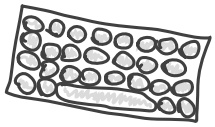
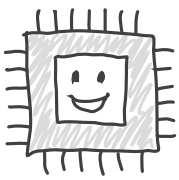


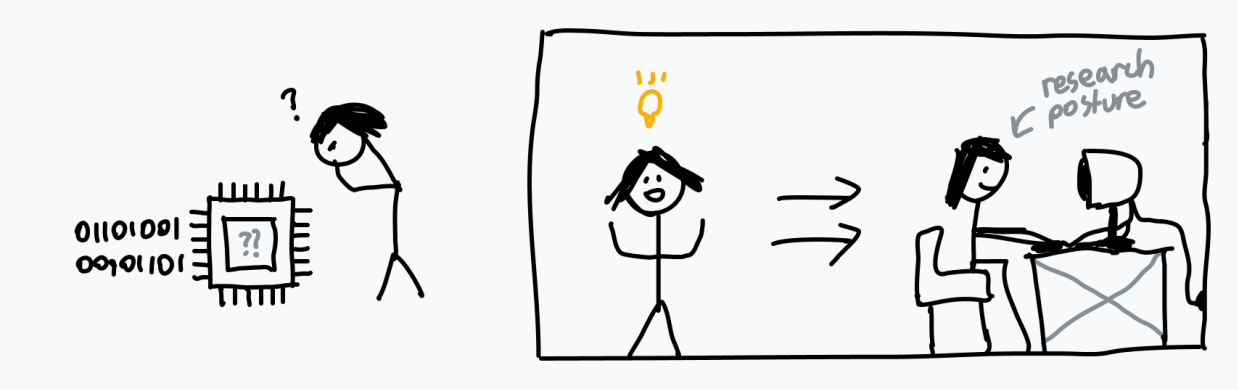


Putting the “You” in CPU

작성자: Lexi Mattick & Hack Club · 2023년 7월

챕터 0: 소개

저는 컴퓨터로 많은 일들을 해왔지만, 항상 지식에 공백이 있었습니다. 컴퓨터에서 프로그램을 실행하면 정확히 무슨 일이 일어나는 걸까요? 이 공백에 대해 생각해봤습니다 — 저는 대부분의 필수적인 저수준 지식을 가지고 있었지만, 모든 것을 하나로 연결하는 데 어려움을 겪고 있었습니다. 프로그램이 정말로 CPU에서 직접 실행되는 걸까요, 아니면 다른 무언가가 진행되는 걸까요? 저는 시스템 콜(syscall)을 사용해봤지만, 그것들이 어떻게 *작동* 할까요? 정말로 무엇일까요? 어떻게 여러 프로그램이 동시에 실행될까요?



참다 못해 가능한 한 많은 것을 알아내기 시작했습니다. 대학에 가지 않는다면 포괄적인 시스템 리소스가 많지 않기 때문에, 품질이 다양하고 때로는 상충되는 정보를 가진 수많은 출처를 뒤져야 했습니다. 몇 주간의 연구와 거의 40페이지에 달하는 노트 후에, 저는 컴퓨터가 시작부터 프로그램 실행까지 어떻게 작동하는지 훨씬 더 잘 이해하게 되었다고 생각합니다. 제가 배운 것을 설명하는 확실한 글 하나가 있었다면 목숨이라도 걸었을 것이기에, 제가 원했던 그 글을 쓰고 있습니다.

그리고 사람들이 뭐라고 하죠... 다른 사람에게 설명할 수 있을 때만 진정으로 이해한다고.

> 바쁘신가요? 이미 이런 걸 안다고 생각하시나요?
>
> **[3장을 읽어보세요]** 그러면 새로운 것을 배우실 거라 장담합니다. 당신이 리누스 토르발스 본인이 아닌 한 말이죠.

챕터 1: 기초

이 글을 쓰면서 계속해서 저를 놀라게 한 한 가지는 컴퓨터가 얼마나 *단순한지*였습니다. 실제보다 더 복잡하거나 추상적일 것이라고 기대하며 스스로를 긴장시키지 않는 것이 여전히 어렵습니다! 계속 진행하기 전에 뇌에 새겨야 할 한 가지가 있다면, 단순해 보이는 모든 것이 실제로 그만큼 단순하다는 것입니다. 이 단순함은 매우 아름답고 때로는 매우, 매우 저주받았습니다.

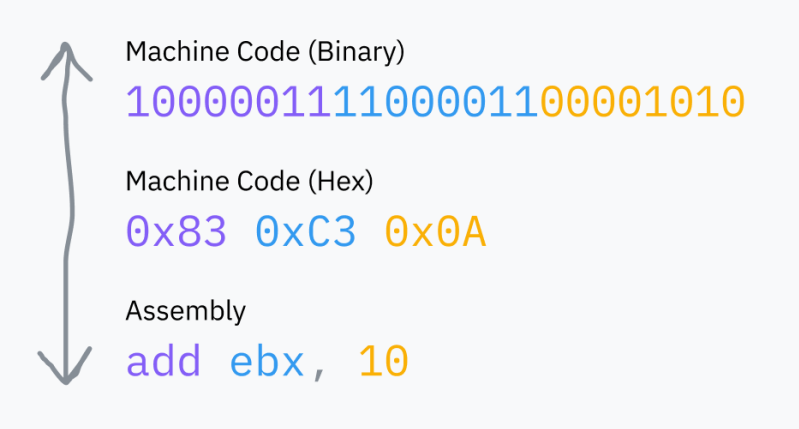
컴퓨터가 핵심에서 어떻게 작동하는지에 대한 기초부터 시작하겠습니다.

컴퓨터의 구조

컴퓨터의 *중앙 처리 장치*(CPU)는 모든 연산을 담당합니다. 최고 책임자입니다. 요술봉입니다. 컴퓨터를 시작하자마자 작동하기 시작하여 명령어를 연속적으로 실행합니다.

최초로 대량 생산된 CPU는 1960년대 후반 이탈리아 물리학자이자 엔지니어인 페데리코 파긴(Federico Faggin)이 설계한 Intel 4004였습니다. 오늘날 사용하는 64비트 시스템 대신 4비트 아키텍처였으며, 현대 프로세서보다 훨씬 덜 복잡했지만, 그 단순함의 많은 부분이 여전히 남아있습니다.

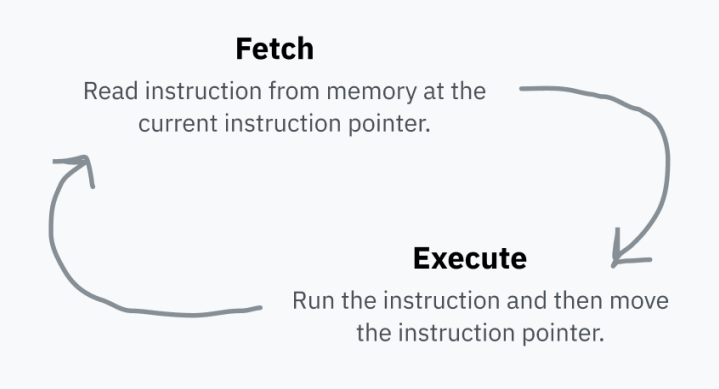
CPU가 실행하는 "명령어"는 단지 이진 데이터입니다: 실행되는 명령어를 나타내는 1~2바이트(opcode)와 그 뒤에 명령어를 실행하는 데 필요한 모든 데이터가 따릅니다. 우리가 *기계어(machine code)*라고 부르는 것은 이러한 이진 명령어들의 연속일 뿐입니다. 어셈블리는 원시 비트보다 인간이 읽고 쓰기 쉬운 기계어를 읽고 쓰는 데 유용한 구문입니다. 항상 CPU가 읽을 수 있는 이진수로 컴파일됩니다.



> 참고: 명령어가 위의 예제처럼 항상 기계어에서 1:1로 표현되는 것은 아닙니다. 예를 들어, `add eax, 512`는 `05 00 02 00 00`으로 변환됩니다.
>
> 첫 번째 바이트(`05`)는 특별히 *EAX 레지스터에 32비트 숫자 더하기*를 나타내는 opcode입니다. 나머지 바이트는 리틀 엔디안 바이트 순서로 512(`0x200`)입니다.
>
> Defuse Security는 어셈블리와 기계어 간의 변환을 실험해볼 수 있는 유용한 도구를 만들었습니다.

RAM은 컴퓨터의 주 메모리 뱅크로, 컴퓨터에서 실행 중인 프로그램이 사용하는 모든 데이터를 저장하는 대용량 다목적 공간입니다. 여기에는 프로그램 코드 자체와 운영 체제 핵심의 코드가 포함됩니다. CPU는 항상 RAM에서 직접 기계어를 읽으며, RAM에 로드되지 않은 코드는 실행할 수 없습니다.

CPU는 다음에 가져올 명령어가 있는 RAM의 위치를 가리키는 *명령어 포인터(instruction pointer)*를 저장합니다. 각 명령어를 실행한 후, CPU는 포인터를 이동하고 반복합니다. 이것이 *페치-실행 사이클(fetch-execute cycle)*입니다.



명령어를 실행한 후 포인터는 RAM에서 명령어 바로 다음으로 이동하여 이제 다음 명령어를 가리킵니다. 그래서 코드가 실행됩니다! 명령어 포인터는 계속 앞으로 나아가며 메모리에 저장된 순서대로 기계어를 실행합니다. 일부 명령어는 명령어 포인터에게 다른 곳으로 점프하거나 특정 조건에 따라 다른 곳으로 점프하도록 지시할 수 있습니다. 이것이 재사용 가능한 코드와 조건부 로직을 가능하게 합니다.

이 명령어 포인터는 *레지스터(register)*에 저장됩니다. 레지스터는 CPU가 읽고 쓰기에 매우 빠른 작은 저장 공간입니다. 각 CPU 아키텍처에는 고정된 레지스터 집합이 있으며, 계산 중 임시 값 저장부터 프로세서 구성까지 모든 용도로 사용됩니다.

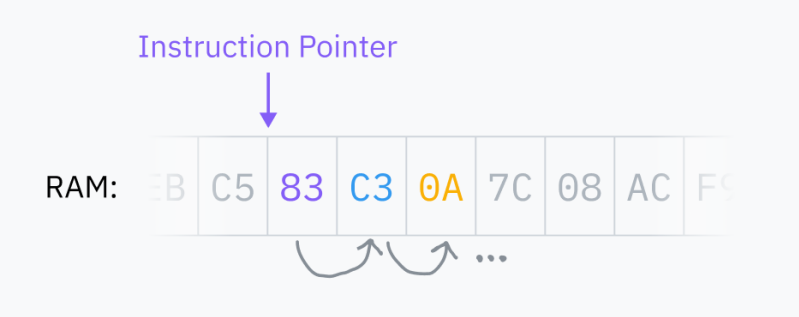
일부 레지스터는 기계어에서 직접 액세스할 수 있습니다(이전 다이어그램의 `ebx`처럼).

다른 레지스터는 CPU 내부에서만 사용되지만 특수 명령어를 사용하여 업데이트하거나 읽을 수 있는 경우가 많습니다. 한 예는 명령어 포인터로, 직접 읽을 수는 없지만 점프 명령어로 업데이트할 수 있습니다.

프로세서는 순진합니다

원래 질문으로 돌아가 봅시다: 컴퓨터에서 실행 가능한 프로그램을 실행하면 무슨 일이 일어날까요? 먼저, 실행 준비를 위해 많은 마법이 일어납니다 — 이 모든 것은 나중에 설명하겠습니다 — 하지만 프로세스가 끝나면 어딘가 파일에 기계어가 있습니다. 운영 체제는 이것을 RAM에 로드하고 CPU에게 명령어 포인터를 RAM의 해당 위치로 점프하도록 지시합니다. CPU는 평소처럼 페치-실행 사이클을 계속 실행하므로 프로그램이 실행되기 시작합니다!

(이것은 저에게 스스로를 긴장시키는 순간 중 하나였습니다 — 정말로, 이것이 당신이 이 글을 읽는 데 사용하는 프로그램이 실행되는 방식입니다! CPU는 브라우저의 명령어를 RAM에서 순차적으로 가져와 직접 실행하고, 이 글을 렌더링하고 있습니다.)



CPU는 매우 기본적인 세계관을 가지고 있습니다. 현재 명령어 포인터와 약간의 내부 상태만 봅니다. 프로세스는 전적으로 운영 체제 추상화이며, CPU가 기본적으로 이해하거나 추적하는 것이 아닙니다.

**손을 휘두르며* 프로세스는 더 많은 컴퓨터를 팔기 위해 ~~os 개발자들~~ 빅 바이트가 만들어낸 추상화입니다*

저에게 이것은 답하는 것보다 더 많은 질문을 불러일으킵니다:

1. CPU가 멀티프로세싱에 대해 알지 못하고 명령어를 순차적으로 실행하기만 한다면, 실행 중인 프로그램 안에 갇히지 않는 이유는 무엇일까요? 어떻게 여러 프로그램이 동시에 실행될 수 있을까요?
2. 프로그램이 CPU에서 직접 실행되고 CPU가 RAM에 직접 액세스할 수 있다면, 왜 코드가 다른 프로세스의 메모리나, 더 나쁘게는 커널에 액세스할 수 없을까요?
3. 그리고 말이 나와서 말인데, 모든 프로세스가 모든 명령어를 실행하고 컴퓨터에 무엇이든 하는 것을 방지하는 메커니즘은 무엇일까요? 그리고 대체 시스템 콜이 뭡니까?

메모리에 대한 질문은 별도의 섹션이 필요하며 **[5장]**에서 다룹니다 — 요약하자면 대부분의 메모리 액세스는 실제로 전체 주소 공간을 다시 매핑하는 간접 계층을 거칩니다. 지금은 프로그램이 모든 RAM에 직접 액세스할 수 있고 컴퓨터가 한 번에 하나의 프로세스만 실행할 수 있다고 가정하겠습니다. 이 두 가지 가정은 시간이 지나면 설명할 것입니다.

이제 시스템 콜과 보안 링으로 가득 찬 땅으로의 첫 번째 토끼굴을 뛰어넘을 시간입니다.

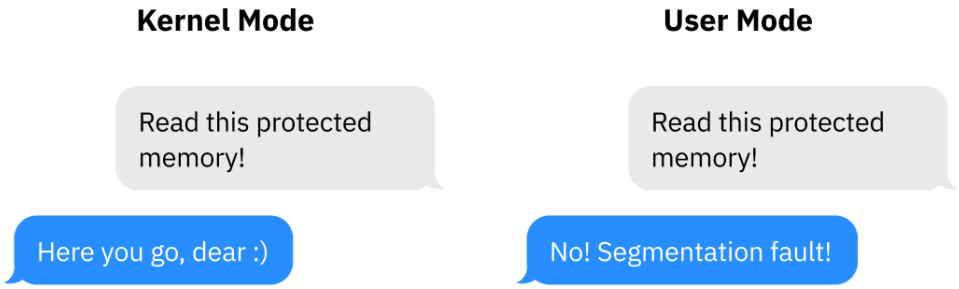
> **참고: 그런데 커널이 뭐죠?**
>
> macOS, Windows 또는 Linux와 같은 컴퓨터의 운영 체제는 컴퓨터에서 실행되어 모든 기본 작업을 수행하는 소프트웨어 모음입니다. “기본 작업”은 매우 일반적인 용어이며 “운영 체제”도 마찬가지입니다 — 누구에게 묻느냐에 따라 기본적으로 컴퓨터와 함께 제공되는 앱, 글꼴 및 아이콘과 같은 것들을 포함할 수 있습니다.
>
> 그러나 커널은 운영 체제의 핵심입니다. 컴퓨터를 부팅하면 명령어 포인터는 어딘가의 프로그램에서 시작합니다. 그 프로그램이 커널입니다. 커널은 컴퓨터의 메모리, 주변 장치 및 기타 리소스에 대한 거의 완전한 액세스 권한을 가지며, 컴퓨터에 설치된 소프트웨어(사용자 공간 프로그램)를 실행하는 책임이 있습니다. 커널이 이 액세스 권한을 어떻게 가지는지 — 그리고 사용자 공간 프로그램이 어떻게 가지지 못하는지는 이 글의 과정에서 배우게 될 것입니다.
>
> Linux는 단지 커널일 뿐이며 사용 가능하려면 셸 및 디스플레이 서버와 같은 많은 사용자 공간 소프트웨어가 필요합니다. macOS의 커널은 XNU라고 불리며 Unix와 유사하고, 현대 Windows 커널은 NT Kernel이라고 불립니다.

모두를 지배하는 두 개의 링

프로세서가 있는 *모드(mode)*(권한 수준 또는 링이라고도 함)는 프로세서가 허용하는 작업을 제어합니다. 현대 아키텍처에는 최소 두 가지 옵션이 있습니다: 커널/슈퍼바이저 모드와 사용자 모드. 아키텍처가 두 개 이상의 모드를 지원할 수 있지만, 요즘에는 커널 모드와 사용자 모드만 일반적으로 사용됩니다.

커널 모드에서는 무엇이든 가능합니다: CPU는 지원되는 모든 명령어를 실행하고 모든 메모리에 액세스할 수 있습니다. 사용자 모드에서는 명령어의 하위 집합만 허용되고, I/O 및 메모리 액세스가 제한되며, 많은 CPU 설정이 잠겨 있습니다. 일반적으로 커널과 드라이버는 커널 모드에서 실행되고 애플리케이션은 사용자 모드에서 실행됩니다.

프로세서는 커널 모드에서 시작합니다. 프로그램을 실행하기 전에 커널은 사용자 모드로의 전환을 시작합니다.



실제 아키텍처에서 프로세서 모드가 어떻게 나타나는지의 예: x86-64에서 현재 권한 수준(CPL)은 `cs`(코드 세그먼트)라는 레지스터에서 읽을 수 있습니다. 특히 CPL은 `cs` 레지스터의 최하위 비트 2개에 포함되어 있습니다. 이 두 비트는 x86-64의 네 가지 가능한 링을 저장할 수 있습니다: 링 0은 커널 모드이고 링 3은 사용자 모드입니다. 링 1과 2는 드라이버 실행을 위해 설계되었지만 소수의 오래된 틈새 운영 체제에서만 사용됩니다. CPL 비트가 `11`이면, 예를 들어 CPU는 링 3에서 실행 중입니다: 사용자 모드.

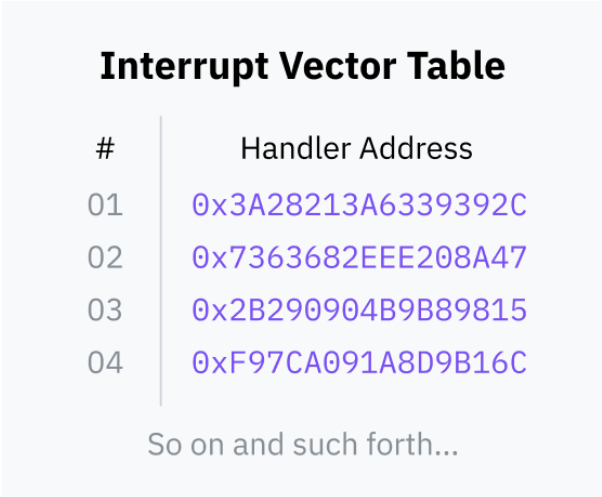
시스템 콜이 대체 뭔가요?

프로그램은 컴퓨터에 대한 완전한 액세스 권한을 신뢰할 수 없기 때문에 사용자 모드에서 실행됩니다. 사용자 모드는 컴퓨터의 대부분에 대한 액세스를 방지하는 역할을 합니다 — 하지만 프로그램은 *어떻게든* I/O에 액세스하고, 메모리를 할당하고, 운영 체제와 상호 작용할 수 있어야 합니다! 이를 위해 사용자 모드에서 실행되는 소프트웨어는 운영 체제 커널에 도움을 요청해야 합니다. 그러면 OS는 프로그램이 악의적인 작업을 하는 것을 방지하기 위해 자체 보안 보호를 구현할 수 있습니다.

