16

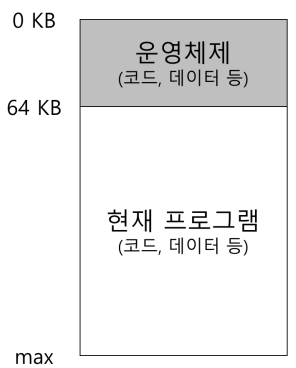
주소 공간의 개념

초기에는 컴퓨터 시스템을 구현하는 것이 쉬웠다. 왜냐고? 사용자가 많은 것을 기대하지 않았기 때문이다. 이 모든 두통거리를 만든 사람들은 바로 "사용의 편이", "고성능", "신뢰성" 등을 기대한 훌륭한 사용자이다. 다음에 그런 컴퓨터 사용자를 만나게 되면 그들이 제시한 모든 문제에 대해 감사하기 바란다.

16.1 초기 시스템

메모리 관점에서 초기 컴퓨터는 많은 개념을 사용자에게 제공하지 않았다. 컴퓨터의 물리 메모리는 그림 16.1에서 보는 것과 같이 생겼다.

운영체제는 메모리에(이 예에서는 물리 주소 0부터) 상주하는 루틴(실제로는 라이브러리)의 집합이었다. 물리 메모리에 하나의 실행 중인 프로그램(프로세스)이 존재하였고(이 예에서 물리 주소 64KB부터 시작하여) 나머지 메모리를 사용하였다. 특별한 가상화는 거의 존재하지 않았고 사용자는 운영체제로부터 그리 많은 것을 기대하지 않았다. 그 당시 운영체제 개발자의 삶은 쉬운 게 확실했다. 그렇지 않나?



〈그림 16.1〉 **운영체제 : 초기**

16.2 멀티프로그래밍과 시분할

시간이 흐른 후, 컴퓨터는 고가 장비였기 때문에, 사람들이 더 효과적으로 컴퓨터를 공유하기 시작했다. **멀티프로그래밍**(**multi-programming**) 시대가 도래하였다 [DH66]. 여러 프로세스가 실행 준비 상태에 있고 운영체제는 그들을 전환하면서 실행하였다. 예를 들어 한 프로세스가 입출력을 실행하면, CPU는 다른 프로세스로 전환하였다. 이런 전환은 CPU의 **이용률**을 증가시켰다. 당시에는 특히 이러한 **효율성**의 개선이 중요하였다. 시스템의 가격이 수십만 또는 수백만 달러나 하였다(그리고 당신은 Mac이 비싸다고 생각하고 있다!).

곧 사람들이 컴퓨터를 더 많이 사용하길 원하기 시작했으며 **시분할**(**time-sharing**) 시대가 시작되었다 [Str59; Lic60; McC62; McC83]. 많은 사람들이 일괄처리방식(batch computing) 컴퓨팅의 한계를 인식하였으며, 특히 오랜 시간이 걸리는 (따라서 비효과적인) 프로그램-디버그 사이클에 지친 프로그래머 자신들 [CV65]이 일괄처리방식 컴퓨팅의 한계를 느꼈다. 많은 사용자가 동시에 컴퓨터를 사용하고, 현재 실행 중인 작업으로부터 즉시 응답을 원하기 때문에 **대화식 이용**(**interactivity**)의 개념이 중요하게 되었다.

시분할을 구현하는 한 가지 방법은 하나의 프로세스를 짧은 시간 동안 실행시키는 것이다. 해당 기간 동안 프로세스에게 모든 메모리를 접근할 권한이 주어진다(그림 16.1). 그런 후에, 이 프로세스를 중단하고, 중단 시점의 모든 상태를 디스크 종류의 장치(모든 물리 메모리를 포함하여)에 저장하고 다른 프로세스의 상태를 탑재하여 또 짧은 시간 동안 실행시킨다. 시분할 시스템을 이런 식으로 엉성하게 구현하였다 [McC+63].

이 방법에는 커다란 문제가 있다. 너무 느리게 동작한다는 것이고, 특히 메모리가 커질수록 느리게 된다. 레지스터 상태를 저장하고 복원하는 것은 빠르지만 메모리의 내용 전체를 디스크에 저장하는 것은 엄청나게 느리다. 우리가 할 일은 프로세스 전환 시 프로세스를 메모리에 그대로 유지하면서, 운영체제가 시분할 시스템을 효율적으로 구현할 수 있게 하는 것이다(그림 16.2)

〈그림 16.2〉 **세 개의 프로세스 : 공유 메모리**

그림에 세 개의 프로세스(A, B, C)가 있고 각 프로세스는 512 KB 물리 메모리에서 각기 작은 부분을 할당받았다. 하나의 CPU를 가정할 때, 운영체제는 실행할 한 개의 프로세스(A라 하자)를 선택하고, 다른 프로세스들(B와 C)은 준비 큐에서 실행을 기다린다.

