Program_runtime_view

빌드 시스템을 더욱 잘 이해하기 위해서는 실행 중인 프로그램이 어떻게 동작하는지를 잘 알고 있어야 한다. 프로그램이 메모리에 로드되고 실행되는 방법은 어떤 오브젝트 파일과 실행 파일, 그리고 배포 패키지가 만들어져야 하는 지 결정하게 된다. 프로그램이 순수한 기계어 코드로 변환돼 실행되는지, 아니면 런타임 시에 부분적으로 인터프리트돼 실행되는지, 또한 단일 프로그램으로 구성돼 있는지, 아니면 여러 개의 프로그램으로 구성돼 있는지에 따라 빌드 시스템이 생성해야 하는 결과물은 달라진다.

또한 프로그램의 런타임 뷰는 사용되는 프로그래밍 언어와 런타임 환경을 지원하는 운영체제에 따라 달라진다.

실행프로그램

CPU에서 실행되는 컴퓨터가 이해할 수 있는 명령 집합과 해당 데이터 값이며, 메모리로 로드돼 실행될 수 있는 코드 전체가 컴파일된 프로그램이다.

라이브러리

여러 프로그램에서 공통으로 사용되는 오브젝트 코드의 집합이다. 운영체제 대부분은 재사용 가능한 일단의 표준 라이브러리를 포함하고 있기 때문에 각 프로그램은 고유의 라이브러리를 반드시 갖고 있지 않아도 된다. 라이브러리는 직접 메모리로 로드되거나 실행될 수 없으며, 반드시 실행 프로그램과 링크 된 후 사용된다.

설정 파일과 데이터 파일

실행 가능한 명령으로 구성돼 있지 않다. 유용한 데이터와 설정 정보를 제공하며, 프로그램이 디스크로부터 로드해 사용한다.

실행 프로그램

실행 프로그램은 명령 집합으로 구성되며, 메모리에 로드돼 CPU에 의해 실행된다. 일반적으로 윈도우 환경에서는 아이콘을 더블클릭해서 프로그램을 실행시키거나 command shell에서 프로그램이름으로 실행시킨다. 때로는 컴퓨터가 부팅할 때 시작되거나 스케줄러에 의해 특정 시간에 호출되는 때도 있다.

프로그램이 메모리에 로드되고 난 후 컴파일이 이뤄진 정도와 프로그램 실행에 필요한 부분을 운영체제가 제공하는 정도에 따라 프로그램을 실행시키는 다양한 방법이 존재한다.

네이티브 기계어 코드

빌드 시스템은 실행 프로그램을 CPU에 종속하는 네이티브 기계어 코드로 변환하며, CPU는 프로그램의 시작 주소로 점프해 모든 명령을 실행한다. 프로그램은 실행 중에 운영체제에 파일과 같은 시스템 리소스를 요청하기도 한다.

소프트웨어 시스템에서 프로그램의 실행 속도가 아주 중요하거나 CPU의 기능을 최대한 활용하는 경우 C와 C++ 같은 언어로 네이티브 기계어 코드를 만들어 사용한다. 이는 프로그램의 실행 속도가 가장 빠른 방법이다.

네이티브 기계어 코드 프로그램에서 빌드 시스템은 실행 프로그램 파일을 생성하는데, 이 파일을 일반적으로 프로그램이나 실행 파일, 바이너리라고 부른다.

모놀리식 시스템 이미지

맥 OS X이나 윈도우, 리눅스 같은 운영체제가 탑재된 데스크톱 컴퓨터의 경우 컴퓨터를 사용하기 위한 마우스와 키보드, 그리고 프로그램의 결과를 확인하는 모니터가 보편적으로 사용된다. 하지만 자동차와 텔레비전 그리고 주방기기 등과 같은 장치에 내장된 임베디드 시스템은 아주 작은 운영체제가 실행되거나 심지어 운영체제가 없는 때도 존재한다. 대부분 이와 같은 컴퓨터는 한 번에 하나의 프로그램만 실행된다.