OS와 상호 작용하는 코드를 작성한 적이 있다면 `open`, `read`, `fork`, `exit`와 같은 함수를 인식할 것입니다. 몇 가지 추상화 계층 아래에서 이러한 함수는 모두 *시스템 콜*을 사용하여 OS에 도움을 요청합니다. 시스템 콜은 프로그램이 사용자 공간에서 커널 공간으로의 전환을 시작하여 프로그램의 코드에서 OS 코드로 점프할 수 있게 하는 특수 절차입니다.

사용자 공간에서 커널 공간으로의 제어 전송은 *소프트웨어 인터럽트*라는 프로세서 기능을 사용하여 수행됩니다:

1. 부팅 프로세스 중에 운영 체제는 *인터럽트 벡터 테이블*(IVT; x86-64는 이것을 인터럽트 디스크립터 테이블이라고 부릅니다)이라는 테이블을 RAM에 저장하고 CPU에 등록합니다. IVT는 인터럽트 번호를 핸들러 코드 포인터에 매핑합니다.



2. 그런 다음 사용자 공간 프로그램은 INT와 같은 명령어를 사용하여 프로세서에게 IVT에서 주어진 인터럽트 번호를 찾고, 커널 모드로 전환한 다음 명령어 포인터를 IVT에 저장된 메모리 주소로 점프하도록 지시

할 수 있습니다.

이 커널 코드가 완료되면 IRET와 같은 명령어를 사용하여 CPU에게 사용자 모드로 다시 전환하고 명령어 포인터를 인터럽트가 트리거되었을 때의 위치로 반환하도록 지시합니다.

(궁금하시다면, Linux에서 시스템 콜에 사용되는 인터럽트 ID는 `0x80`입니다. Michael Kerrisk의 온라인 manpage 디렉토리에서 Linux 시스템 콜 목록을 읽을 수 있습니다.)

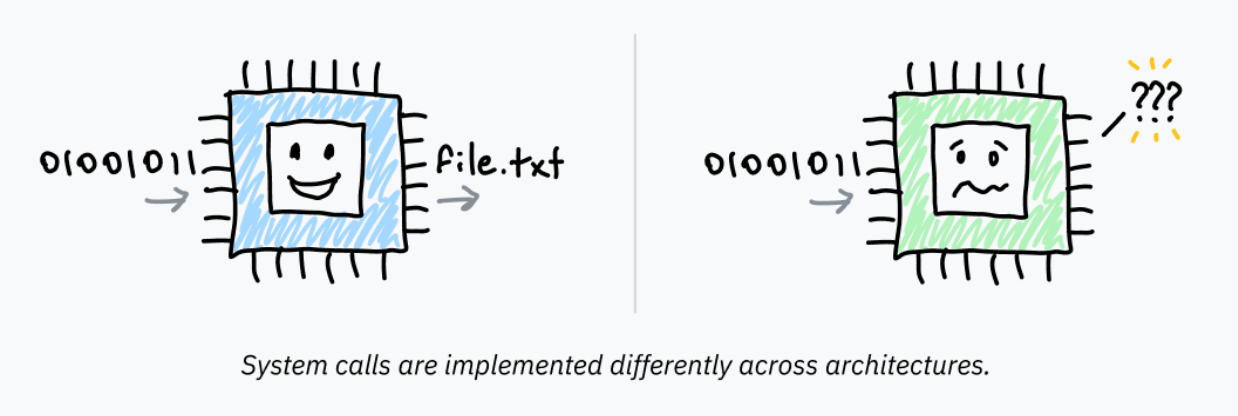
래퍼 API: 인터럽트 추상화

지금까지 시스템 콜에 대해 알고 있는 내용은 다음과 같습니다:

- 사용자 모드 프로그램은 I/O나 메모리에 직접 액세스할 수 없습니다. 외부 세계와 상호 작용하려면 OS에 도움을 요청해야 합니다.
- 프로그램은 INT 및 IRET와 같은 특수 기계어 명령어로 OS에 제어를 위임할 수 있습니다.
- 프로그램은 권한 수준을 직접 전환할 수 없습니다. 소프트웨어 인터럽트는 프로세서가 OS 코드의 어디로 점프할지 *OS에 의해* 미리 구성되었기 때문에 안전합니다. 인터럽트 벡터 테이블은 커널 모드에서만 구성할 수 있습니다.

프로그램은 시스템 콜을 트리거할 때 운영 체제에 데이터를 전달해야 합니다. OS는 실행할 특정 시스템 콜과 시스템 콜 자체가 필요로 하는 데이터(예: 열 파일 이름)를 알아야 합니다. 이 데이터를 전달하는 메커니즘은 운영 체제와 아키텍처에 따라 다르지만, 일반적으로 인터럽트를 트리거하기 전에 특정 레지스터나 스택에 데이터를 배치하여 수행됩니다.

장치 간에 시스템 콜이 호출되는 방식의 차이는 프로그래머가 모든 프로그램에 대해 시스템 콜을 직접 구현하는 것이 엄청나게 비실용적이라는 것을 의미합니다. 이것은 또한 운영 체제가 오래된 시스템을 사용하도록 작성된 모든 프로그램을 손상시킬 우려 없이 인터럽트 처리를 변경할 수 없다는 것을 의미합니다. 마지막으로, 우리는 일반적으로 더 이상 원시 어셈블리로 프로그램을 작성하지 않습니다 — 프로그래머가 파일을 읽거나 메모리를 할당하려고 할 때마다 어셈블리로 내려갈 것으로 기대할 수 없습니다.



System calls are implemented differently across architectures.

따라서 운영 체제는 이러한 인터럽트 위에 추상화 계층을 제공합니다. 필요한 어셈블리 명령어를 래핑하는 재사용 가능한 상위 수준 라이브러리 함수는 Unix 계열 시스템에서 libc에 의해 제공되고 Windows에서는 ntdll.dll이라는 라이브러리의 일부로 제공됩니다. 이러한 라이브러리 함수에 대한 호출 자체는 커널 모드로의 전환을 발생시키지 않으며, 표준 함수 호출일 뿐입니다. 라이브러리 내부에서 어셈블리 코드는 실제로 커널로 제어를 전송하며, 래핑 라이브러리 서브루틴보다 플랫폼에 훨씬 더 의존적입니다.

Unix 계열 시스템에서 실행되는 C에서 `exit(1)`을 호출하면 해당 함수는 내부적으로 기계어를 실행하여 인터럽트를 트리거하며, 시스템 콜의 opcode와 인수를 올바른 레지스터/스택/등에 배치한 후입니다. 컴퓨터는 정말 멋집니다!

속도의 필요성 / CISC해집시다

x86-64와 같은 많은 CISC 아키텍처에는 시스템 콜 패러다임의 보급으로 인해 생성된 시스템 콜용으로 설계된 명령어가 포함되어 있습니다.

Intel과 AMD는 x86-64에서 조율을 잘 못했습니다. 실제로 *두 가지* 최적화된 시스템 콜 명령어 세트가 있습니다. SYSCALL과 SYSENTER는 `INT 0x80`과 같은 명령어에 대한 최적화된 대안입니다. 이에 해당하는 반환 명령어인 SYSRET과 SYSEXIT는 사용자 공간으로 빠르게 전환하고 프로그램 코드를 재개하도록 설계되었습니다.

(AMD 및 Intel 프로세서는 이러한 명령어와 약간 다른 호환성을 가지고 있습니다. `SYSCALL`은 일반적으로 64비트 프로그램에 가장 적합한 옵션이며 `SYSENTER`는 32비트 프로그램에 더 나은 지원을 제공합니다.)

스타일을 대표하는 RISC 아키텍처는 그러한 특수 명령어를 갖지 않는 경향이 있습니다. Apple Silicon이 기반으로 하는 RISC 아키텍처인 AArch64는 시스템 콜 및 소프트웨어 인터럽트 모두에 대해 하나의 인터럽트 명령어만 사용합니다. Mac 사용자들이 잘하고 있다고 생각합니다 :)

휴, 많았네요! 간단히 요약해 보겠습니다:

- 프로세서는 무한 페치-실행 루프에서 명령어를 실행하며 운영 체제나 프로그램에 대한 개념이 없습니다. 일반적으로 레지스터에 저장되는 프로세서의 모드는 실행할 수 있는 명령어를 결정합니다. 운영 체제 코드는 커널 모드에서 실행되고 프로그램을 실행하기 위해 사용자 모드로 전환합니다.
- 바이너리를 실행하기 위해 운영 체제는 사용자 모드로 전환하고 프로세서를 RAM의 코드 진입점으로 가리킵니다. 사용자 모드의 권한만 가지고 있기 때문에 세계와 상호 작용하려는 프로그램은 도움을 위해 OS

코드로 점프해야 합니다. 시스템 콜은 프로그램이 사용자 모드에서 커널 모드로 전환하고 OS 코드로 들어가는 표준화된 방법입니다.

- 프로그램은 일반적으로 공유 라이브러리 함수를 호출하여 이러한 시스템 콜을 사용합니다. 이것들은 OS 커널로 제어를 전송하고 링을 전환하는 소프트웨어 인터럽트 또는 아키텍처별 시스템 콜 명령어에 대한 기계어를 래핑합니다. 커널은 작업을 수행하고 사용자 모드로 다시 전환하여 프로그램 코드로 돌아갑니다.

이전의 첫 번째 질문에 답하는 방법을 알아봅시다:

> CPU가 하나 이상의 프로세스를 추적하지 않고 명령어를 순차적으로 실행하기만 한다면, 실행 중인 프로그램 안에 갇히지 않는 이유는 무엇일까요? 어떻게 여러 프로그램이 동시에 실행될 수 있을까요?

이것에 대한 답은, 친애하는 친구여, Coldplay가 왜 그렇게 인기 있는지에 대한 답이기도 합니다... 시계! (음, 엄밀히 말하면 타이머입니다. 그냥 그 농담을 끼워 넣고 싶었습니다.)

챕터 2: 시간을 나누다

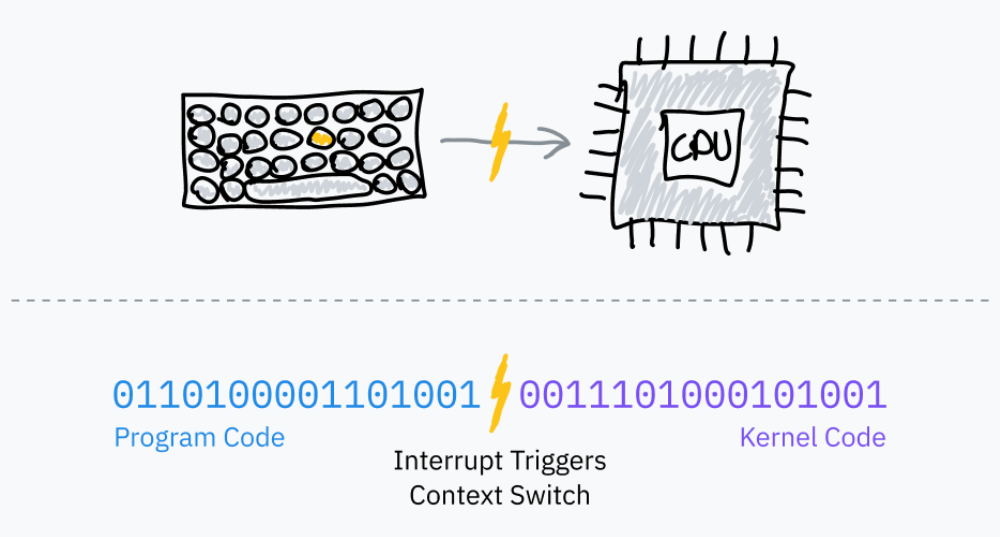
운영체제(Operating System)를 만들고 있는데 사용자가 여러 프로그램을 동시에 실행할 수 있게 하고 싶다고 가정해봅시다. 하지만 멋진 멀티코어 프로세서가 없어서 CPU는 한 번에 하나의 명령어만 실행할 수 있습니다!

다행히도, 당신은 매우 똑똑한 OS 개발자입니다. CPU에서 프로세스들이 순서대로 실행되도록 하여 병렬성(parallelism)을 가짜로 만들 수 있다는 것을 알아냈습니다. 프로세스들을 순환하며 각 프로세스에서 몇 개의 명령어를 실행하면, 어떤 단일 프로세스도 CPU를 독점하지 않으면서 모든 프로세스가 반응할 수 있습니다.

그런데 프로세스를 전환하기 위해 프로그램 코드에서 어떻게 제어권을 다시 가져올까요? 약간의 연구를 해본 결과, 대부분의 컴퓨터에는 타이머 칩이 함께 제공된다는 것을 발견했습니다. 타이머 칩을 프로그래밍하여 일정 시간이 지나면 OS 인터럽트 핸들러로 전환을 트리거할 수 있습니다.

하드웨어 인터럽트(Hardware Interrupts)

앞서 소프트웨어 인터럽트(software interrupts)가 사용자 영역 프로그램에서 OS로 제어권을 넘기는 데 어떻게 사용되는지 이야기했습니다. 이것들이 "소프트웨어" 인터럽트라고 불리는 이유는 프로그램에 의해 자발적으로 트리거되기 때문입니다 — 정상적인 fetch-execute 사이클에서 프로세서가 실행하는 기계 코드가 커널로 제어권을 전환하라고 지시합니다.



OS 스케줄러는 멀티태스킹을 위해 PIT와 같은 *타이머 칩*을 사용하여 하드웨어 인터럽트를 트리거합니다:

1. 프로그램 코드로 점프하기 전에, OS는 타이머 칩을 설정하여 일정 시간 후 인터럽트를 트리거하도록 합니다.
2. OS는 사용자 모드로 전환하고 프로그램의 다음 명령어로 점프합니다.

3. 타이머가 경과하면, 커널 모드로 전환하고 OS 코드로 점프하는 하드웨어 인터럽트를 트리거합니다.
4. 이제 OS는 프로그램이 중단된 위치를 저장하고, 다른 프로그램을 로드하고, 프로세스를 반복할 수 있습니다.

이것을 *선점형 멀티태스킹(preemptive multitasking)*이라고 합니다; 프로세스의 중단을 *선점(preemption)*이라고 합니다. 예를 들어, 같은 기계에서 브라우저로 이 글을 읽으면서 음악을 듣고 있다면, 당신의 컴퓨터는 아마도 초당 수천 번 이 정확한 사이클을 따르고 있을 것입니다.

타임슬라이스 계산(Timeslice Calculation)

*타임슬라이스(timeslice)*는 OS 스케줄러가 프로세스를 선점하기 전에 프로세스가 실행되도록 허용하는 시간입니다. 타임슬라이스를 선택하는 가장 간단한 방법은 모든 프로세스에 동일한 타임슬라이스를 제공하는 것입니다. 아마도 10 ms 범위에서 순서대로 작업을 순환하는 것입니다. 이것을 *고정 타임슬라이스 라운드 로빈(fixed timeslice round-robin)* 스케줄링이라고 합니다.

> **참고: 재미있는 전문 용어 사실!**
>
> 타임슬라이스를 종종 "quantum"이라고 부른다는 것을 알고 계셨나요? 이제 아셨으니, 모든 기술 친구들에게 깊은 인상을 줄 수 있습니다. 저는 이 글의 모든 문장마다 quantum이라고 말하지 않은 것에 대해 많은 칭찬을 받을 자격이 있다고 생각합니다.
>
> 타임슬라이스 전문 용어에 대해 말하자면, 리눅스 커널 개발자들은 고정 주파수 타이머 틱을 세기 위해 jiffy 시간 단위를 사용합니다. 무엇보다도, jiffy는 타임슬라이스의 길이를 측정하는 데 사용됩니다. 리눅스의 jiffy 주파수는 일반적으로 1000 Hz이지만 커널을 컴파일할 때 구성할 수 있습니다.

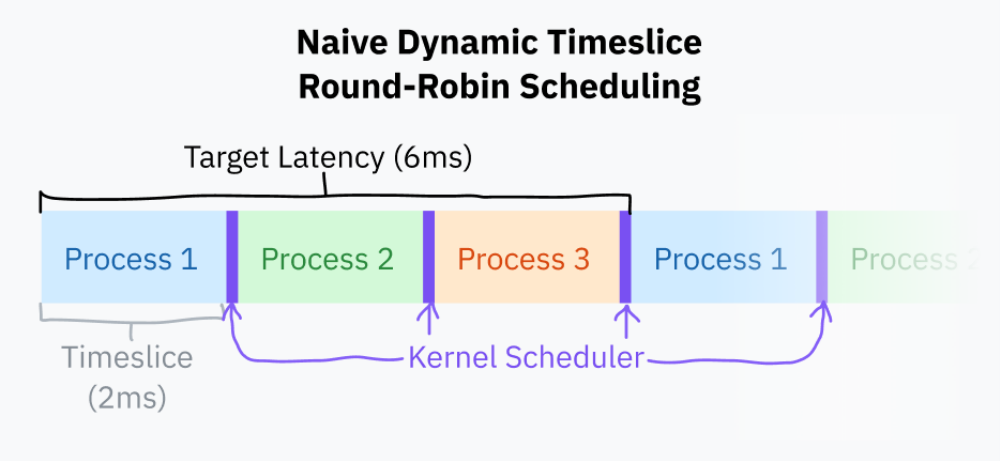
고정 타임슬라이스 스케줄링에 대한 약간의 개선은 *목표 지연 시간(target latency)*을 선택하는 것입니다 — 프로세스가 응답하는 데 이상적인 가장 긴 시간입니다. 목표 지연 시간은 합리적인 수의 프로세스를 가정할 때 프로세스가 선점된 후 실행을 재개하는 데 걸리는 시간입니다. *이것은 시각화하기가 매우 어렵습니다! 걱정하지 마세요, 곧 다이어그램이 나옵니다.*

타임슬라이스는 목표 지연 시간을 총 작업 수로 나누어 계산됩니다; 이것은 프로세스가 적을 때 낭비적인 작업 전환을 제거하기 때문에 고정 타임슬라이스 스케줄링보다 낫습니다. 목표 지연 시간이 15 ms이고 프로세스가 10개인 경우, 각 프로세스는 15/10 또는 1.5 ms를 실행합니다. 프로세스가 3개만 있으면, 각 프로세스는 목표 지연 시간을 여전히 충족하면서 더 긴 5 ms 타임슬라이스를 얻습니다.

프로세스 전환은 현재 프로그램의 전체 상태를 저장하고 다른 프로그램을 복원해야 하므로 계산적으로 비용이 많이 듭니다. 특정 지점을 넘어서면, 너무 작은 타임슬라이스는 프로세스가 너무 빠르게 전환되어 성능 문제를 일으킬 수 있습니다. 타임슬라이스 기간에 하한(*최소 세분성, minimum granularity*)을 부여하는 것이 일반적입니

다. 이것은 최소 세분성이 효과를 발휘할 만큼 충분한 프로세스가 있을 때 목표 지연 시간이 초과된다는 것을 의미합니다.

이 글을 쓰는 시점에서, 리눅스의 스케줄러는 6 ms의 목표 지연 시간과 0.75 ms의 최소 세분성을 사용합니다.