시분할 시스템이 대중화되면서 운영체제에게 새로운 요구 사항이 부과되었다. 여러 프로그램이 메모리에 동시에 존재하려면 **보호**(**protection**)가 중요한 문제가 된다. 우리는 한 프로세스가 다른 프로세스의 메모리를 읽거나 혹은 더 안 좋게는 쓸 수 있는 상황을 원하지 않는다.

16.3 주소 공간

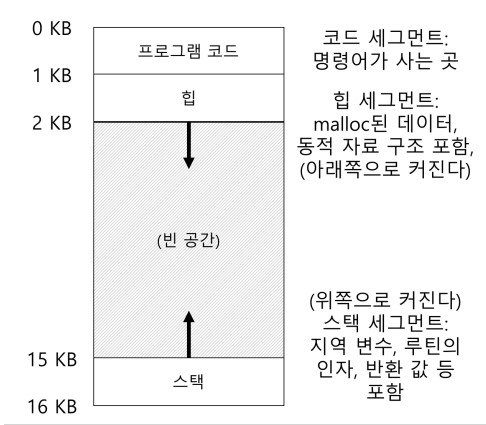
우리는 그런 위험한 행위를 하는 사용자를 염두에 두어야 한다. 그런 위험에 대비하여 운영체제는 **사용하기 쉬운**(**easy to use**) 메모리 개념을 만들어야 한다. 이 개념이 **주소 공간**(**address space**)이다. 실행 중인 프로그램이 가정하는 메모리의 모습이다. 운영체제의 메모리 개념을 이해하는 것이 메모리를 어떻게 가상화할지를 이해하는 핵심이다.

주소 공간은 실행 프로그램의 모든 메모리 상태를 갖고 있다. 예를 들어, 프로그램의 **코드**(**code**, 명령어)는 반드시 메모리에 존재해야 하고 따라서 주소 공간에 존재한다. **스택**은 함수 호출 체인 상의 현재 위치, 지역 변수, 함수 인자와 반환 값 등을 저장하는 데 사용된다. 마지막으로 **힙**(**heap**)은 동적으로 할당되는 메모리를 위해 사용된다. C 언어의 `malloc()`, C++나 Java 같은 객체 지향 언어의 `new`를 통해 메모리를 동적으로 할당받는다. 주소 공간 구성 요소에는 정적으로 초기화된 변수 등의 다른 것들도 있지만, 현재로선 코드, 스택 및 힙의 세 가지만 있다고 가정하자.

그림 16.3은 아주 작은 주소 공간을 보이고 있다(겨우 16 KB)[1]. 프로그램 코드는 주소 공간의 위쪽에 위치한다. 이 예제에서는 주소 0부터 시작하고, 주소 공간의 첫 1 KB를 차지한다. 코드는 정적이기 때문에 메모리에 저장하기 쉽다. 따라서 주소 공간의 상단에 배치하고, 프로그램이 실행되면서 추가 메모리를 필요로 하지 않는다.

다음으로 프로그램 실행과 더불어 확장되거나 축소될 수 있는 두 종류의 주소 공간이 존재한다. 주소 공간의 상단에 존재하는 힙과 하단에 존재하는 스택이다. 두 메모리 영역은 확장할 수 있어야 하기 때문에 이런 방식으로 배치하고, 주소 공간의 양 끝단에 배치해야 두 영역 모두 확장하는 것이 가능하다. 두 영역은 확장 방향이 반대 방향일 수밖에 없다. 힙은 코드 바로 뒤 1 KB부터 시작하고 아래 방향으로 확장한다(예를 들면, 사용자가 `malloc()` 함수를 통해 더 많은 메모리를 요청할 때). 스택은 16 KB에서 시작하고 위쪽 방향으로 확장한다(예를 들어, 사용자가 프로시저를 호출할 때). 그러나 스택과 힙의 이러한 배치는 관례일 뿐이다. 원한다면 주소 공간을 다른 방식으로 배치할 수 있다. 나중에 보겠지만 주소 공간에 여러 **쓰레드**가 공존할 때는 이런 식으로 주소 공간을 나누게 되면 동작하지 않는다.

