알아보자(위의 질문에서 사용한 매개변수를 사용하여). 세그멘트 0의 합법적인 가상 주소의 최댓값은 무엇인가? 세그멘트 1의 합법적인 가상 주소의 최저값은 무엇인가? 이 전체 주소 공간의 잘못된 주소 중 가장 큰 주소와 가장 작은 주소는 무엇인가? 마지막으로, 플래그 `-A`와 함께 실행한 `segmentation.py`의 정확성을 어떻게 확인할 수 있을까?

3. 16바이트 주소 공간과 128바이트의 물리 메모리가 있다고 하자. 시뮬레이터가 지정된 주소 스트림에 대해 다음과 같은 변환 결과를 생성하기 위해서는 베이스와 바운드를 어떤 값으로 설정해야 하는가: 유효, 유효, 위반, ..., 위반, 유효, 유효? 다음과 같은 매개변수를 가정한다.

```
segmentation.py -a 16 -p 128
        -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
        --b0 ? --l0 ? --b1 ? --l1 ?
```

4. 무작위로 생성된 가상 주소 중 약 90%가 유효한 시스템에서(즉, 세그멘테이션 위반이 아닌) 문제를 만들고 싶다고 가정하자. 그런 결과를 내려면 시뮬레이터를 어떻게 설정해야 하는가? 어떤 매개변수가 중요한가?

5. 모든 가상 주소가 유효하지 않도록 시뮬레이터를 실행할 수 있는가? 어떻게?