이 기본적인 타임슬라이스 계산을 사용하는 라운드 로빈 스케줄링은 요즘 대부분의 컴퓨터가 하는 것과 비슷합니다. 여전히 약간 순진합니다; 대부분의 운영체제는 프로세스 우선순위와 데드라인을 고려하는 더 복잡한 스케줄러를 가지는 경향이 있습니다. 2007년부터 리눅스는 완전 공정 스케줄러(Completely Fair Scheduler)라는 스케줄러를 사용하고 있습니다. CFS는 작업의 우선순위를 정하고 CPU 시간을 나누기 위해 매우 멋진 컴퓨터 과학적인 것들을 많이 합니다.

OS가 프로세스를 선점할 때마다 가상 주소에서 물리 주소로의 매핑인 *페이지 테이블(page table)*을 포함하여 새 프로그램의 저장된 실행 컨텍스트를 로드해야 합니다. 이것은 CPU에게 다른 *페이지 테이블*을 사용하라고 지시함으로써 달성됩니다. 이것은 또한 프로그램이 서로의 메모리에 액세스하는 것을 방지하는 시스템입니다; 우리는 이 글의 **[5]**장과 **[6]**장에서 이 토끼굴로 내려갈 것입니다.

참고 사항 #1: 커널 선점성(Kernel Preemptability)

지금까지 우리는 사용자 영역 프로세스의 선점과 스케줄링에 대해서만 이야기했습니다. 커널 코드가 시스템 콜을 처리하거나 드라이버 코드를 실행하는 데 너무 오래 걸리면 프로그램이 느리게 느껴질 수 있습니다.

리눅스를 포함한 현대 커널은 선점형 커널(preemptive kernels)입니다. 이것은 커널 코드 자체가 사용자 영역 프로세스처럼 인터럽트되고 스케줄링될 수 있는 방식으로 프로그래밍되었다는 것을 의미합니다.

커널이나 무언가를 작성하지 않는 한 이것을 아는 것은 그다지 중요하지 않지만, 기본적으로 제가 읽은 모든 글에서 언급했기 때문에 저도 언급하려고 합니다! 추가 지식이 나쁜 것은 거의 없습니다.

참고 사항 #2: 역사 수업

고전적인 Mac OS와 NT 이전의 Windows 버전을 포함한 고대 운영체제는 선점형 멀티태스킹의 전신을 사용했습니다. OS가 프로그램을 선점할 시기를 결정하는 대신, 프로그램 자체가 OS에 양보하기로 선택했습니다. 프로그램은 소프트웨어 인터럽트를 트리거하여 "이봐요, 이제 다른 프로그램을 실행할 수 있어요"라고 말했습니다. 이러한 명시적 양보는 OS가 제어권을 되찾고 다음 예약된 프로세스로 전환하는 유일한 방법이었습니다.

이것을 *협력적 멀티태스킹(cooperative multitasking)*이라고 합니다. 이것은 몇 가지 주요 결함이 있습니다: 악의적이거나 단지 잘못 설계된 프로그램이 전체 운영체제를 쉽게 동결시킬 수 있으며, 실시간/시간에 민감한 작업에 대한 시간적 일관성을 보장하는 것이 거의 불가능합니다. 이러한 이유로, 기술 세계는 오래 전에 선점형 멀티태스킹으로 전환했고 결코 되돌아보지 않았습니다.

챕터 3: 프로그램 실행 방법

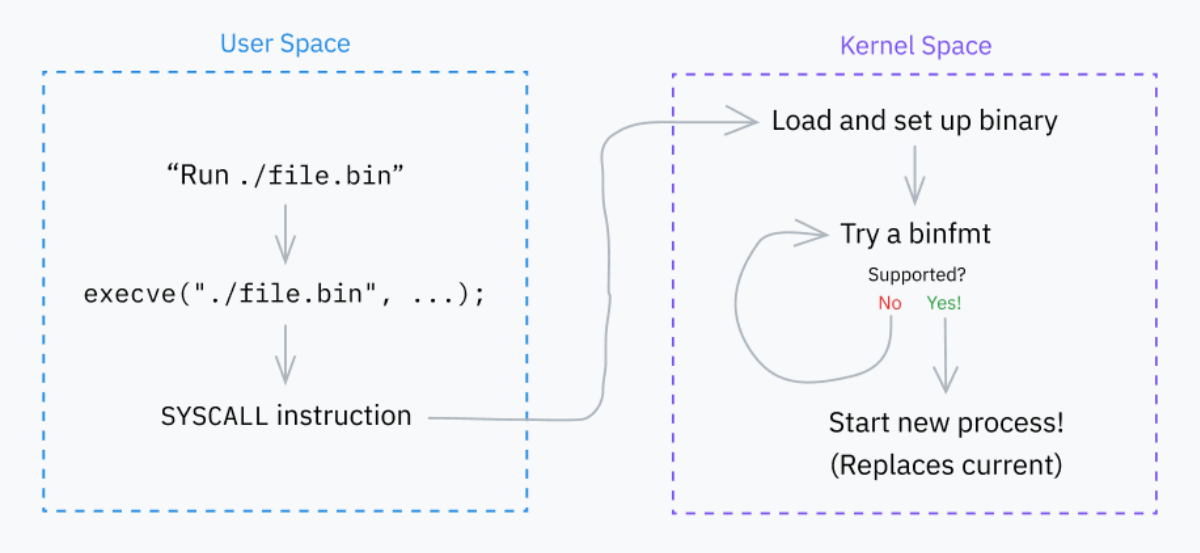
지금까지 우리는 CPU가 실행 파일에서 로드된 기계어를 어떻게 실행하는지, 링 기반 보안이 무엇인지, 시스템 콜이 어떻게 작동하는지를 다뤘습니다. 이번 섹션에서는 Linux 커널을 깊이 파고들어 프로그램이 처음부터 어떻게 로드되고 실행되는지 알아보겠습니다.

특별히 x86-64에서의 Linux를 살펴볼 것입니다. 왜일까요?

- Linux는 데스크톱, 모바일, 서버 사용 사례를 위한 완전한 기능을 갖춘 프로덕션 OS입니다. Linux는 오픈 소스이므로 소스 코드를 읽기만 하면 연구하기가 매우 쉽습니다. 이 글에서 커널 코드를 직접 참조할 것입니다!
- x86-64는 대부분의 현대 데스크톱 컴퓨터가 사용하는 아키텍처이며, 많은 코드의 타겟 아키텍처입니다. 제가 언급하는 x86-64 특정 동작의 하위 집합은 잘 일반화될 것입니다.

우리가 배울 대부분의 내용은 비록 다양한 특정 방식에서 차이가 있더라도 다른 운영 체제와 아키텍처에 잘 일반화될 것입니다.

Exec 시스템 콜의 기본 동작



매우 중요한 시스템 콜인 `execve`부터 시작하겠습니다. 이것은 프로그램을 로드하고, 성공하면 현재 프로세스를 해당 프로그램으로 교체합니다. 몇 가지 다른 시스템 콜(`execlp`, `execvpe` 등)이 존재하지만, 모두 다양한 방식으로 `execve` 위에 계층화되어 있습니다.

> **참고: `execveat`**
>
> `execve`는 *실제로는* `execveat` 위에 구축되어 있습니다. `execveat`은 일부 구성 옵션과 함께 프로그램을 실행하는 보다 일반적인 시스템 콜입니다. 간단함을 위해 대부분 `execve`에 대해 이야기하겠습니다; 유일한 차이점은 `execveat`에 몇 가지 기본값을 제공한다는 것입니다.
>
> `ve`가 무엇을 의미하는지 궁금하신가요? `v`는 한 매개변수가 인수의 벡터(목록)(`argv`)임을 의미하고, `e`는 다른 매개변수가 환경 변수의 벡터(`envp`)임을 의미합니다. 다른 다양한 exec 시스템 콜은 다른 호출 서명을 지정하기 위해 다른 접미사를 갖습니다. `execveat`의 `at`은 단지 "at"이며, `execve`를 실행할 위치를 지정하기 때문입니다.

`execve`의 호출 서명은 다음과 같습니다:

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- `filename` 인수는 실행할 프로그램의 경로를 지정합니다.
- `argv`는 프로그램에 대한 인수의 널 종료(마지막 항목이 널 포인터임을 의미) 목록입니다. C main 함수에 전달되는 것을 흔히 볼 수 있는 `argc` 인수는 실제로 나중에 시스템 콜에 의해 계산되므로 널 종료가 필요합니다.
- `envp` 인수는 애플리케이션의 컨텍스트로 사용되는 환경 변수의 또 다른 널 종료 목록을 포함합니다. 그것들은... 관례적으로 `KEY=VALUE` 쌍입니다. *관례적으로.* 컴퓨터가 좋아요.

재미있는 사실! 프로그램의 첫 번째 인수가 프로그램의 이름이라는 그 관례를 아시나요? 그것은 *순전히 관례*이며, 실제로 `execve` 시스템 콜 자체에 의해 설정되지 않습니다! 첫 번째 인수는 `argv` 인수의 첫 번째 항목으로 `execve`에 전달되는 모든 것이 될 것이며, 프로그램 이름과 아무 관련이 없더라도 마찬가지입니다.

흥미롭게도, `execve`는 `argv[0]`이 프로그램 이름이라고 가정하는 일부 코드를 가지고 있습니다. 해석된 스크립팅 언어에 대해 이야기할 때 이에 대해 더 자세히 설명하겠습니다.

단계 0: 정의

우리는 이미 시스템 콜이 어떻게 작동하는지 알고 있지만, 실제 코드 예제는 본 적이 없습니다! Linux 커널의 소스 코드를 살펴보고 `execve`가 내부적으로 어떻게 정의되는지 봅시다:

fs/exec.c

```
2105  SYSCALL_DEFINE3(execve,
2106                  const char __user *, filename,
2107                  const char __user *const __user *, argv,
2108                  const char __user *const __user *, envp)
2109  {
2110          return do_execve(getname(filename), argv, envp);
2111  }
```

`SYSCALL_DEFINE3`는 3개 인수 시스템 콜의 코드를 정의하기 위한 매크로입니다.

> 저는 왜 arity가 매크로 이름에 하드코딩되어 있는지 궁금했습니다; 검색해 보니 이것이 일부 보안 취약점을 수정하기 위한 해결 방법이었다는 것을 알게 되었습니다.

filename 인수는 `getname()` 함수에 전달되며, 이 함수는 사용자 공간에서 커널 공간으로 문자열을 복사하고 일부 사용량 추적 작업을 수행합니다. `include/linux/fs.h`에 정의된 `filename` 구조체를 반환합니다. 이것은 사용자 공간의 원래 문자열에 대한 포인터와 커널 공간에 복사된 값에 대한 새 포인터를 저장합니다:

include/linux/fs.h

```
2294  struct filename {
2295          const char              *name;  /* pointer to actual string */
2296          const __user char       *uptr;  /* original userland pointer */
2297          int                     refcnt;
2298          struct audit_names      *aname;
2299          const char              iname[];
2300  };
```

그런 다음 `execve` 시스템 콜은 `do_execve()` 함수를 호출합니다. 이것은 차례로 일부 기본값과 함께 `do_execveat_common()`을 호출합니다. 앞서 언급한 `execveat` 시스템 콜도 `do_execveat_common()`을 호출하지만, 더 많은 사용자 제공 옵션을 전달합니다.

아래 스니펫에서, 나는 `do_execve`와 `do_execveat`의 정의를 모두 포함했습니다:

fs/exec.c

```
2028  static int do_execve(struct filename *filename,
2029          const char __user *const __user *__argv,
2030          const char __user *const __user *__envp)
2031  {
2032          struct user_arg_ptr argv = { .ptr.native = __argv };
2033          struct user_arg_ptr envp = { .ptr.native = __envp };
2034          return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
2035  }
2036
2037  static int do_execveat(int fd, struct filename *filename,
2038                  const char __user *const __user *__argv,
2039                  const char __user *const __user *__envp,
2040                  int flags)
2041  {
2042          struct user_arg_ptr argv = { .ptr.native = __argv };
2043          struct user_arg_ptr envp = { .ptr.native = __envp };
2044
2045          return do_execveat_common(fd, filename, argv, envp, flags);
2046  }
```

[spacing sic]

`execveat`에서, 파일 디스크립터(*어떤 리소스*를 가리키는 일종의 id)가 시스템 콜에 전달된 다음 `do_execveat_common`에 전달됩니다. 이것은 프로그램을 실행할 디렉토리를 상대적으로 지정합니다.

`execve`에서는 파일 디스크립터 인수에 특수 값인 `AT_FDCWD`가 사용됩니다. 이것은 Linux 커널의 공유 상수로, 함수에 경로 이름을 현재 작업 디렉토리에 상대적인 것으로 해석하도록 지시합니다. 파일 디스크립터를 허용하는 함수는 일반적으로 `if (fd == AT_FDCWD) { /* special codepath */ }`와 같은 수동 검사를 포함합니다.

단계 1: 설정

우리는 이제 프로그램 실행을 처리하는 핵심 함수인 `do_execveat_common`에 도달했습니다. 이 함수가 무엇을 하는지에 대한 더 큰 그림을 보기 위해 코드를 응시하는 것에서 잠깐 물러나겠습니다.

`do_execveat_common`의 첫 번째 주요 작업은 `linux_binprm`이라는 구조체를 설정하는 것입니다. 전체 구조체 정의의 복사본을 포함하지는 않겠지만, 살펴볼 몇 가지 중요한 필드가 있습니다:

- `mm_struct` 및 `vm_area_struct`와 같은 데이터 구조는 새 프로그램을 위한 가상 메모리 관리를 준비하기 위해 정의됩니다.

- `argc`와 `envc`는 계산되어 프로그램에 전달되도록 저장됩니다.
- `filename`과 `interp`는 각각 프로그램의 파일 이름과 인터프리터를 저장합니다. 이것들은 서로 같게 시작하지만 일부 경우에 변경될 수 있습니다: 그러한 경우 중 하나는 shebang이 있는 해석된 스크립트를 실행할 때입니다. 예를 들어, Python 프로그램을 실행할 때 `filename`은 소스 파일을 가리키지만 `interp`는 Python 인터프리터의 경로입니다.
- `buf`는 실행할 파일의 처음 256바이트로 채워진 배열입니다. 파일의 형식을 감지하고 스크립트 shebang을 로드하는 데 사용됩니다.

(TIL: binprm은 **bin**ary **pr**og**r**a**m**을 의미합니다.)

이 버퍼 `buf`를 더 자세히 살펴봅시다:

linux_binprm @ include/linux/binfmts.h

```
64          char buf[BINPRM_BUF_SIZE];
```

보시다시피, 그 길이는 상수 `BINPRM_BUF_SIZE`로 정의됩니다. 코드베이스에서 이 문자열을 검색하면, `include/uapi/linux/binfmts.h`에서 이에 대한 정의를 찾을 수 있습니다:

include/uapi/linux/binfmts.h

```
18  /* sizeof(linux_binprm->buf) */
19  #define BINPRM_BUF_SIZE 256
```

따라서 커널은 실행된 파일의 처음 256바이트를 이 메모리 버퍼에 로드합니다.

> **참고: UAPI가 뭔가요?**
>
> 위 코드의 경로에 `/uapi/`가 포함되어 있음을 알 수 있습니다. 왜 길이가 `linux_binprm` 구조체와 같은 파일인 `include/linux/binfmts.h`에 정의되지 않았을까요?
>
> UAPI는 "userspace API"를 의미합니다. 이 경우, 누군가가 버퍼의 길이가 커널의 공개 API의 일부여야 한다고 결정했음을 의미합니다. 이론적으로 모든 UAPI는 사용자 공간에 노출되고, 모든 비-UAPI는 커널 코드에 비공개입니다.
>
> 커널과 사용자 공간 코드는 원래 하나의 뒤죽박죽 덩어리로 공존했습니다. 2012년에 UAPI 코드는 유지 관리성을 개선하기 위한 시도로 별도의 디렉토리로 리팩토링되었습니다.

단계 2: Binfmts

커널의 다음 주요 작업은 여러 "binfmt"(바이너리 형식) 핸들러를 반복하는 것입니다. 이러한 핸들러는 `fs/binfmt_elf.c` 및 `fs/binfmt_flat.c`와 같은 파일에 정의되어 있습니다. 커널 모듈도 자체 binfmt 핸들러를 풀에 추가할 수 있습니다.

각 핸들러는 `linux_binprm` 구조체를 받아 핸들러가 프로그램의 형식을 이해하는지 확인하는 `load_binary()` 함수를 노출합니다.

이것은 종종 버퍼에서 매직 넘버를 찾거나, (또한 버퍼에서) 프로그램의 시작을 디코딩하려고 시도하거나, 파일 확장자를 확인하는 것을 포함합니다. 핸들러가 형식을 지원하면 실행을 위해 프로그램을 준비하고 성공 코드를 반환합니다. 그렇지 않으면 일찍 종료하고 오류 코드를 반환합니다.

커널은 성공하는 것에 도달할 때까지 각 binfmt의 `load_binary()` 함수를 시도합니다. 때때로 이것들은 재귀적으로 실행됩니다; 예를 들어, 스크립트에 인터프리터가 지정되어 있고 그 인터프리터가 그 자체로 스크립트인 경우, 계층 구조는 `binfmt_script` > `binfmt_script` > `binfmt_elf`일 수 있습니다 (여기서 ELF는 체인의 끝에 있는 실행 가능한 형식입니다).

형식 하이라이트: 스크립트

Linux가 지원하는 많은 형식 중에서 `binfmt_script`는 제가 특별히 이야기하고 싶은 첫 번째 형식입니다.

shebang을 읽거나 쓴 적이 있나요? 인터프리터의 경로를 지정하는 일부 스크립트의 시작 부분에 있는 그 줄 말이죠?

```
1  #!/bin/bash
```

저는 항상 이것들이 셸에 의해 처리된다고 가정했지만, 아닙니다! Shebang은 실제로 커널의 기능이며, 스크립트는 다른 모든 프로그램과 동일한 시스템 콜로 실행됩니다. 컴퓨터는 *정말 멋집니다*.