1) 우리는 종종 이런 작은 크기의 예제를 사용할 것이다. 왜냐하면 (a) 32비트 주소 공간을 표현하는 것은 고통이고 (b) 관련 수학은 더 어렵기 때문이다. 우리는 단순한 수학을 좋아한다.



〈그림 16.3〉 **주소 공간의 예**

주소 공간을 설명할 때, 운영체제가 실행 중인 프로그램에게 제공하는 **개념**(**abstraction**)을 설명한다. 실제로 프로그램이 물리 주소 0에서 16 KB 사이에 존재하는 것은 아니다. 실제로는 임의의 물리 주소에 탑재된다. 그림 16.2의 프로세스 A, B, C를 보자. 각 프로세스가 어떤 식으로 다른 주소에 탑재되었는지 알 수 있다. 우리의 문제는 다음과 같다.

핵심 질문: 메모리를 어떻게 가상화하는가
운영체제는 물리 메모리를 공유하는 다수의 프로세스에게 어떻게 프로세스 전용의 커다란 주소 공간이라는 개념을 제공할 수 있는가?

운영체제가 이 일을 할 때, 우리는 운영체제가 **메모리를 가상화**(**virtualizing memory**)한다고 말한다. 왜냐하면 실행 중인 프로그램은 자신이 특정 주소의 메모리에(예를 들어 0) 탑재되고 매우 큰 주소 공간을(예를 들어, 32비트 또는 64비트) 가지고 있다고 생각하기 때문이다. 현실은 다르다.

예를 들어, 그림 16.2의 프로세스 A가 주소 0으로부터(우리는 이를 **가상 주소**(**virtual address**)라고 부를 것이다) `load` 연산을 수행할 때, 운영체제는 하드웨어의 지원을 통해 물리 주소 0이 아니라 물리 주소 320 KB(A가 탑재된 메모리)를 읽도록 보장해야 한다. 이것이 메모리 가상화의 열쇠이고, 현대 모든 컴퓨터 시스템의 기저를 이룬다.

16.4 목표

이제 메모리 가상화라는 운영체제 기능을 논의할 시점이 되었다. 운영체제는 메모리를 가상화할 뿐 아니라 그 가상화를 멋진 방식으로 한다. 운영체제가 가상화를 멋지게 하기 위해서는, 몇 가지 목표가 필요하다. 이전에 목표들을 보았지만, 다시 보도록 하자. 충분히 반복할 가치가 있다.

팁: 고립의 원칙

고립은 신뢰할 수 있는 시스템을 구축하는 데 중요한 원칙이다. 두 개체가 서로 적절하게 고립된 경우, 한 개체가 실패하더라도 상대 개체에 아무 영향을 주지 않는다는 것을 암시한다. 운영체제는 프로세스를 서로 고립시키기 위해 노력하고 이런 방식으로 한 프로세스가 다른 프로세스에게 피해를 주는 것을 방지한다. 더 나아가 메모리 고립을 사용하여 운영체제는 프로그램이 운영체제 동작에 영향을 줄 수 없다는 것을 보장한다. 일부 현대 운영체제는 고립을 더 확장하여 운영체제의 구성 요소를 서로 고립시킨다. 이러한 **마이크로커널** [Han70; Ras+89; SBL03]은 전통적인 모놀리식 커널보다 더 큰 신뢰성을 제공할 수 있다.

가상 메모리 시스템(VM)의 주요 목표 중 하나는 **투명성**(**transparency**)이다[2]. 운영체제는 실행 중인 프로그램이 가상 메모리의 존재를 인지하지 못하도록 가상 메모리 시스템을 구현해야 한다. 프로그램은 메모리가 가상화되었다는 사실을 인지해서는 안 된다. 오히려 프로그램은 자신이 전용 물리 메모리를 소유한 것처럼 행동해야 한다. 많은 작업들이 메모리를 공유할 수 있도록, 무대 뒤에서 운영체제와 하드웨어가 모든 작업을 수행한다.

VM의 또 다른 목표는 **효율성**(**efficiency**)이다. 운영체제는 가상화가 시간과 공간 측면에서 효율적이도록 해야 한다. 시간적으로는 프로그램이 너무 느리게 실행되서는 안 되고 공간적으로는 가상화를 지원하기 위한 구조를 위해 너무 많은 메모리를 사용해서는 안 된다. 시간-효율적인 가상화를 구현할 때, 운영체제는 TLB 등의 하드웨어 기능을 포함하여 하드웨어의 지원을 받아야 한다. TLB에 대해서는 나중에 적절한 때에 배우게 될 것이다.