`fs/binfmt_script.c`가 파일이 shebang으로 시작하는지 어떻게 확인하는지 살펴보세요:

load_script @ fs/binfmt_script.c

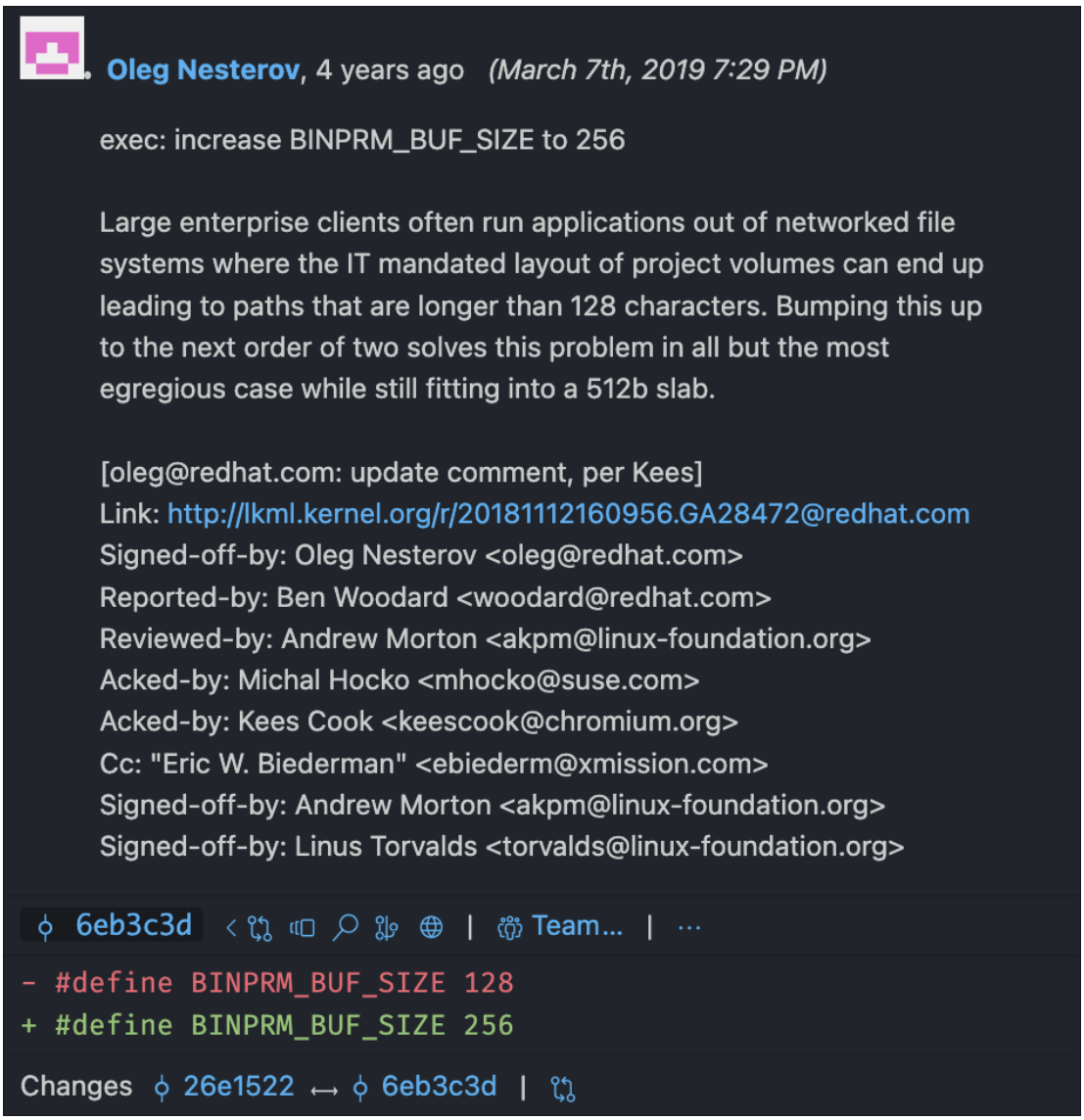
```
40          /* Not ours to exec if we don't start with "#!". */
41          if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!'))
42                  return -ENOEXEC;
```

파일이 shebang으로 시작하면, binfmt 핸들러는 인터프리터 경로와 경로 뒤의 공백으로 구분된 인수를 읽습니다. 개행 문자나 버퍼의 끝에 도달하면 멈춥니다.

여기에서 두 가지 흥미롭고 이상한 일이 일어나고 있습니다.

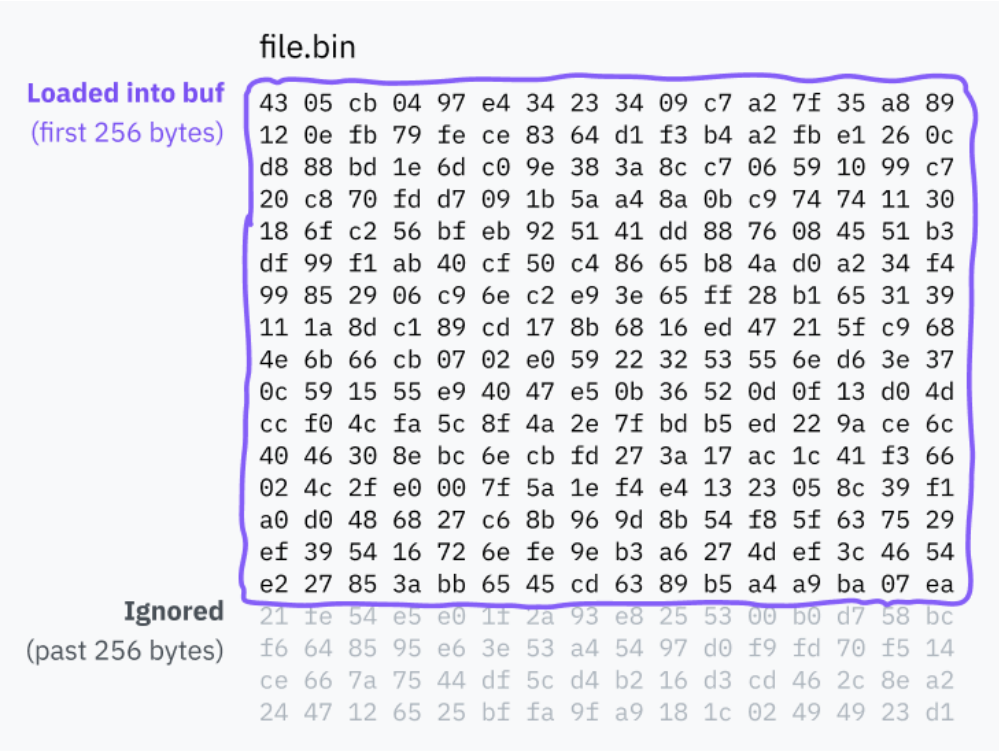
첫째, 파일의 처음 256바이트로 채워진 `linux_binprm`의 버퍼를 기억하시나요? 그것은 실행 가능한 형식 감지에 사용되지만, 그 동일한 버퍼는 또한 `binfmt_script`에서 shebang이 읽혀지는 곳입니다.

제 연구 중에, 버퍼를 128바이트 길이로 설명한 글을 읽었습니다. 그 글이 게시된 후 어느 시점에, 길이가 256바이트로 두 배가 되었습니다! 왜인지 궁금해서, Linux 소스 코드에서 `BINPRM_BUF_SIZE`가 정의된 줄의 Git blame — 특정 코드 줄을 편집한 모든 사람의 로그 — 를 확인했습니다. 놀랍게도...



컴퓨터는 정말 멋집니다!

Shebang은 커널에 의해 처리되고, 전체 파일을 로드하는 대신 `buf`에서 가져오기 때문에, *항상* `buf`의 길이로 잘립니다. 분명히, 4년 전에 누군가가 커널이 >128자 경로를 자르는 것에 짜증이 났고, 그들의 해결책은 버퍼 크기를 두 배로 늘려 절단 지점을 두 배로 늘리는 것이었습니다! 오늘날, 여러분의 Linux 머신에서 256자보다 긴 shebang 줄이 있으면 256자를 넘는 모든 것이 *완전히 손실*됩니다.



이것 때문에 버그가 있다고 상상해보세요. 코드를 망가뜨리는 것의 근본 원인을 알아내려고 노력한다고 상상해보세요. 문제가 Linux 커널 깊숙한 곳에 있다는 것을 발견했을 때 어떤 기분일지 상상해보세요. 경로의 일부가 신비롭게 사라진 것을 발견하는 대규모 기업의 다음 IT 담당자에게 화가 있을 것입니다.

두 번째 이상한 것: `argv[0]`이 프로그램 이름이라는 것은 *관례*일 뿐이며, 호출자가 원하는 `argv`를 exec 시스템 콜에 전달할 수 있고 그것이 통제 없이 통과할 것이라는 것을 기억하시나요?

`binfmt_script`가 `argv[0]`이 프로그램 이름이라고 *가정*하는 곳 중 하나라는 것이 우연히 발생합니다. 항상 `argv[0]`을 제거하고, 그런 다음 `argv`의 시작 부분에 다음을 추가합니다:

- 인터프리터로의 경로
- 인터프리터에 대한 인수
- 스크립트의 파일 이름

> **예: 인수 수정**
>
> 샘플 `execve` 호출을 살펴봅시다:
>
> ```
> // Arguments: filename, argv, envp
> execve("./script", [ "A", "B", "C" ], []);
> ```
>
> 이 가상의 `script` 파일은 첫 번째 줄로 다음 shebang을 가지고 있습니다:
>
> script
>
> ```
> 1  #!/usr/bin/node --experimental-module
> ```
>
> Node 인터프리터에 최종적으로 전달되는 수정된 `argv`는 다음과 같습니다:
>
> ```
> [ "/usr/bin/node", "--experimental-module", "./script", "B", "C" ]
> ```

`argv`를 업데이트한 후, 핸들러는 `linux_binprm.interp`를 인터프리터 경로(이 경우 Node 바이너리)로 설정하여 실행을 위한 파일 준비를 마칩니다. 마지막으로, 프로그램 실행 준비 성공을 나타내기 위해 0을 반환합니다.

형식 하이라이트: 기타 인터프리터

또 다른 흥미로운 핸들러는 `binfmt_misc`입니다. 이것은 `/proc/sys/fs/binfmt_misc/`에 특수 파일 시스템을 마운트함으로써 사용자 공간 구성을 통해 일부 제한된 형식을 추가할 수 있는 능력을 엽니다. 프로그램은 이 디렉토리의 파일에 특별히 형식화된 쓰기를 수행하여 자체 핸들러를 추가할 수 있습니다. 각 구성 항목은 다음을 지정합니다:

- 파일 형식을 감지하는 방법. 이것은 특정 오프셋의 매직 넘버나 찾을 파일 확장자를 지정할 수 있습니다.
- 인터프리터 실행 파일의 경로. 인터프리터 인수를 지정할 방법이 없으므로 원하는 경우 래퍼 스크립트가 필요합니다.
- `binfmt_misc`가 `argv`를 업데이트하는 방법을 지정하는 하나를 포함한 일부 구성 플래그.

이 `binfmt_misc` 시스템은 종종 Java 설치에서 사용되며, `0xCAFEBABE` 매직 바이트로 클래스 파일을 감지하고 확장자로 JAR 파일을 감지하도록 구성됩니다. 제 특정 시스템에서는 .pyc 확장자로 Python 바이트코드를

감지하고 적절한 핸들러에 전달하도록 구성된 핸들러가 있습니다.

이것은 프로그램 설치 프로그램이 높은 권한의 커널 코드를 작성할 필요 없이 자체 형식에 대한 지원을 추가할 수 있게 하는 꽤 멋진 방법입니다.

결국에는 (Linkin Park 노래가 아님)

exec 시스템 콜은 항상 두 가지 경로 중 하나로 끝날 것입니다:

- 여러 계층의 스크립트 인터프리터를 거친 후 결국 이해하는 실행 가능한 바이너리 형식에 도달하여 해당 코드를 실행합니다. 이 시점에서 이전 코드는 교체되었습니다.
- ... 또는 모든 옵션을 소진하고 꼬리를 다리 사이에 끼고 호출 프로그램에 오류 코드를 반환합니다.

Unix 계열 시스템을 사용한 적이 있다면, 터미널에서 실행된 셸 스크립트가 shebang 줄이나 `.sh` 확장자가 없어도 여전히 실행된다는 것을 알아차렸을 것입니다. 비-Windows 터미널이 있다면 지금 바로 테스트해볼 수 있습니다:

Shell session

```
$ echo "echo hello" > ./file
$ chmod +x ./file
$ ./file
hello
```

(`chmod +x`는 OS에 파일이 실행 가능하다고 알려줍니다. 그렇지 않으면 실행할 수 없습니다.)

그렇다면, 왜 셸 스크립트가 셸 스크립트로 실행될까요? 커널의 형식 핸들러는 식별 가능한 레이블이 없는 셸 스크립트를 감지할 명확한 방법이 없어야 합니다!

글쎄요, 이 동작은 커널의 일부가 아니라는 것이 밝혀졌습니다. 실제로 *셸*이 실패 사례를 처리하는 일반적인 방법입니다.

셸을 사용하여 파일을 실행하고 exec 시스템 콜이 실패하면, 대부분의 셸은 첫 번째 인수로 파일 이름을 사용하여 셸을 실행하여 *파일을 셸 스크립트로 다시 실행하려고 시도*합니다. Bash는 일반적으로 자신을 이 인터프리터로 사용하는 반면, ZSH는 `sh`가 무엇이든 사용하며, 일반적으로 Bourne shell입니다.

이 동작이 매우 일반적인 이유는 *POSIX*에 지정되어 있기 때문입니다. POSIX는 Unix 시스템 간에 코드를 이식 가능하게 만들기 위해 설계된 오래된 표준입니다. POSIX는 대부분의 도구나 운영 체제에 의해 엄격하게 따라지지는 않지만, 많은 관례가 여전히 공유됩니다.

> [exec 시스템 콜이] `[ENOEXEC]` 오류와 동등한 오류로 인해 실패하면, **셸은 명령 이름을 첫 번째 피연산자로 하여 셸을 호출한 것과 동등한 명령을 실행해야 합니다**, 나머지 인수는 새 셸에 전달됩니다. 실행 가능한 파일이 텍스트 파일이 아니면, 셸은 이 명령 실행을 우회할 수 있습니다. 이 경우 오류 메시지를 작성하고 종료 상태 126을 반환해야 합니다.
>
> *출처: Shell Command Language, POSIX.1-2017*

컴퓨터는 정말 멋집니다!

챕터 4: 엘프 로드가 되다

우리는 이제 `execve`를 꽤 철저하게 이해했습니다. 대부분의 경로의 끝에서 커널은 실행할 기계어를 포함하는 최종 프로그램에 도달할 것입니다. 일반적으로, 실제로 코드로 점프하기 전에 설정 프로세스가 필요합니다 — 예를 들어, 프로그램의 다른 부분들이 메모리의 올바른 위치에 로드되어야 합니다. 각 프로그램은 다른 것들을 위해 다른 양의 메모리가 필요하므로, 실행을 위해 프로그램을 설정하는 방법을 지정하는 표준 파일 형식이 있습니다. Linux는 많은 그러한 형식을 지원하지만, 가장 일반적인 형식은 단연코 *ELF* (executable and linkable format)입니다.

(Nicky Case에게 사랑스러운 그림을 그려주셔서 감사합니다.)

> **참고: 엘프는 어디에나 있나요?**
>
> Linux에서 앱이나 명령줄 프로그램을 실행할 때, 그것이 ELF 바이너리일 가능성이 매우 높습니다. 그러나 macOS에서는 사실상의 형식이 Mach-O입니다. Mach-O는 ELF와 모든 동일한 작업을 수행하지만 다르게 구조화되어 있습니다. Windows에서 .exe 파일은 Portable Executable 형식을 사용하며, 이것은 또한 동일한 개념을 가진 다른 형식입니다.

Linux 커널에서 ELF 바이너리는 `binfmt_elf` 핸들러에 의해 처리되며, 이것은 많은 다른 핸들러보다 더 복잡하고 수천 줄의 코드를 포함합니다. 이것은 ELF 파일에서 특정 세부 사항을 파싱하고 프로세스를 메모리에 로드

하고 실행하는 데 사용하는 책임이 있습니다.

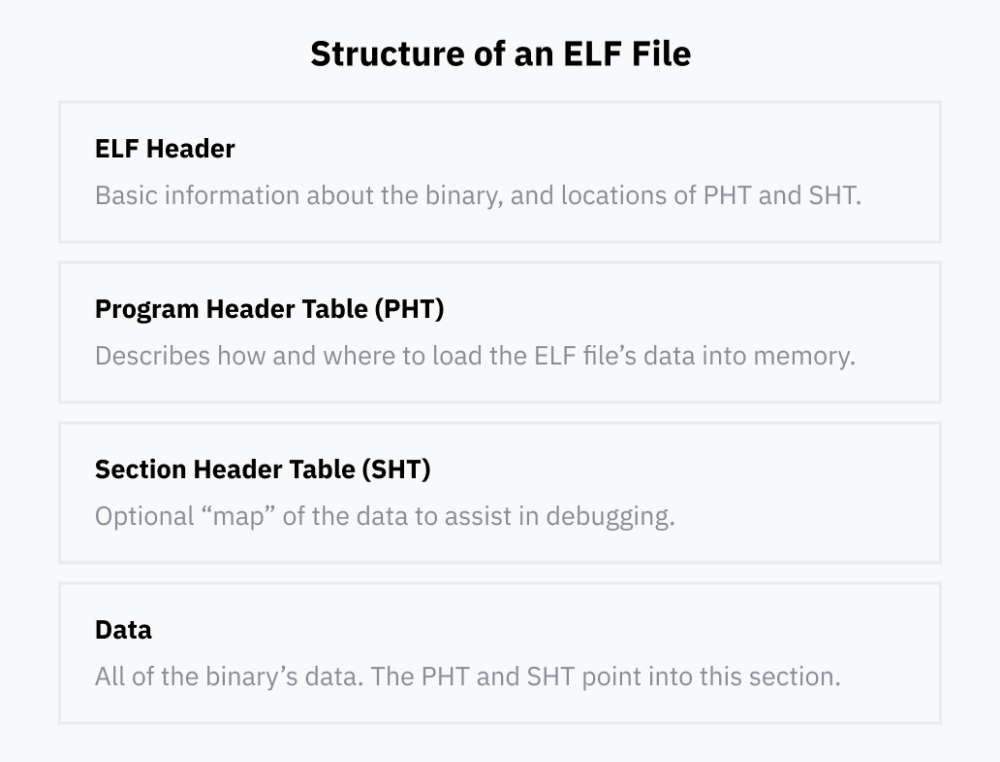
줄 수로 binfmt 핸들러를 정렬하기 위해 일부 명령줄 kung fu를 실행했습니다:

Shell session

```
$ wc -l binfmt_* | sort -nr | sed 1d
    2181 binfmt_elf.c
    1658 binfmt_elf_fdpic.c
     944 binfmt_flat.c
     836 binfmt_misc.c
     158 binfmt_script.c
      64 binfmt_elf_test.c
```

파일 구조

`binfmt_elf`가 ELF 파일을 실행하는 방법을 더 깊이 살펴보기 전에, 파일 형식 자체를 살펴봅시다. ELF 파일은 일반적으로 네 부분으로 구성됩니다:



ELF 헤더

모든 ELF 파일에는 ELF 헤더가 있습니다. 이것은 다음과 같은 바이너리에 대한 기본 정보를 전달하는 매우 중요한 작업을 가지고 있습니다:

- 실행되도록 설계된 프로세서. ELF 파일은 ARM 및 x86과 같은 다른 프로세서 유형에 대한 기계어를 포함할 수 있습니다.
- 바이너리가 실행 파일로 자체적으로 실행되도록 의도되었는지, 아니면 다른 프로그램에 의해 "동적으로 링크된 라이브러리"로 로드되도록 의도되었는지. 동적 링킹이 무엇인지 곧 자세히 살펴보겠습니다.
- 실행 파일의 진입점. 나중 섹션은 ELF 파일에 포함된 데이터를 메모리에 정확히 어디에 로드할지 지정합니다. 진입점은 전체 프로세스가 로드된 후 메모리에서 첫 번째 기계어 명령어가 있는 위치를 가리키는 메모리 주소입니다.

ELF 헤더는 항상 파일의 시작 부분에 있습니다. 파일 내 어디에나 있을 수 있는 프로그램 헤더 테이블과 섹션 헤더의 위치를 지정합니다. 그 테이블들은 차례로 파일의 다른 곳에 저장된 데이터를 가리킵니다.

프로그램 헤더 테이블

프로그램 헤더 테이블은 런타임에 바이너리를 로드하고 실행하는 방법에 대한 특정 세부 사항을 포함하는 일련의 항목입니다. 각 항목에는 지정하는 세부 사항을 나타내는 유형 필드가 있습니다 — 예를 들어, `PT_LOAD`는 메모리에 로드되어야 하는 데이터를 포함함을 의미하지만, `PT_NOTE`는 세그먼트가 반드시 어디에나 로드될 필요가 없는 정보 텍스트를 포함함을 의미합니다.

Common Program Header Types

PT_LOAD	Data to be loaded into memory.
PT_NOTE	Freeform text like copyright notices, version info, etc.
PT_DYNAMIC	Info about dynamic linking.
PT_INTERP	Path to the location of an "ELF interpreter."

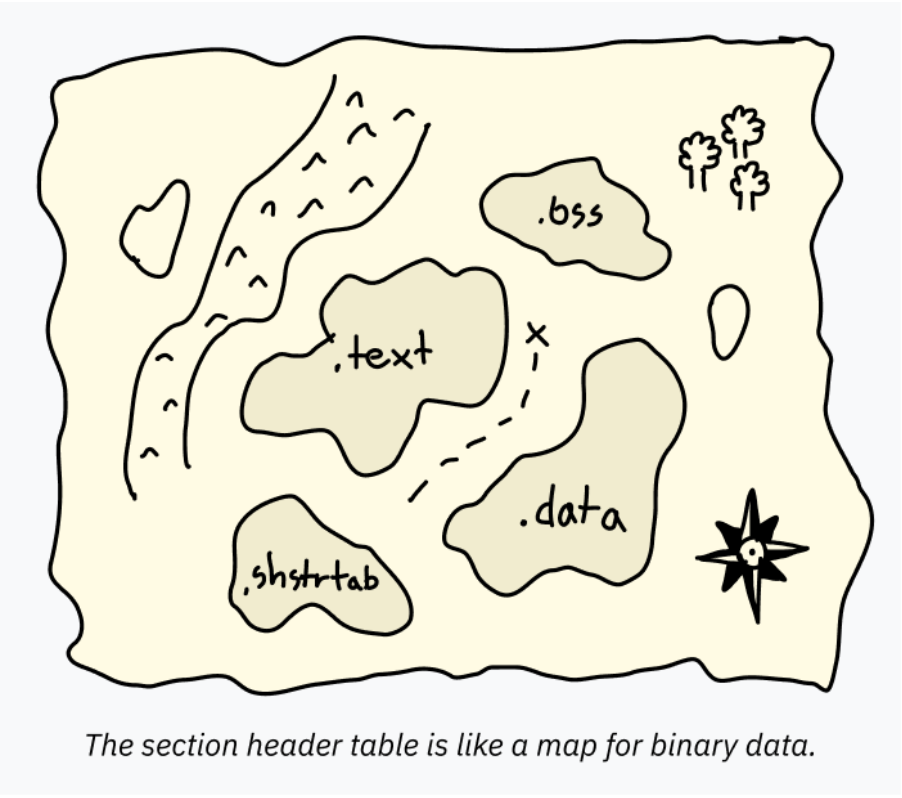
각 항목은 데이터가 파일의 어디에 있는지, 그리고 때때로 데이터를 메모리에 로드하는 방법에 대한 정보를 지정합니다:

- ELF 파일 내에서 데이터의 위치를 가리킵니다.
- 데이터가 로드되어야 하는 가상 메모리 주소를 지정할 수 있습니다. 이것은 일반적으로 세그먼트가 메모리에 로드되도록 의도되지 않은 경우 비어 있습니다.

- 두 필드는 데이터의 길이를 지정합니다: 하나는 파일의 데이터 길이를 위한 것이고, 하나는 생성될 메모리 영역의 길이를 위한 것입니다. 메모리 영역 길이가 파일의 길이보다 길면, 추가 메모리는 0으로 채워집니다. 이것은 런타임에 사용할 정적 메모리 세그먼트를 원할 수 있는 프로그램에 유익합니다; 이러한 빈 메모리 세그먼트는 일반적으로 BSS 세그먼트라고 불립니다.
- 마지막으로, 플래그 필드는 메모리에 로드된 경우 허용되어야 하는 작업을 지정합니다: `PF_R`은 읽을 수 있게 만들고, `PF_W`는 쓸 수 있게 만들며, `PF_X`는 CPU에서 실행될 수 있어야 하는 코드임을 의미합니다.

섹션 헤더 테이블

섹션 헤더 테이블은 *섹션*에 대한 정보를 포함하는 일련의 항목입니다. 이 섹션 정보는 지도와 같아서 ELF 파일 내부의 데이터를 도식화합니다. 디버거와 같은 프로그램이 데이터의 다른 부분의 의도된 용도를 쉽게 이해할 수 있게 합니다.



The section header table is like a map for binary data.

예를 들어, 프로그램 헤더 테이블은 함께 메모리에 로드될 대용량 데이터 덩어리를 지정할 수 있습니다. 그 단일 `PT_LOAD` 블록은 코드와 전역 변수를 모두 포함할 수 있습니다! 프로그램을 *실행*하기 위해 별도로 지정할 필요가 없습니다; CPU는 진입점에서 시작하여 앞으로 나아가며, 프로그램이 요청하는 곳에서 데이터에 액세스합니다. 그러나 프로그램을 *분석*하기 위한 디버거와 같은 소프트웨어는 각 영역이 정확히 어디에서 시작하고 끝나는지 알아야 하며, 그렇지 않으면 "hello"라고 말하는 텍스트를 코드로 디코딩하려고 시도할 수 있습니다 (그리고 그것은 유효한 코드가 아니므로 폭발합니다). 이 정보는 섹션 헤더 테이블에 저장됩니다.

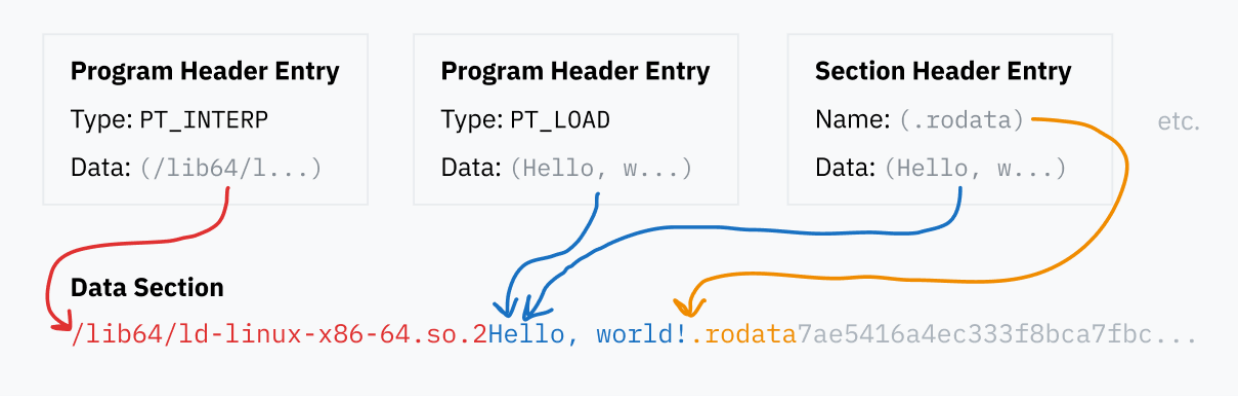
일반적으로 포함되지만, 섹션 헤더 테이블은 실제로 선택 사항입니다. ELF 파일은 섹션 헤더 테이블이 완전히 제거되어도 완벽하게 실행될 수 있으며, 코드가 무엇을 하는지 숨기려는 개발자는 때때로 의도적으로 ELF 바이너리에서 섹션 헤더 테이블을 제거하거나 손상시켜 디코딩을 더 어렵게 만듭니다.

각 섹션에는 이름, 유형, 그리고 사용 및 디코딩 방법을 지정하는 일부 플래그가 있습니다. 표준 이름은 관례적으로 점으로 시작합니다. 가장 일반적인 섹션은 다음과 같습니다:

- `.text`: CPU에서 메모리에 로드되어 실행될 기계어. 실행 가능하다고 표시하기 위한 `SHF_EXECINSTR` 플래그와 실행을 위해 메모리에 로드된다는 의미의 `SHF_ALLOC` 플래그가 있는 `SHT_PROGBITS` 유형입니다. (이름에 속지 마세요, 여전히 그냥 바이너리 기계어입니다! 읽을 수 있는 "텍스트"가 아님에도 `.text`라고 불리는 것이 항상 다소 이상하다고 생각했습니다.)
- `.data`: 실행 파일에 하드코딩되어 메모리에 로드될 초기화된 데이터. 예를 들어, 일부 텍스트를 포함하는 전역 변수가 이 섹션에 있을 수 있습니다. 저수준 코드를 작성하면, 이것은 정적 변수가 들어가는 섹션입니다. 이것 또한 `SHT_PROGBITS` 유형을 가지며, 이것은 단순히 섹션이 "프로그램을 위한 정보"를 포함한다는 것을 의미합니다. 플래그는 쓰기 가능한 메모리로 표시하기 위한 `SHF_ALLOC`과 `SHF_WRITE`입니다.
- `.bss`: 앞서 0으로 시작하는 할당된 메모리를 갖는 것이 일반적이라고 언급했습니다. ELF 파일에 많은 빈 바이트를 포함하는 것은 낭비이므로, BSS라는 특수 세그먼트 유형이 사용됩니다. 디버깅 중에 BSS 세그먼트에 대해 아는 것이 도움이 되므로, 할당될 메모리의 길이를 지정하는 섹션 헤더 테이블 항목도 있습니다. `SHT_NOBITS` 유형이며, `SHF_ALLOC`과 `SHF_WRITE`로 플래그가 지정됩니다.
- `.rodata`: 이것은 `.data`와 같지만 읽기 전용입니다. `printf("Hello, world!")`를 실행하는 매우 기본적인 C 프로그램에서 "Hello world!" 문자열은 `.rodata` 섹션에 있을 것이고, 실제 인쇄 코드는 `.text` 섹션에 있을 것입니다.
- `.shstrtab`: 이것은 재미있는 구현 세부 사항입니다! 섹션의 이름 자체(`.text` 및 `.shstrtab`과 같은)는 섹션 헤더 테이블에 직접 포함되지 않습니다. 대신, 각 항목은 이름을 포함하는 ELF 파일의 위치에 대한 오프셋을 포함합니다. 이렇게 하면, 섹션 헤더 테이블의 각 항목이 동일한 크기가 될 수 있어 파싱하기가 더 쉬워집니다 — 이름에 대한 오프셋은 고정 크기 숫자인 반면, 테이블에 이름을 포함하면 가변 크기 문자열을 사용하게 됩니다. 이 모든 이름 데이터는 `SHT_STRTAB` 유형의 `.shstrtab`이라는 자체 섹션에 저장됩니다.

데이터

프로그램 및 섹션 헤더 테이블 항목은 모두 메모리에 로드하거나, 프로그램 코드가 어디에 있는지 지정하거나, 단순히 섹션 이름을 지정하기 위해 ELF 파일 내의 데이터 블록을 가리킵니다. 이러한 다양한 데이터 조각은 모두 ELF 파일의 데이터 섹션에 포함됩니다.



링킹에 대한 간략한 설명

`binfmt_elf` 코드로 돌아가서: 커널은 프로그램 헤더 테이블의 두 가지 유형의 항목에 관심을 갖습니다.

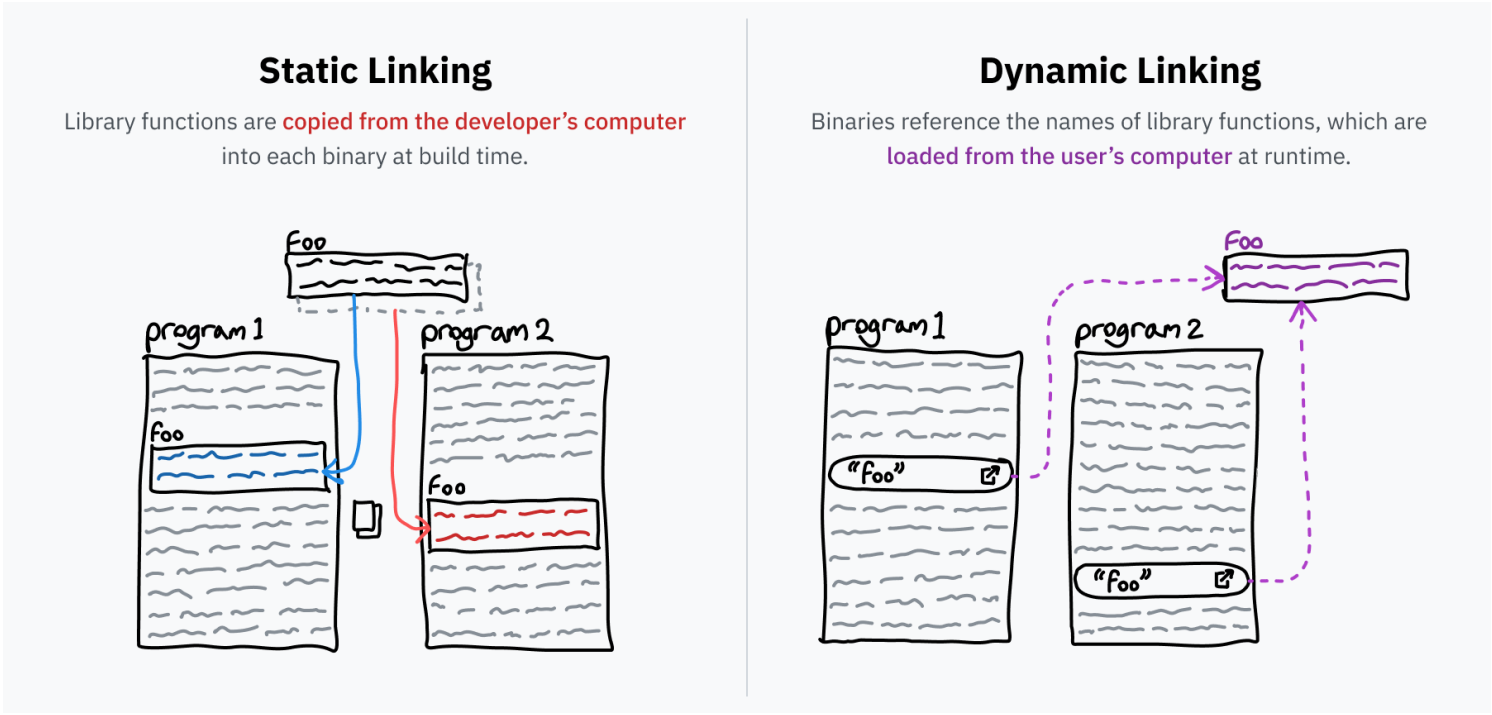
`PT_LOAD` 세그먼트는 `.text` 및 `.data` 섹션과 같은 모든 프로그램 데이터를 메모리에 어디에 로드해야 하는지 지정합니다. 커널은 ELF 파일에서 이러한 항목을 읽어 CPU에서 프로그램이 실행될 수 있도록 데이터를 메모리에 로드합니다.

커널이 관심을 갖는 다른 유형의 프로그램 헤더 테이블 항목은 "동적 링킹 런타임"을 지정하는 `PT_INTERP`입니다.

동적 링킹이 무엇인지 이야기하기 전에, 일반적으로 "링킹"에 대해 이야기합시다. 프로그래머는 재사용 가능한 코드 라이브러리 위에 프로그램을 구축하는 경향이 있습니다 — 예를 들어, 앞서 이야기한 libc입니다. 소스 코드를 실행 가능한 바이너리로 변환할 때, 링커라는 프로그램이 라이브러리 코드를 찾고 바이너리에 복사하여 이러한 모든 참조를 해결합니다. 이 프로세스를 *정적 링킹*이라고 하며, 외부 코드가 배포되는 파일에 직접 포함됨을 의미합니다.

그러나, 일부 라이브러리는 매우 일반적입니다. libc는 기본적으로 태양 아래 모든 프로그램에서 사용되는데, OS와 시스템 콜을 통해 상호 작용하기 위한 표준 인터페이스이기 때문입니다. 컴퓨터의 모든 단일 프로그램에 libc의 별도 복사본을 포함하는 것은 공간의 끔찍한 낭비일 것입니다. 또한, 라이브러리의 버그가 라이브러리를 사용하는 각 프로그램이 업데이트되기를 기다리지 않고 한 곳에서 수정될 수 있다면 좋을 것입니다. 동적 링킹은 이러한 문제에 대한 해결책입니다.

정적으로 링크된 프로그램이 `bar`라는 라이브러리에서 `foo` 함수를 필요로 하는 경우, 프로그램은 `foo`의 전체 복사본을 포함할 것입니다. 그러나 동적으로 링크된 경우 "라이브러리 `bar`에서 `foo`가 필요합니다"라는 참조만 포함할 것입니다. 프로그램이 실행될 때, `bar`가 컴퓨터에 설치되어 있기를 바라며 `foo` 함수의 기계어를 필요에 따라 메모리에 로드할 수 있습니다. 컴퓨터의 `bar` 라이브러리 설치가 업데이트되면, 프로그램 자체의 변경 없이 다음에 프로그램이 실행될 때 새 코드가 로드됩니다.



실제로 사용되는 동적 링킹

Linux에서 `bar`와 같은 동적으로 링크 가능한 라이브러리는 일반적으로 .so (Shared Object) 확장자를 가진 파일로 패키징됩니다. 이러한 .so 파일은 프로그램과 마찬가지로 ELF 파일입니다 — ELF 헤더에 파일이 실행 파일인지 라이브러리인지 지정하는 필드가 포함되어 있음을 기억하실 것입니다. 또한, 공유 객체는 섹션 헤더 테이블에 파일에서 어떤 심볼이 내보내지고 동적으로 링크될 수 있는지에 대한 정보를 포함하는 `.dynsym` 섹션을 가지고 있습니다.

Windows에서는 `bar`와 같은 라이브러리가 .dll (**d**ynamic **l**ink **l**ibrary) 파일로 패키징됩니다. macOS는 .dylib (**dy**namically linked **lib**rary) 확장자를 사용합니다. macOS 앱과 Windows .exe 파일과 마찬가지로, 이것들은 ELF 파일과 약간 다르게 형식화되어 있지만 동일한 개념과 기술입니다.

두 가지 유형의 링킹 사이의 흥미로운 차이점은 정적 링킹을 사용하면 사용되는 라이브러리의 부분만 실행 파일에 포함되어 메모리에 로드된다는 것입니다. 동적 링킹을 사용하면 *전체 라이브러리*가 메모리에 로드됩니다. 이것은 처음에는 덜 효율적으로 들릴 수 있지만, 실제로는 현대 운영 체제가 라이브러리를 메모리에 한 번 로드한 다음 프로세스 간에 해당 코드를 공유함으로써 *더 많은* 공간을 절약할 수 있게 합니다. 라이브러리가 다른 프로그램에 대해 다른 상태를 필요로 하므로 코드만 공유할 수 있지만, 절약은 여전히 수십에서 수백 메가바이트의 RAM 정도일 수 있습니다.

실행

커널이 ELF 파일을 실행하는 것으로 다시 돌아가 봅시다: 실행 중인 바이너리가 동적으로 링크된 경우, OS는 바로 바이너리의 코드로 점프할 수 없습니다. 왜냐하면 누락된 코드가 있기 때문입니다 — 기억하세요, 동적으로 링크된 프로그램은 필요한 라이브러리 함수에 대한 참조만 가지고 있습니다!

바이너리를 실행하기 위해, OS는 어떤 라이브러리가 필요한지 파악하고, 그것들을 로드하고, 모든 이름이 지정된 포인터를 실제 점프 명령어로 교체한 *다음* 실제 프로그램 코드를 시작해야 합니다. 이것은 ELF 형식과 깊이 상호작용하는 매우 복잡한 코드이므로, 일반적으로 커널의 일부가 아니라 독립 실행형 프로그램입니다. ELF 파일은 프로그램 헤더 테이블의 `PT_INTERP` 항목에서 사용하려는 프로그램의 경로를 지정합니다(일반적으로 `/lib64/ld-linux-x86-64.so.2`와 같은 것).

ELF 헤더를 읽고 프로그램 헤더 테이블을 스캔한 후, 커널은 새 프로그램을 위한 메모리 구조를 설정할 수 있습니다. 모든 `PT_LOAD` 세그먼트를 메모리에 로드하여 시작하며, 프로그램의 정적 데이터, BSS 공간 및 기계어를 채웁니다. 프로그램이 동적으로 링크된 경우, 커널은 ELF 인터프리터 (`PT_INTERP`)를 실행해야 하므로, 인터프리터의 데이터, BSS 및 코드도 메모리에 로드합니다.