마지막으로 VM의 세 번째 목표는 **보호**(**protection**)이다. 운영체제는 프로세스를 다른 프로세스로부터 보호해야 하고 운영체제 자신도 프로세스로부터 보호해야 한다. 프로세스가 탑재, 저장, 혹은 명령어 반입 등을 실행할 때 어떤 방법으로든 다른 프로세스나 운영체제의 메모리 내용에 접근하거나 영향을 줄 수 있어서는 안 된다. 즉, 자신의 주소 공간 밖의 어느 것도 접근할 수 있어서는 안 된다. 보호 성질을 이용하여 우리는 프로세스들을 서로 **고립**(**isolate**)시킬 수 있다. 각 프로세스는 잘못된 혹은 악성 프로세스로부터 안전한 자신만의 보호막 안에서 실행되어야 한다.

다음 장에서 운영체제와 하드웨어 지원을 포함하여 메모리를 가상화하기 위해 필요한 기본적인 기법을 집중적으로 탐구한다. 또한, 빈 공간을 관리하는 방법과 공간이 부족할 때 어떤 페이지를 내보낼 것인가 등의 운영체제 정책들에 대해서도 학습한다. 그런 탐구와 조사를 수행하면서 현대 가상 메모리 시스템의 실제 동작에 대해 이해하게 될

2) 투명성의 이런 사용은 때때로 혼란스럽다. 일부 학생들은 "투명하다"가 모든 것을 숨김 없이 밝혀야 한다는 의미라고, 즉 정부가 그래야 한다는 것처럼 생각한다. 여기서는 그 반대를 의미한다. 운영체제에 의해 제공된 환상은 응용 프로그램에게 보여서는 안 된다. 일반적인 사용에서 투명한 시스템은 인지하기 어려운 시스템이지, Freedom of Information Act에 의해 명기된 것처럼 요청에 반응하는 시스템이 아니다.

여담: 당신이 보는 모든 주소는 가상 주소이다

포인터를 출력하는 C 프로그램을 작성한 적이 있는가? 당신이 보는 값은(좀 큰 변수, 종종 16진수로 출력되는) **가상 주소**(**virtual address**)이다. 프로그램 코드가 탑재된 주소가 어디인지 궁금한 적은 없는가? 그 주소를 출력할 수 있다. 그러나 주소를 출력했다면, 그 주소는 가상 주소이다. 사용자 프로그램(user level program)이 볼 수 있는 주소는 모두 가상 주소이다. 메모리 가상화의 기술 때문에, 명령어와 데이터가 탑재되어 있는 물리 메모리 주소를 알 수 있는 것은 오직 운영체제뿐이다. 결코 잊지 말아야 할 사실: 프로그램에서 주소를 출력하면, 그 주소는 가상 주소, 즉 메모리 배치에 대한 환상이다. 오직 운영체제와 하드웨어만이 진실을 알고 있다.

다음은 `main()` 함수, `malloc()`으로 할당된 메모리, 스택에 존재하는 정수의 위치를 출력하는 프로그램이다.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     printf("location of code : %p\n", (void *) main);
5     printf("location of heap : %p\n", (void *) malloc(1));
6     int x = 3;
7     printf("location of stack : %p\n", (void *) &x);
8     return x;
9 }
```

64비트 Mac OS X 컴퓨터에서 실행하면 우리는 다음과 같은 출력을 얻을 수 있다.

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack : 0x7fff691aea64
```

이 예에서 주소 공간에서 코드가 제일 먼저 등장하고, 다음에 힙이 배치되고, 마지막으로 대규모 가상 공간의 반대편에 스택이 위치한다는 것을 알 수 있다. 이 모든 주소는 가상 주소이며, 저장된 실제 물리 메모리 위치에서 값을 반입하기 위해 운영체제와 하드웨어에 의해 물리 주소로 변환된다.

것이다[3].

16.5 요약

우리는 가상 메모리라는 운영체제의 구성 요소에 대해 소개를 마쳤다. VM 시스템은 프로세스 전용 공간이라는 환상을 프로그램에게 제공할 책임이 있다. 이 공간에 프로그램 명령어 전부와 데이터 전부가 저장된다. 하드웨어의 도움을 받아 운영체제는 가상 메모리 주소를 받아 물리 주소로 변환한다. 물리 주소는 원하는 정보를 반입하기 위하여 물리 메모리에게 전달된다. 운영체제는 많은 프로세스를 대상으로 이러한 작업을 수행하여

3) 그렇지 않다면 수강취소를 해야겠다는 확신을 주게 될 것이다. 그러나 잠깐, 당신이 VM을 잘 이해할 수 있다면 끝까지 내용을 다 소화할 가능성이 높다고 생각한다.