이제 커널은 사용자 공간으로 복귀할 때 복원할 CPU의 명령어 포인터를 설정해야 합니다. 실행 파일이 동적으로 링크된 경우, 커널은 명령어 포인터를 메모리에서 ELF 인터프리터 코드의 시작으로 설정합니다. 그렇지 않으면, 커널은 실행 파일의 시작으로 설정합니다.

커널은 이제 시스템 콜에서 복귀할 준비가 거의 되었습니다 (`execve`에 있다는 것을 기억하세요). 프로그램이 시작할 때 읽을 수 있도록 `argc`, `argv` 및 환경 변수를 스택에 푸시합니다.

레지스터는 이제 지워집니다. 시스템 콜을 처리하기 전에, 커널은 레지스터의 현재 값을 사용자 공간으로 다시 전환할 때 복원되도록 스택에 저장합니다. 사용자 공간으로 복귀하기 전에, 커널은 스택의 이 부분을 0으로 만듭니다.

마지막으로, 시스템 콜이 끝나고 커널은 사용자 공간으로 복귀합니다. 이제 0이 된 레지스터를 복원하고, 저장된 명령어 포인터로 점프합니다. 그 명령어 포인터는 이제 새 프로그램(또는 ELF 인터프리터)의 시작점이며 현재 프로세스가 교체되었습니다!

챕터 5: 컴퓨터 안의 번역기

지금까지 메모리를 읽고 쓰는 것에 대해 이야기할 때마다 조금 애매했습니다. 예를 들어, ELF 파일은 데이터를 로드할 특정 메모리 주소를 지정하는데, 서로 다른 프로세스가 충돌하는 메모리를 사용하려고 하는 문제가 발생하지 않는 이유는 무엇일까요? 왜 각 프로세스가 다른 메모리 환경을 가지고 있는 것처럼 보일까요?

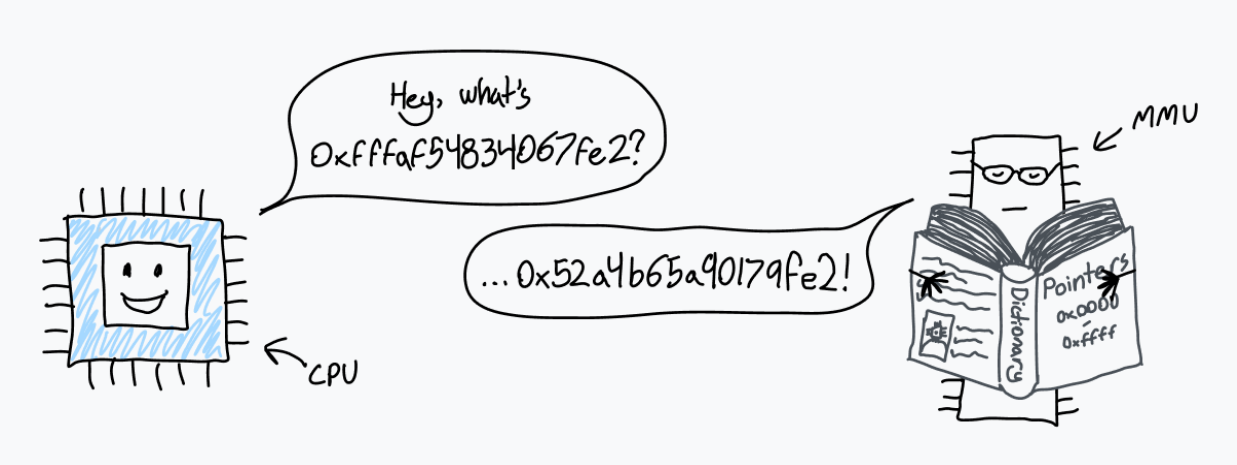
또한, 정확히 어떻게 여기까지 왔을까요? `execve`가 현재 프로세스를 새 프로그램으로 *교체*하는 시스템 콜이라는 것을 이해했지만, 이것은 여러 프로세스를 시작할 수 있는 방법을 설명하지 않습니다. 첫 번째 프로그램이 어떻게 실행되는지도 확실히 설명하지 않습니다 — 어떤 닭(프로세스)이 다른 모든 알(다른 프로세스들)을 낳습니까(생성합니까)?

우리는 여정의 끝에 가까워지고 있습니다. 이 두 가지 질문에 답하고 나면, 컴퓨터가 부팅에서 지금 사용하고 있는 소프트웨어를 실행하기까지 어떻게 되었는지에 대한 거의 완전한 이해를 갖게 될 것입니다.

메모리는 가짜입니다

그래서... 메모리에 대해. CPU가 메모리 주소를 읽거나 쓸 때, 실제로 *물리* 메모리(RAM)의 해당 위치를 참조하는 것이 아니라는 것이 밝혀졌습니다. 오히려, *가상* 메모리 공간의 위치를 가리키고 있습니다.

CPU는 *메모리 관리 장치* (MMU)라는 칩과 통신합니다. MMU는 가상 메모리의 위치를 RAM의 위치로 변환하는 사전을 가진 번역기처럼 작동합니다. CPU가 메모리 주소 `0xfffaf54834067fe2`에서 읽으라는 명령어를 받으면, MMU에 해당 주소를 변환하도록 요청합니다. MMU는 사전에서 조회하고, 일치하는 물리 주소가 `0x53a4b64a90179fe2`임을 발견하고, 숫자를 CPU로 다시 보냅니다. 그러면 CPU는 RAM의 해당 주소에서 읽을 수 있습니다.



컴퓨터가 처음 부팅될 때, 메모리 액세스는 물리 RAM으로 직접 이동합니다. 시작 직후, OS는 변환 사전을 만들고 CPU에 MMU 사용을 시작하도록 지시합니다.

이 사전은 실제로 *페이지 테이블*이라고 불리며, 모든 메모리 액세스를 변환하는 이 시스템을 *페이징*이라고 합니다. 페이지 테이블의 항목은 *페이지*라고 불리며 각각은 가상 메모리의 특정 청크가 RAM에 어떻게 매핑되는지를 나타냅니다. 이러한 청크는 항상 고정된 크기이며, 각 프로세서 아키텍처는 다른 페이지 크기를 가지고 있습니다. x86-64는 기본 4 KiB 페이지 크기를 가지며, 이는 각 페이지가 4,096바이트 길이의 메모리 블록에 대한 매핑을 지정함을 의미합니다.

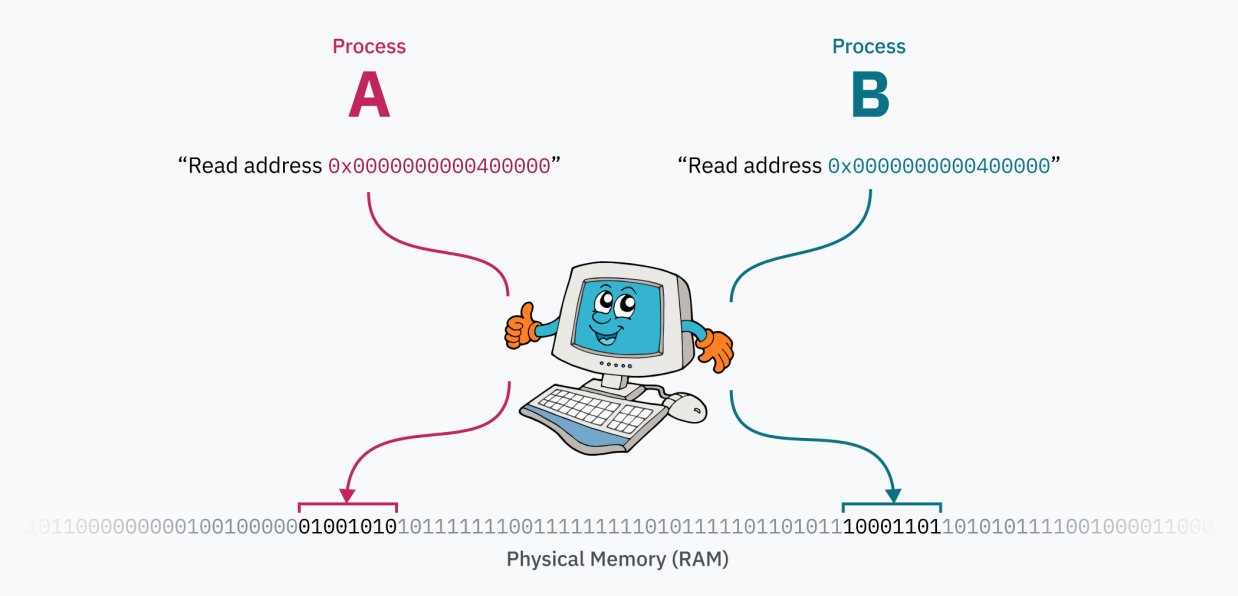
즉, 4 KiB 페이지를 사용하면 주소의 하위 12비트는 MMU 변환 전후에 항상 동일할 것입니다 — 12비트인 이유는 변환 후 얻는 4,096바이트 페이지를 인덱싱하는 데 필요한 비트 수이기 때문입니다.

x86-64는 또한 운영 체제가 더 큰 2 MiB 또는 4 GiB 페이지를 활성화할 수 있게 하며, 이것은 주소 변환 속도를 향상시킬 수 있지만 메모리 단편화와 낭비를 증가시킵니다. 페이지 크기가 클수록, MMU에 의해 변환되는 주소의 일부가 작아집니다.

페이지 테이블 자체는 RAM에 있습니다. 수백만 개의 항목을 포함할 수 있지만, 각 항목의 크기는 몇 바이트 정도에 불과하므로 페이지 테이블은 그다지 많은 공간을 차지하지 않습니다.

부팅 시 페이징을 활성화하기 위해, 커널은 먼저 RAM에 페이지 테이블을 구성합니다. 그런 다음, 페이지 테이블 베이스 레지스터(PTBR)라는 레지스터에 페이지 테이블의 시작 부분의 물리 주소를 저장합니다. 마지막으로, 커널은 MMU로 모든 메모리 액세스를 변환하도록 페이징을 활성화합니다. x86-64에서 제어 레지스터 3(CR3)의 상위 20비트는 PTBR로 기능합니다. 페이징을 위해 지정된 CR0의 비트 31, PG는 페이징을 활성화하기 위해 1로 설정됩니다.

페이징 시스템의 마법은 컴퓨터가 실행되는 동안 페이지 테이블을 편집할 수 있다는 것입니다. 이것이 각 프로세스가 자체 격리된 메모리 공간을 가질 수 있는 방법입니다 — OS가 한 프로세스에서 다른 프로세스로 컨텍스트를 전환할 때, 중요한 작업은 가상 메모리 공간을 물리 메모리의 다른 영역으로 다시 매핑하는 것입니다. 두 프로세스가 있다고 가정해 봅시다: 프로세스 A는 `0x0000000000400000`에 코드와 데이터(ELF 파일에서 로드되었을 가능성이 높음)를 가질 수 있고, 프로세스 B는 정확히 동일한 주소에서 코드와 데이터에 액세스할 수 있습니다. 이 두 프로세스는 실제로 해당 주소 범위를 두고 싸우지 않기 때문에 동일한 프로그램의 인스턴스일 수도 있습니다! 프로세스 A의 데이터는 물리 메모리에서 프로세스 B로부터 멀리 떨어진 곳에 있으며, 프로세스로 전환할 때 커널에 의해 `0x0000000000400000`에 매핑됩니다.



> **참고: 저주받은 ELF 사실**
>
> 특정 상황에서, `binfmt_elf`는 메모리의 첫 번째 페이지를 0으로 매핑해야 합니다. 1988년의 OS이자 ELF를 지원한 최초의 OS인 UNIX System V Release 4.0 (SVr4)용으로 작성된 일부 프로그램은 널 포인터를 읽을 수 있다는 것에 의존합니다. 그리고 어떻게든, 일부 프로그램은 여전히 그 동작에 의존합니다.
>
> 이것을 구현하는 Linux 커널 개발자가 조금 불만스러웠던 것 같습니다:
>
> *“왜 이것을, 물어보시나요??? 글쎄요 SVr4는 페이지 0을 읽기 전용으로 매핑하고, 일부 애플리케이션은 이 동작에 ‘의존’합니다. 이것들을 다시 컴파일할 수 있는 권한이 없기 때문에, 우리는 SVr4 동작을 에뮬레이트 합니다. 한숨.”*
>
> 한숨.

페이징을 통한 보안

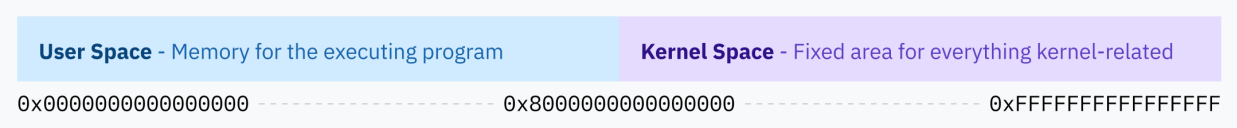
메모리 페이징에 의해 활성화된 프로세스 격리는 코드 인체공학을 개선하지만(프로세스가 메모리를 사용하기 위해 다른 프로세스를 인식할 필요가 없음), 보안 수준도 만듭니다: 프로세스는 다른 프로세스의 메모리에 액세스할 수 없습니다. 이것은 이 글의 시작 부분에서 원래 질문 중 하나에 절반 정도 답합니다:

> 프로그램이 CPU에서 직접 실행되고 CPU가 RAM에 직접 액세스할 수 있다면, 왜 코드가 다른 프로세스의 메모리나, 더 나쁘게는 커널에 액세스할 수 없을까요?

그것을 기억하시나요? 정말 오래전 같네요...

그런데 그 커널 메모리는 어떨까요? 우선: 커널은 명백히 실행 중인 모든 프로세스를 추적하고 심지어 페이지 테이블 자체를 추적하기 위해 자체 데이터를 많이 저장해야 합니다. 하드웨어 인터럽트, 소프트웨어 인터럽트 또는 시스템 콜이 트리거되고 CPU가 커널 모드로 진입할 때마다, 커널 코드는 어떻게든 그 메모리에 액세스해야 합니다.

Linux의 해결책은 항상 가상 메모리 공간의 상위 절반을 커널에 할당하는 것이므로 Linux는 *상위 절반 커널*이라고 불립니다. Windows는 유사한 기술을 사용하는 반면, macOS는... 약간 더 복잡하여 그것에 대해 읽으면서 제 뇌가 귀 밖으로 흘러나왔습니다. ~(++)~



그러나 사용자 공간 프로세스가 커널 메모리를 읽거나 쓸 수 있다면 보안에 끔찍할 것이므로, 페이징은 두 번째 보안 계층을 활성화합니다: 각 페이지는 권한 플래그를 지정해야 합니다. 한 플래그는 영역이 쓰기 가능한지 또는 읽기 전용인지 결정합니다. 다른 플래그는 CPU에 커널 모드만 영역의 메모리에 액세스할 수 있도록 허용한다고 알립니다. 이 후자의 플래그는 전체 상위 절반 커널 공간을 보호하는 데 사용됩니다 — 전체 커널 메모리 공간은 실제로 사용자 공간 프로그램의 가상 메모리 매핑에서 사용 가능하지만, 액세스 권한이 없습니다.

Page Table Entry

Present:	true
Read/write:	read only
User/kernel:	all modes
Dirty:	false
Accessed:	true

etc.

페이지 테이블 자체는 실제로 커널 메모리 공간 내에 포함되어 있습니다! 타이머 칩이 프로세스 전환을 위한 하드웨어 인터럽트를 트리거하면, CPU는 권한 수준을 커널 모드로 전환하고 Linux 커널 코드로 점프합니다. 커널 모드(Intel 링 0)에 있으면 CPU가 커널로 보호된 메모리 영역에 액세스할 수 있습니다. 그러면 커널은 새 프로세스를 위해 가상 메모리의 하위 절반을 다시 매핑하기 위해 페이지 테이블(메모리의 상위 절반 어딘가에 있음)에 쓸 수 있습니다. 커널이 새 프로세스로 전환하고 CPU가 사용자 모드로 진입하면, 더 이상 커널 메모리에 액세스할 수 없습니다.

거의 모든 메모리 액세스는 MMU를 거칩니다. 인터럽트 디스크립터 테이블 핸들러 포인터? 그것들도 커널의 가상 메모리 공간을 주소로 지정합니다.

계층적 페이징 및 기타 최적화

64비트 시스템은 64비트 길이의 메모리 주소를 가지므로, 64비트 가상 메모리 공간은 무려 16 엑스비바이트의 크기입니다. 이것은 엄청나게 크며, 오늘날 존재하는 컴퓨터나 곧 존재할 컴퓨터보다 훨씬 큽니다. 제가 아는 한, 지금까지 컴퓨터에서 가장 많은 RAM은 Blue Waters 슈퍼컴퓨터에 있었으며, 1.5페타바이트 이상의 RAM을 가지고 있습니다. 그것은 여전히 16 EiB의 0.01% 미만입니다.

가상 메모리 공간의 모든 4 KiB 섹션에 대해 페이지 테이블의 항목이 필요하다면, 4,503,599,627,370,496 개의 페이지 테이블 항목이 필요할 것입니다. 8바이트 길이의 페이지 테이블 항목을 사용하면, 페이지 테이블만 저장하는 데 32페비바이트의 RAM이 필요할 것입니다. 이것이 여전히 컴퓨터에서 가장 많은 RAM의 세계 기록보다 크다는 것을 알 수 있습니다.

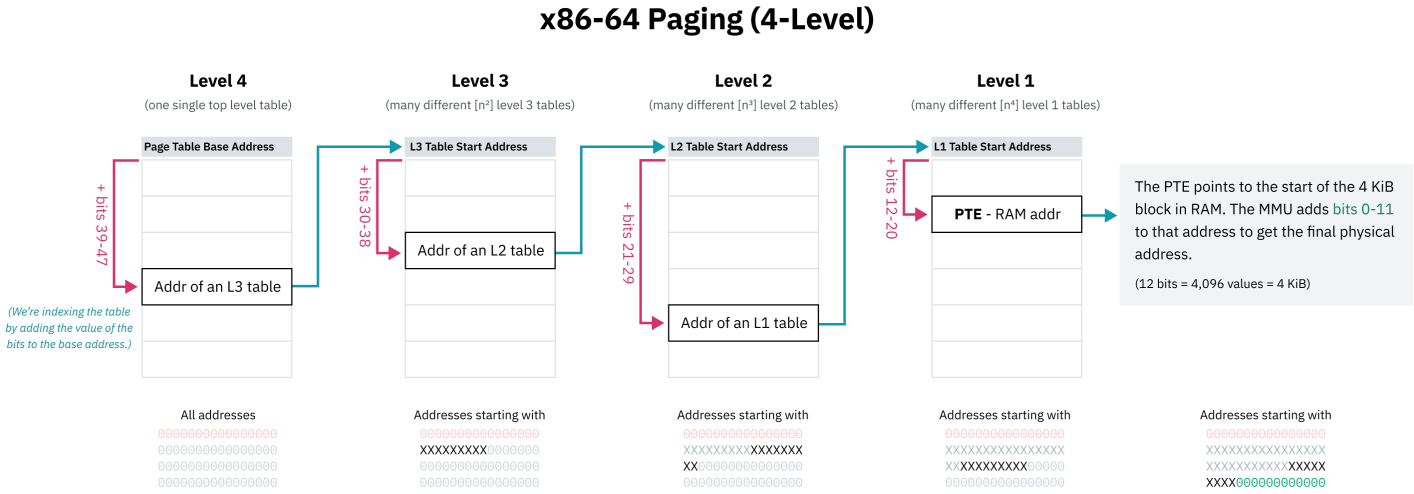
> **참고: 왜 이상한 단위를?**
>
> 흔하지 않고 정말 보기 흉하다는 것을 알고 있지만, 이진 바이트 크기 단위(2의 거듭제곱)와 메트릭 단위(10의 거듭제곱)를 명확하게 구분하는 것이 중요하다고 생각합니다. 킬로바이트, kB는 1,000바이트를 의미하는 SI 단위입니다. 키비바이트, KiB는 1,024바이트를 의미하는 IEC 권장 단위입니다. CPU와 메모리 주소의 관점에서, 바이트 수는 일반적으로 2의 거듭제곱입니다. 왜냐하면 컴퓨터는 이진 시스템이기 때문입니다. KB (또는 더 나쁘게는 kB)를 1,024를 의미하는 데 사용하면 더 모호할 것입니다.

전체 가능한 가상 메모리 공간에 대해 순차 페이지 테이블 항목을 갖는 것은 불가능(또는 적어도 엄청나게 비실용적)하므로, CPU 아키텍처는 *계층적 페이징*을 구현합니다. 계층적 페이징 시스템에서는 점점 더 작은 세분성의 여러 수준의 페이지 테이블이 있습니다. 최상위 수준 항목은 큰 메모리 블록을 다루고 더 작은 블록의 페이지 테이블을 가리키며, 트리 구조를 만듭니다. 4 KiB 또는 페이지 크기가 무엇이든 간에 블록에 대한 개별 항목은 트리의 리프입니다.

x86-64는 역사적으로 4수준 계층적 페이징을 사용합니다. 이 시스템에서, 각 페이지 테이블 항목은 포함하는 테이블의 시작 부분에서 주소의 일부만큼 오프셋하여 찾습니다. 이 부분은 최상위 비트로 시작하며, 접두사로 작동하므로 항목은 해당 비트로 시작하는 모든 주소를 다룹니다. 항목은 해당 메모리 블록에 대한 하위 트리를 포함하는 다음 수준 테이블의 시작을 가리키며, 이것은 다음 비트 모음으로 다시 인덱싱됩니다.

x86-64의 4수준 페이징 설계자는 페이지 테이블 공간을 절약하기 위해 모든 가상 포인터의 상위 16비트를 무시하기로 선택했습니다. 48비트는 128 TiB 가상 주소 공간을 제공하며, 이것은 충분히 크다고 여겨졌습니다. (전체 64비트는 16 EiB를 제공하는데, 이것은 꽤 많습니다.)

첫 16비트는 건너뛰므로, 첫 번째 수준 페이지 테이블을 인덱싱하기 위한 "최상위 비트"는 실제로 63이 아니라 비트 47에서 시작합니다. 이것은 또한 이 장 앞부분의 상위 절반 커널 다이어그램이 기술적으로 부정확했다는 것을 의미합니다; 커널 공간 시작 주소는 64비트보다 작은 주소 공간의 중간점으로 묘사되었어야 했습니다.



계층적 페이징은 트리의 모든 수준에서 다음 항목에 대한 포인터가 null(`0x0`)일 수 있기 때문에 공간 문제를 해결합니다. 이것은 페이지 테이블의 전체 하위 트리를 생략할 수 있게 하며, 가상 메모리 공간의 매핑되지 않은 영역은 RAM에서 공간을 차지하지 않습니다. 매핑되지 않은 메모리 주소의 조회는 CPU가 트리에서 더 높은 곳에서 빈 항목을 보는 즉시 오류를 발생시킬 수 있기 때문에 빠르게 실패할 수 있습니다. 페이지 테이블 항목에는 주소가 유효해 보이더라도 사용할 수 없는 것으로 표시하는 데 사용할 수 있는 존재 플래그도 있습니다.

계층적 페이징의 또 다른 이점은 가상 메모리 공간의 큰 섹션을 효율적으로 교체할 수 있는 능력입니다. 큰 가상 메모리 덩어리는 한 프로세스에 대해 물리 메모리의 한 영역에 매핑될 수 있고, 다른 프로세스에 대해 다른 영역에 매핑될 수 있습니다. 커널은 두 매핑을 모두 메모리에 저장하고 프로세스를 전환할 때 트리의 최상위 수준에서 포인터를 업데이트하기만 하면 됩니다. 전체 메모리 공간 매핑이 항목의 평면 배열로 저장되면, 커널은 많은 항목을 업데이트해야 하므로 느리고 여전히 각 프로세스에 대한 메모리 매핑을 독립적으로 추적해야 합니다.

저는 x86-64가 "역사적으로" 4수준 페이징을 사용한다고 말했는데, 최근 프로세서는 5수준 페이징을 구현하기 때문입니다. 5수준 페이징은 57비트 주소로 주소 공간을 128 PiB로 확장하기 위해 또 다른 간접 계층과 9개의 주소 지정 비트를 추가합니다. 5수준 페이징은 2017년 이후 Linux와 최근 Windows 10 및 11 서버 버전을 포함한 운영 체제에서 지원됩니다.

> **참고: 물리 주소 공간 제한**
>
> 운영 체제가 가상 주소에 모든 64비트를 사용하지 않는 것처럼, 프로세서는 전체 64비트 물리 주소를 사용하지 않습니다. 4수준 페이징이 표준이었을 때, x86-64 CPU는 46비트 이상을 사용하지 않았으며, 물리 주소 공간이 64 TiB로만 제한되었음을 의미합니다. 5수준 페이징을 사용하면, 지원이 52비트로 확장되어 4 PiB 물리 주소 공간을 지원합니다.
>
> OS 수준에서, 가상 주소 공간이 물리 주소 공간보다 큰 것이 유리합니다. Linus Torvalds가 말했듯이, "[i]t needs to be bigger, by a factor of *at least* two, and that's quite frankly pushing it, and you're much better off having a factor of ten or more. Anybody who doesn't get that is a moron. End of discussion."

스와핑과 요구 페이징

메모리 액세스는 몇 가지 이유로 실패할 수 있습니다: 주소가 범위를 벗어날 수 있거나, 페이지 테이블에 매핑되지 않을 수 있거나, 존재하지 않는 것으로 표시된 항목이 있을 수 있습니다. 이러한 경우 중 어느 것이든, MMU는 커널이 문제를 처리할 수 있도록 *페이지 폴트*라는 하드웨어 인터럽트를 트리거합니다.

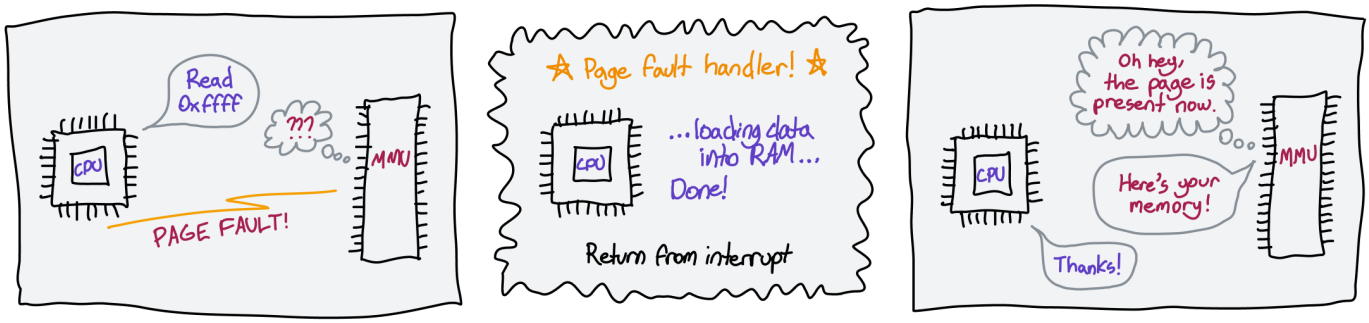
일부 경우에, 읽기가 진정으로 유효하지 않거나 금지되었습니다. 이러한 경우, 커널은 아마도 segmentation fault 오류와 함께 프로그램을 종료할 것입니다.

Shell session

```
$ ./program
Segmentation fault (core dumped)
$
```

> **참고: segfault 존재론**
>
> "Segmentation fault"는 다른 컨텍스트에서 다른 것을 의미합니다. MMU는 권한 없이 메모리를 읽을 때 "segmentation fault"라는 하드웨어 인터럽트를 트리거하지만, "segmentation fault"는 또한 OS가 불법 메모리 액세스로 인해 실행 중인 프로그램을 종료하기 위해 보낼 수 있는 신호의 이름이기도 합니다.

다른 경우에, 메모리 액세스는 *의도적으로* 실패할 수 있으며, OS가 메모리를 채운 다음 *다시 시도하기 위해 CPU에 제어를 돌려줍니다*. 예를 들어, OS는 실제로 RAM에 로드하지 않고 디스크의 파일을 가상 메모리에 매핑할 수 있으며, 주소가 요청되고 페이지 폴트가 발생할 때 물리 메모리에 로드합니다. 이것을 *요구 페이징*이라고 합니다.



우선, 이것은 전체 파일을 디스크에서 가상 메모리로 느리게 매핑하는 mmap과 같은 시스템 콜이 존재할 수 있게 합니다. LLaMa.cpp에 익숙하다면, 유출된 Facebook 언어 모델을 위한 런타임인데, Justine Tunney가 최근 모든 로딩 로직이 mmap을 사용하도록 만들어 크게 최적화했습니다. (그녀에 대해 들어본 적이 없다면, 그녀의 것들을 확인해보세요! Cosmopolitan Libc와 APE는 정말 멋지고 이 글을 즐기고 있다면 흥미로울 수 있습니다.)

> *분명히 Justine의 이 변경에 대한 참여에 대해 많은 드라마 가 있습니다. 랜덤 인터넷 사용자에게 소리지르지 않도록 이것을 지적하는 것입니다. 저는 대부분의 드라마를 읽지 않았으며, Justine의 것이 멋지다고 말한 모든 것은 여전히 매우 사실입니다.*

프로그램과 라이브러리를 실행할 때, 커널은 실제로 메모리에 아무것도 로드하지 않습니다. 파일의 mmap만 생성합니다 — CPU가 코드를 실행하려고 시도하면, 페이지가 즉시 폴트하고 커널은 페이지를 실제 메모리 블록으로 교체합니다.

요구 페이징은 또한 "스와핑" 또는 "페이징"이라는 이름으로 본 적이 있을 기술을 가능하게 합니다. 운영 체제는 메모리 페이지를 디스크에 쓴 다음 물리 메모리에서 제거하지만 존재 플래그를 0으로 설정하여 가상 메모리에 유지함으로써 물리 메모리를 해제할 수 있습니다. 해당 가상 메모리가 읽히면, OS는 디스크에서 RAM으로 메모리를 복원하고 존재 플래그를 다시 1로 설정할 수 있습니다. OS는 디스크에서 로드되는 메모리를 위한 공간을 만들기 위해 RAM의 다른 섹션을 스왑해야 할 수 있습니다. 디스크 읽기 및 쓰기는 느리므로, 운영 체제는 효율적인 페이지 교체 알고리즘으로 스와핑이 가능한 한 적게 발생하도록 노력합니다.

흥미로운 해킹은 페이지 테이블 물리 메모리 포인터를 사용하여 물리 저장소 내의 파일 위치를 저장하는 것입니다. MMU가 음수 존재 플래그를 보는 즉시 페이지 폴트하므로, 그것들이 유효하지 않은 메모리 주소라는 것은 중요하지 않습니다. 이것은 모든 경우에 실용적이지는 않지만, 생각하면 재미있습니다.

챕터 6: Fork와 Cow에 대해 이야기해봅시다

마지막 질문: 우리는 어떻게 여기까지 왔을까요? 첫 번째 프로세스는 어디에서 왔을까요?

이 글은 거의 끝났습니다. 우리는 마지막 단계에 있습니다. 홈런을 칠 준비가 되었습니다. 더 푸른 목장으로 이동하고 있습니다. 그리고 당신이 15,000단어짜리 CPU 아키텍처에 관한 글을 읽지 않을 때 하는 잔디 만지기나 다른 것으로부터 *6장의 길이* 거리에 있다는 것을 의미하는 다양한 다른 끔찍한 관용구들.

`execve`가 현재 프로세스를 교체하여 새 프로그램을 시작한다면, 별도의 새 프로세스에서 새 프로그램을 어떻게 시작할까요? 컴퓨터에서 여러 가지 일을 하고 싶다면 이것은 꽤 중요한 능력입니다; 앱을 시작하기 위해 더블 클릭하면, 앱이 별도로 열리면서 이전에 사용하던 프로그램이 계속 실행됩니다.

답은 또 다른 시스템 콜입니다: 모든 멀티프로세싱의 기본이 되는 시스템 콜인 `fork`입니다. `fork`는 실제로 꽤 간단합니다 — 현재 프로세스와 메모리를 복제하고, 저장된 명령어 포인터를 정확히 그 위치에 두고, 두 프로세스가 평소대로 진행하도록 합니다. 개입 없이, 프로그램은 서로 독립적으로 계속 실행되며 모든 계산이 두 배가 됩니다.

새로 실행되는 프로세스는 "자식"이라고 하며, 원래 `fork`를 호출한 프로세스는 "부모"입니다. 프로세스는 `fork`를 여러 번 호출할 수 있으므로 여러 자식을 가질 수 있습니다. 각 자식은 *프로세스 ID* (PID)로 번호가 매겨지며, 1부터 시작합니다.

동일한 코드를 무작정 두 배로 만드는 것은 꽤 쓸모가 없으므로, `fork`는 부모와 자식에서 다른 값을 반환합니다. 부모에서는 새 자식 프로세스의 PID를 반환하고, 자식에서는 0을 반환합니다. 이것은 새 프로세스에서 다른 작업을 수행하는 것을 가능하게 하여 포크하는 것이 실제로 유용하게 만듭니다.

main.c

```
pid_t pid = fork();

// 코드는 평소처럼 이 지점에서 계속되지만, 이제
// 두 개의 "동일한" 프로세스에 걸쳐 있습니다.
//
// 동일합니다... fork에서 반환된 PID를 제외하고!
//
// 이것은 어느 프로그램에게나 그들이 유일하지 않다는
// 유일한 지표입니다.

if (pid == 0) {
        // 우리는 자식에 있습니다.
        // 일부 계산을 수행하고 부모에게 결과를 제공합니다!
} else {
        // 우리는 부모에 있습니다.
        // 아마도 전에 하던 일을 계속합니다.
}
```

프로세스 포크는 머리를 감싸기가 조금 어려울 수 있습니다. 이 시점부터 당신이 그것을 파악했다고 가정하겠습니다; 그렇지 않다면, 꽤 좋은 설명을 위해 이 보기 흉한 웹사이트를 확인하세요.

어쨌든, Unix 프로그램은 `fork`를 호출한 다음 자식 프로세스에서 즉시 `execve`를 실행하여 새 프로그램을 시작합니다. 이것을 *fork-exec 패턴*이라고 합니다. 프로그램을 실행하면, 컴퓨터는 다음과 유사한 코드를 실행합니다:

launcher.c

```
pid_t pid = fork();

if (pid == 0) {
        // 자식 프로세스를 새 프로그램으로 즉시 교체합니다.
        execve(...);
}

// 여기까지 왔으므로, 프로세스가 교체되지 않았습니다. 우리는 부모에 있습니다!
// 유용하게도, 우리는 이제 PID 변수에 새 자식 프로세스의 PID도 가지고 있으며,
// 죽여야 할 경우를 대비해서입니다.

// 부모 프로그램은 여기서 계속됩니다...
```

음메!

프로세스의 메모리를 복제한 다음 다른 프로그램을 로드할 때 모두 버리는 것은 조금 비효율적으로 들릴 수 있습니다. 다행히, 우리에게는 MMU가 있습니다. 물리 메모리에서 데이터를 복제하는 것이 느린 부분이지, 페이지 테이블을 복제하는 것이 아니므로, 우리는 단순히 RAM을 복제하지 *않습니다*. 새 프로세스를 위해 이전 프로세스의 페이지 테이블의 복사본을 만들고 매핑을 동일한 기본 물리 메모리를 가리키도록 유지합니다.

하지만 자식 프로세스는 부모로부터 독립적이고 격리되어 있어야 합니다! 자식이 부모의 메모리에 쓰거나 그 반대의 경우는 괜찮지 않습니다!

COW (copy on write) 페이지를 소개합니다. COW 페이지를 사용하면, 두 프로세스가 메모리에 쓰려고 시도하지 않는 한 동일한 물리 주소에서 읽습니다. 그들 중 하나가 메모리에 쓰려고 하는 즉시, 해당 페이지가 RAM에 복사됩니다. COW 페이지는 전체 메모리 공간을 복제하는 초기 비용 없이 두 프로세스가 메모리 격리를 가질 수 있게 합니다. 이것이 fork-exec 패턴이 효율적인 이유입니다; 이전 프로세스의 메모리 중 어느 것도 새 바이너리를 로드하기 전에 쓰여지지 않으므로, 메모리 복사가 필요하지 않습니다.

COW는 많은 재미있는 것들과 마찬가지로 페이징 해킹과 하드웨어 인터럽트 처리로 구현됩니다. `fork`가 부모를 복제한 후, 두 프로세스의 모든 페이지를 읽기 전용으로 플래그를 지정합니다. 프로그램이 메모리에 쓸 때, 메모리가 읽기 전용이므로 쓰기가 실패합니다. 이것은 커널에 의해 처리되는 segfault (하드웨어 인터럽트 종류)를 트리거합니다. 커널은 메모리를 복제하고, 페이지를 쓰기를 허용하도록 업데이트한 다음, 쓰기를 재시도하기 위해 인터럽트에서 복귀합니다.

> *A: 똑똑! B: 누구세요? A: 인터럽팅 소. B: 인터럽팅 소 — A:* ***음메!***

태초에 (창세기 1:1이 아님)

컴퓨터의 모든 프로세스는 하나를 제외하고 부모 프로그램에 의해 fork-exec되었습니다: *init 프로세스*. init 프로세스는 커널에 의해 직접 수동으로 설정됩니다. 이것은 실행될 첫 번째 사용자 공간 프로그램이며 종료 시 마지막으로 죽습니다.