프로그램과 운영체제를 보호한다. 전체적인 접근 방식은 많은 기법들, 많은 저수준 하드웨어 기능과 핵심적인 정책들을 필요로 한다. 우리는 기초적인 필수 "기법"들에 대한 설명을 시작으로 점차 고수준의 정책을 설명할 것이다.

참고 문헌

[CV65] **"Introduction and Overview of the Multics System"**
F.J. Corbato and V.A. Vyssotsky
Fall Joint Computer Conference, 1965
훌륭한 초창기 Multics 논문. 시분할에 대한 좋은 인용이 여기 있다: "시분할에 대한 추진력은 배치 처리 기관에서 프로그램을 디버깅 할 때 경험한 끊임없는 좌절감 때문에 전문 프로그래머들로부터 처음 발생했다. 따라서 시분할의 원래 목표는 컴퓨터를 시분할 하여 많은 사람들이 동시에 사용할 수 있게 하면서 사용자에게는 자신이 컴퓨터 전부를 마음대로 할 수 있다는 환상을 주는 것이다."

[DH66] **"Programming Semantics for Multiprogrammed Computations"**
Jack B. Dennis and Earl C. Van Horn
Communications of the ACM, Volume 9, Number 3, March 1966
멀티프로그래밍에 관한 초기 논문(첫 번째 논문은 아님).

[Han70] **"The Nucleus of a Multiprogramming System"**
Per Brinch Hansen
Communications of the ACM, 13:4, April 1970
운영체제 또는 커널은 맞춤형 운영체제를 구현하기 위한 최소 기능과 융통성을 가지는 층이어야 한다는 첫 번째 논문. 이 주제는 운영체제 연구 역사상 전반에 걸쳐 재논의 된다.

[Lic60] **"Man-Computer Symbiosis"**
J.C.R. Licklider
IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960
컴퓨터와 사람이 어떻게 공존할 수 있는지에 관한 고풍스러운 논문. 시대를 앞서 나갔지만 재미있게 읽을 수 있다.

[McC62] **"Time-Sharing Computer Systems"**
J. McCarthy
Management and the Computer of the Future, MIT Press, Cambridge, Mass, 1962
아마도 McCarthy의 시분할에 관한 기록된 초기 논문. 그러나 다른 논문[McC83]에서 1957년부터 이 아이디어를 구상 중이었다고 말함. McCarthy는 시스템 분야를 떠나 Stanford에서 LISP 프로그래밍 언어를 만드는 등 인공 지능 분야의 거인이 되었다. 자세한 사항은 McCarthy의 홈페이지를 보시오: `http://www-formal.stanford.edu/jmc/`

[McC+63] **"A Time-Sharing Debugging System for a Small Computer"**
J. McCarthy, S. Boilen, E. Fredkin, and J.C.R. Licklider
AFIPS '63 (Spring), May, 1963, New York, USA
프로그램이 실행되지 않을 때 프로그램 메모리를 "drum"에 스왑-아웃하고 다시 실행되어야 할 때 "core" 메모리에 다시 탑재되는 시스템의 초기 훌륭한 예.

[McC83] **"Reminiscences on the History of Time Sharing"**
John McCarthy
Winter or Spring of 1983
URL: `http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html`
시분할 아이디어가 어디서 비롯되었는지에 관한 멋진 역사적인 노트. Strachey의 연구[Str59]를 이 분야의 선구적 연구라고 인용한 사람들에 대한 의문 제기도 포함하고 있음.

[Ras+89] **"Mach: A System Software kernel"**

Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael Jones
COMPCON 89, February 1989
마이크로커널에 관한 첫 번째 프로젝트는 아닐지라도 CMU의 Mach 프로젝트는 유명하고 영향을 크게 미쳤다. 이 운영체제는 Mac OS X의 깊숙한 곳에 아직도 현존하고 있다.

[Str59] **"Time Sharing in Large Fast Computers"**
C. Strachey
Proceedings of the International Conference on Information Processing, UNESCO, June 1959
시분할에 관한 초기 참고 문헌의 하나.

[SBL03] **"Improving the Reliability of Commodity Operating Systems"**
Michael M. Swift, Brian N. Bershad, and Henry M. Levy
SOSP 2003
마이크로커널 사고 방식이 운영체제 신뢰성을 얼마나 증대시킬 수 있는지를 보여 주는 첫 번째 논문.