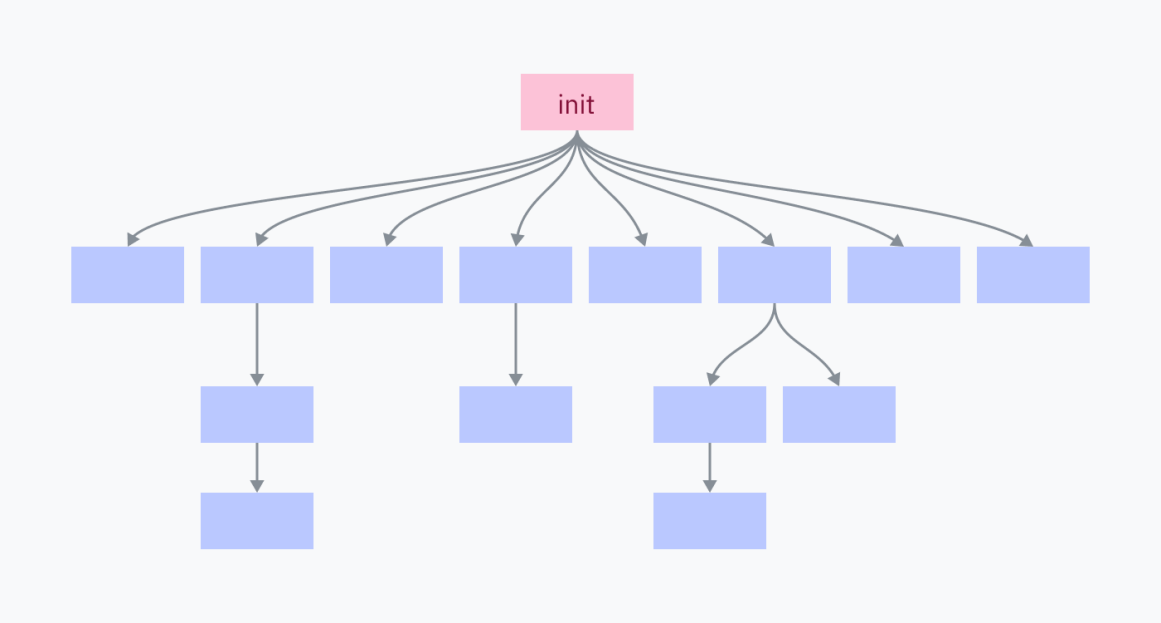
멋진 즉석 블랙스크린을 보고 싶으신가요? macOS나 Linux를 사용 중이라면, 작업을 저장하고 터미널을 열어 init 프로세스 (PID 1)를 종료하세요:

Shell session

```
$ sudo kill 1
```

> *저자의 메모: init 프로세스에 대한 지식은 불행히도 macOS 및 Linux와 같은 Unix 계열 시스템에만 적용됩니다. 지금부터 배우는 대부분은 매우 다른 커널 아키텍처를 가진 Windows를 이해하는 데 적용되지 않습니다.*
>
> *execve에 대한 섹션과 마찬가지로, 저는 이것을 명시적으로 언급하고 있습니다 — NT 커널에 대해 완전히 다른 글을 쓸 수 있지만, 그렇게 하지 않도록 자제하고 있습니다. (지금은.)*

init 프로세스는 운영 체제를 구성하는 모든 프로그램과 서비스를 생성하는 책임이 있습니다. 그들 중 많은 것들은 차례로 자체 서비스와 프로그램을 생성합니다.



init 프로세스를 종료하면 모든 자식과 그들의 모든 자식이 종료되어 OS 환경이 종료됩니다.

커널로 돌아가기

우리는 **[3장]**에서 Linux 커널 코드를 보면서 많은 재미를 느꼈으므로, 조금 더 해봅시다! 이번에는 커널이 init 프로세스를 시작하는 방법을 살펴보겠습니다.

컴퓨터는 다음과 같은 순서로 부팅됩니다:

1. 마더보드는 연결된 디스크에서 *부트로더*라는 프로그램을 검색하는 작은 소프트웨어와 함께 번들로 제공됩니다. 부트로더를 선택하고, 그 기계어를 RAM에 로드하고, 실행합니다.

   우리는 아직 실행 중인 OS의 세계에 있지 않다는 것을 명심하세요. OS 커널이 init 프로세스를 시작할 때까지, 멀티프로세싱과 시스템 콜은 실제로 존재하지 않습니다. init 전 컨텍스트에서 프로그램을 "실행"한

다는 것은 복귀 예상 없이 RAM의 기계어로 직접 점프하는 것을 의미합니다.

2. 부트로더는 커널을 찾아 RAM에 로드하고 실행하는 책임이 있습니다. 일부 부트로더는 GRUB와 같이 구성 가능하거나 여러 운영 체제 중에서 선택할 수 있게 합니다. BootX와 Windows Boot Manager는 각각 macOS 및 Windows의 내장 부트로더입니다.

3. 커널이 이제 실행 중이며 인터럽트 핸들러 설정, 드라이버 로드, 초기 메모리 매핑 생성을 포함한 대규모 초기화 작업 루틴을 시작합니다. 마지막으로, 커널은 권한 수준을 사용자 모드로 전환하고 init 프로그램을 시작합니다.

4. 우리는 마침내 운영 체제의 사용자 공간에 있습니다! init 프로그램은 init 스크립트를 실행하고, 서비스를 시작하고, 셸/UI와 같은 프로그램을 실행하기 시작합니다.

Linux 초기화

Linux에서 3단계(커널 초기화)의 대부분은 init/main.c의 `start_kernel` 함수에서 발생합니다. 이 함수는 200줄이 넘는 다양한 다른 init 함수에 대한 호출이므로, 전체를 이 글에 포함하지는 않겠지만, 훑어보는 것을 권장합니다! `start_kernel`의 끝에서 `arch_call_rest_init`이라는 함수가 호출됩니다:

start_kernel @ init/main.c

```
1087          /* Do the rest non-__init'ed, we're now alive */
1088          arch_call_rest_init();
```

> **non-__init'ed는 무슨 뜻인가요?**
>
> `start_kernel` 함수는 `asmlinkage __visible void __init __no_sanitize_address start_kernel(void)`로 정의됩니다. `__visible`, `__init`, `__no_sanitize_address`와 같은 이상한 키워드는 모두 함수에 다양한 코드나 동작을 추가하기 위해 Linux 커널에서 사용되는 C 전처리기 매크로입니다.
>
> 이 경우, `__init`는 부팅 프로세스가 완료되는 즉시 함수와 데이터를 메모리에서 해제하도록 커널에 지시하는 매크로이며, 단순히 공간을 절약하기 위함입니다.
>
> 어떻게 작동할까요? 너무 깊이 들어가지 않고, Linux 커널 자체는 ELF 파일로 패키징됩니다. `__init` 매크로는 일반적인 `.text` 섹션 대신 `.init.text`라는 섹션에 코드를 배치하는 컴파일러 지시문인 `__section(".init.text")`로 확장됩니다. 다른 매크로는 `__initdata`와 같이 데이터와 상수를 특수 init 섹션에 배치할 수 있게 하며, 이것은 `__section(".init.data")`로 확장됩니다.

arch_call_rest_init는 단순한 래퍼 함수입니다:

init/main.c

```
832  void __init __weak arch_call_rest_init(void)
833  {
834          rest_init();
835  }
```

주석에서 "do the rest non-__init'ed"라고 했던 이유는 rest_init이 __init 매크로로 정의되지 않았기 때문입니다. 이것은 init 메모리를 정리할 때 해제되지 않는다는 것을 의미합니다:

init/main.c

```
689  noinline void __ref rest_init(void)
690  {
```

rest_init은 이제 init 프로세스를 위한 스레드를 생성합니다:

rest_init @ init/main.c

```
695          /*
696           * We need to spawn init first so that it obtains pid 1, however
697           * the init task will end up wanting to create kthreads, which, if
698           * we schedule it before we create kthreadd, will OOPS.
699           */
700          pid = user_mode_thread(kernel_init, NULL, CLONE_FS);
```

user_mode_thread에 전달된 kernel_init 매개변수는 일부 초기화 작업을 마치고 실행할 유효한 init 프로그램을 검색하는 함수입니다. 이 절차는 일부 기본 설정 작업으로 시작합니다; 저는 대부분 이것들을 건너뛸 것이며, free_initmem이 호출되는 곳을 제외하고는 말입니다. 이것이 커널이 우리의 .init 섹션을 해제하는 곳입니다!

kernel_init @ init/main.c

```
1471          free_initmem();
```

이제 커널은 실행할 적합한 init 프로그램을 찾을 수 있습니다:

kernel_init @ init/main.c

```
1495          /*
1496           * We try each of these until one succeeds.
1497           *
1498           * The Bourne shell can be used instead of init if we are
1499           * trying to recover a really broken machine.
1500           */
1501          if (execute_command) {
1502                  ret = run_init_process(execute_command);
1503                  if (!ret)
1504                          return 0;
1505                  panic("Requested init %s failed (error %d).",
1506                        execute_command, ret);
1507          }
1508
1509          if (CONFIG_DEFAULT_INIT[0] != '\0') {
1510                  ret = run_init_process(CONFIG_DEFAULT_INIT);
1511                  if (ret)
1512                          pr_err("Default init %s failed (error %d)\n",
1513                                 CONFIG_DEFAULT_INIT, ret);
1514                  else
1515                          return 0;
1516          }
1517
1518          if (!try_to_run_init_process("/sbin/init") ||
1519              !try_to_run_init_process("/etc/init") ||
1520              !try_to_run_init_process("/bin/init") ||
1521              !try_to_run_init_process("/bin/sh"))
1522                  return 0;
1523
1524          panic("No working init found.  Try passing init= option to kernel
1525                "See Linux Documentation/admin-guide/init.rst for guidance.
```

Linux에서 init 프로그램은 거의 항상 `/sbin/init`에 있거나 심볼릭 링크되어 있습니다. 일반적인 init에는 systemd (비정상적으로 좋은 웹사이트를 가지고 있습니다), OpenRC, 및 runit이 포함됩니다. `kernel_init`은 다른 것을 찾을 수 없으면 `/bin/sh`로 기본값을 설정합니다 — 그리고 `/bin/sh`를 찾을 수 없다면, 뭔가 끔찍하게 잘못되었습니다.

MacOS에도 init 프로그램이 있습니다! 이것은 launchd라고 불리며 `/sbin/launchd`에 있습니다. 터미널에서 그것을 실행하여 커널이 아니라는 소리를 들어보세요.

이 시점부터, 우리는 부팅 프로세스의 4단계에 있습니다: init 프로세스가 사용자 공간에서 실행 중이며 fork-exec 패턴을 사용하여 다양한 프로그램을 시작하기 시작합니다.

Fork 메모리 매핑

Linux 커널이 프로세스를 포크할 때 메모리의 하위 절반을 어떻게 다시 매핑하는지 궁금해서, 조금 살펴봤습니다. kernel/fork.c는 프로세스 포크를 위한 대부분의 코드를 포함하는 것 같습니다. 그 파일의 시작 부분이 유용하게 올바른 위치를 가리켰습니다:

kernel/fork.c

```
 8  /*
 9   *  'fork.c' contains the help-routines for the 'fork' system call
10   * (see also entry.S and others).
11   * Fork is rather simple, once you get the hang of it, but the memory
12   * management can be a bitch. See 'mm/memory.c': 'copy_page_range()'
13   */
```

이 `copy_page_range` 함수는 메모리 매핑에 대한 일부 정보를 받아 페이지 테이블을 복사하는 것처럼 보입니다. 호출하는 함수들을 빠르게 훑어보면, 이것이 또한 페이지를 COW 페이지로 만들기 위해 읽기 전용으로 설정하는 곳입니다. `is_cow_mapping`이라는 함수를 호출하여 이것을 해야 하는지 확인합니다.

`is_cow_mapping`은 include/linux/mm.h에서 정의되며, 메모리 매핑이 메모리가 쓰기 가능하고 프로세스 간에 공유되지 않음을 나타내는 플래그를 가지고 있으면 true를 반환합니다. 공유 메모리는 공유되도록 설계되었기 때문에 COW될 필요가 없습니다. 약간 이해할 수 없는 비트 마스킹을 감상하세요:

include/linux/mm.h

```
1541  static inline bool is_cow_mapping(vm_flags_t flags)
1542  {
1543          return (flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;
1544  }
```

kernel/fork.c로 돌아가서, `copy_page_range`에 대한 간단한 Command-F는 `dup_mmap` 함수에서 하나의 호출을 생성합니다... 이것은 차례로 `dup_mm`에 의해 호출됩니다... 이것은 `copy_mm`에 의해 호출됩니다... 이것은 마침내 대규모 `copy_process` 함수에 의해 호출됩니다! `copy_process`는 fork 함수의 핵심이며, 어떤 면에서 Unix 시스템이 프로그램을 실행하는 방법의 중심점입니다 — 항상 시작 시 첫 번째 프로세스를 위해 생성된 템플릿을 복사하고 편집합니다.

요약하자면...

그래서... 프로그램은 어떻게 실행되나요?

가장 낮은 수준에서: 프로세서는 멍청합니다. 메모리에 포인터를 가지고 있으며 다른 곳으로 점프하라는 명령어에 도달하지 않는 한 연속적으로 명령어를 실행합니다.

점프 명령어 외에도, 하드웨어 및 소프트웨어 인터럽트도 미리 설정된 위치로 점프하여 실행 순서를 깰 수 있으며, 그곳에서 어디로 점프할지 선택할 수 있습니다. 프로세서 코어는 한 번에 여러 프로그램을 실행할 수 없지만, 이것은 타이머를 사용하여 반복적으로 인터럽트를 트리거하고 커널 코드가 다른 코드 포인터 간에 전환할 수 있게 함으로써 시뮬레이션할 수 있습니다.

프로그램은 일관성 있고 격리된 단위로 실행되고 있다고 *속습니다*. 시스템 리소스에 대한 직접 액세스는 사용자 모드에서 방지되고, 메모리 공간은 페이징을 사용하여 격리되며, 시스템 콜은 실제 실행 컨텍스트에 대한 너무 많은 지식 없이 일반적인 I/O 액세스를 허용하도록 설계되었습니다. 시스템 콜은 CPU에 일부 커널 코드를 실행하도록 요청하는 명령어이며, 그 위치는 시작 시 커널에 의해 구성됩니다.

하지만... 프로그램은 어떻게 실행되나요?

컴퓨터가 시작된 후, 커널은 init 프로세스를 시작합니다. 이것은 기계어가 많은 특정 시스템 세부 사항에 대해 걱정할 필요가 없는 더 높은 추상화 수준에서 실행되는 첫 번째 프로그램입니다. init 프로그램은 컴퓨터의 그래픽 환경을 렌더링하고 다른 소프트웨어를 시작하는 책임이 있는 프로그램을 시작합니다.

프로그램을 시작하기 위해, fork 시스템 콜로 자신을 복제합니다. 이 복제는 모든 메모리 페이지가 COW이고 메모리가 물리 RAM 내에서 복사될 필요가 없기 때문에 효율적입니다. Linux에서 이것은 `copy_process` 함수가 작동하는 것입니다.

두 프로세스는 자신이 포크된 프로세스인지 확인합니다. 만약 그렇다면, exec 시스템 콜을 사용하여 커널에 현재 프로세스를 새 프로그램으로 교체하도록 요청합니다.

새 프로그램은 아마도 ELF 파일일 것이며, 커널은 프로그램을 로드하는 방법과 코드와 데이터를 새 가상 메모리 매핑 내에 배치할 위치에 대한 정보를 찾기 위해 파싱합니다. 커널은 또한 프로그램이 동적으로 링크된 경우 ELF 인터프리터를 준비할 수 있습니다.

그런 다음 커널은 프로그램의 가상 메모리 매핑을 로드하고 프로그램이 실행되는 사용자 공간으로 복귀할 수 있으며, 이것은 실제로 CPU의 명령어 포인터를 가상 메모리에서 새 프로그램 코드의 시작으로 설정하는 것을 의미합니다.

챕터 7: 에필로그

축하합니다! 우리는 이제 CPU에 "당신(you)"을 확실히 넣었습니다. 즐거웠기를 바랍니다.

방금 얻은 모든 지식이 실제적이고 활동적이라는 것을 한 번 더 강조하며 보내드립니다. 다음에 컴퓨터가 여러 앱을 실행하는 것에 대해 생각할 때, 타이머 칩과 하드웨어 인터럽트를 상상하기를 바랍니다. 멋진 프로그래밍 언어로 프로그램을 작성하고 링커 오류가 발생하면, 그 링커가 무엇을 하려고 하는지 생각하기를 바랍니다.

이 글에 포함된 내용에 대해 질문이 있거나 (또는 수정 사항이 있다면) lexi@hackclub.com으로 이메일을 보내거나 GitHub에 이슈나 PR을 제출해야 합니다.

... 하지만 잠깐, 더 있습니다!

보너스: C 개념 번역하기

저수준 프로그래밍을 직접 해본 적이 있다면, 스택과 힙이 무엇인지 알고 있을 것이고 `malloc`을 사용해봤을 것입니다. 그것들이 어떻게 구현되는지에 대해 많이 생각하지 않았을 수도 있습니다!

우선, 스레드의 스택은 가상 메모리의 높은 곳에 매핑된 고정된 양의 메모리입니다. 대부분의 (비록 모든 것은 아니지만) 아키텍처에서, 스택 포인터는 스택 메모리의 맨 위에서 시작하여 증가할 때 아래로 이동합니다. 물리 메모리는 전체 매핑된 스택 공간에 대해 미리 할당되지 않습니다; 대신, 요구 페이징이 스택의 프레임에 도달할 때 느리게 메모리를 할당하는 데 사용됩니다.

`malloc`과 같은 힙 할당 함수가 시스템 콜이 아니라는 것을 듣는 것은 놀라울 수 있습니다. 대신, 힙 메모리 관리는 libc 구현에 의해 제공됩니다! `malloc`, `free` 등은 복잡한 절차이며, libc는 메모리 매핑 세부 사항을 스스로

추적합니다. 내부적으로, 사용자 공간 힙 할당자는 `mmap` (파일 이상을 매핑할 수 있음)과 `sbrk`를 포함한 시스템 콜을 사용합니다.

보너스: 잡학

이것들을 일관성 있게 넣을 곳을 찾지 못했지만, 재미있다고 생각했으므로, 여기 있습니다.

> *대부분의 Linux 사용자는 아마도 커널에서 페이지 테이블이 어떻게 표현되는지 상상하는 데 시간을 거의 소비하지 않을 만큼 충분히 흥미로운 삶을 가지고 있을 것입니다.*
>
> *Jonathan Corbet, LWN*

하드웨어 인터럽트의 대체 시각화:

일부 시스템 콜이 커널 공간으로 점프하는 대신 vDSO라는 기술을 사용한다는 메모. 저는 이것에 대해 이야기할 시간이 없었지만, 꽤 흥미로우며 읽어보는 것을 권장합니다.

그리고 마지막으로, Unix 혐의에 대해 언급하자면: 많은 실행 관련 내용이 매우 Unix 특정적이어서 죄송합니다. macOS 또는 Linux 사용자라면 괜찮지만, Windows가 프로그램을 실행하거나 시스템 콜을 처리하는 방법에 훨씬 가까워지지는 않을 것입니다. 비록 CPU 아키텍처 내용은 모두 동일하지만요. 미래에 Windows 세계를 다루는 글을 쓰고 싶습니다.

감사의 말

이 글을 쓰는 동안 GPT-3.5 및 GPT-4와 꽤 많이 대화했습니다. 그들이 많이 거짓말을 했고 대부분의 정보가 쓸모가 없었지만, 때때로 문제를 해결하는 데 매우 도움이 되었습니다. 한계를 인식하고 그들이 말하는 모든 것에 대해 매우 회의적이라면 LLM 지원이 순 긍정적일 수 있습니다. 그렇지만, 그들은 글쓰기에 끔찍합니다. 그들이 당신을 위해 글을 쓰게 하지 마세요.

더 중요한 것은, 교정하고, 격려하고, 브레인스토밍을 도와준 모든 인간들에게 감사합니다 — 특히 Ani, B, Ben, Caleb, Kara, polypixeldev, Pradyun, Spencer, Nicky (**[4장]**에서 멋진 엘프를 그려줌), 그리고 사랑하는 부모님께.

만약 당신이 십대이고 컴퓨터를 좋아하며 아직 Hack Club Slack에 있지 않다면, 지금 바로 가입해야 합니다. 생각과 진행 상황을 공유할 멋진 사람들의 커뮤니티가 없었다면 이 글을 쓰지 않았을 것입니다. 십대가 아니라면, 우리가 멋진 일을 계속할 수 있도록 돈을 주셔야 합니다.

이 글의 모든 평범한 예술은 Figma에서 그렸습니다. 편집에는 Obsidian을 사용했고, 때때로 린팅에는 Vale를 사용했습니다. 이 글의 Markdown 소스는 GitHub에서 사용 가능하며 미래의 nitpick에 열려 있고, 모든 예술은 Figma 커뮤니티 페이지에 게시되어 있습니다.