대부분의 임베디드 시스템은 저렴하고 단순하게 만들기 위해 제한된 CPU파워와 메모리를 갖게 되는데, 유닉스나 윈도우 같은 것이 일반적이다. 임베디드 시스템에서는 하나의 프로그램이 전체 메모리를 사용하는 네이티브 기계어 코드 실행 방식을 사용한다. 빌드 시스템 관점에서 주목해야 할 점은 최종 릴리스 패키지가 컴퓨터 메모리에 직접 로드되는 하나의 큰 파일이라는 사실이다. 이런 파일은 컴퓨터에 단독으로 로드되기 대문에 종종 이미지라고 부른다.

전체가 인터프리트되는 프로그램

꽤 많은 프로그래밍 언어가 기계어 코드로 컴파일되지 않는다. 대신 전체 소스코드가 메모리로 로드된 후 인터프리터에 의해 실행된다. 초기 BASIC 언어가 이에 해당하며, 유닉스 shell script는 지금도 이런 방식으로 실행된다.

소스코드가 오브젝트 파일로 컴파일되지는 않지만, 빌드 시스템은 다음과 같이 많은 작업을 수행해야 한다. 소스코드를 타겟 머신에 설치할 수 있게 릴리스 패키지를 생성해야 할 뿐만 아니라 unit test 생성, static analysis 수행, 그리고 문서 생성은 빌드 시스템이 수행해야 하는 일반적인 작업이다. 마지막으로 일부 인터프리트형 언어는 컴파일 언어와 통합해 컴파일된 코드와 인터프리트될 코드가 결합한 하이브리드 소프트웨어를 만들기도 한다.

인터프리트된 바이트 코드

바이트 코드는 네이티브 기계어 코드와 유사하지만 CPU에서 바로 실행될 수 없으며, 네이티브 기계어 코드로 변환되거나 인터프리터를 거쳐 실행된다. 그러므로 바이트 코드를 실행하려면 그 외의 인터프리터이나 컴파일러가 메모리에 로드돼야 한다.

예를 들면 자바가 플랫폼 독립적인 이유는 빌드 시스템에서 CPU에 종속하는 네이티브 기계어 코드를 생성하는 대신, 자바 컴파일러가 머신 독립적인 바이트 코드를 생성하기 때문이다. 그래서 컴파일된 자바 프로그램은 자바 가상머신과 함께 실행이된다. 가상머신의 동작 방식은 두 가지가 있는데, 바이트 코드를 인터프리트해 프로그램이 실행되는 것 처럼 동작하는 방식과 바이트 코드를 네이티브 기계어 코드로 변환해 실행하는 JIT(Just In Time) 컴파일 방식이 있다. 두 방식 가운데 후자인 JIT 컴파일 방식이 더 많이 사용된다.

바이트 코드 프로그램은 일반적으로 바이트 코드 파일, 클래스 파일, 매니지드 코드 어셈블리로 불린다. 대체로 빌드 시스템은 바이트 코드 인터프리터에 의해 메모리에 로드될 파일을 생성한다.

다음으로 넘어가기 전에 앞에서 설명한 바이트 코드 모델과, 펄과 파이썬의 바이트 코드 모델을 비교해보자.

빌드 시스템 입장에서 펄과 파이썬 언어는 컴파일된다기보다는 인터프리트된다. 실제로 두 언어의 빌드 프로세스에는 컴파일 단계가 없고, 소스 파일을 모아 릴리스 패키지에 모아두기만 한다. 하지만 이런 인터프리트형 언어는 실행 중에 바이트 코드를 사용한다. 펄과 파이썬 스크립트가 바이트 코드로 변환된다는 관점에서 전통적인 빌드 시스템이 실행 환경에 포함돼 있다고 볼 수 있다.

이 방법의 장점은 일련의 수정, 컴파일, 실행 과정 중 컴파일을 하지 않아도 되므로 소스코드의 수정을 바로 다음 실행 때 반영할 수 있다는 점이다. 하지만 실행하기 전까지 소스코드의 구문 에러를 확인할 수 없다는 단점을 가지고 있다.

라이브러리

라이브러리 또한 빌드 시스템에서 생성되는 중요한 결과물이다. 하나의 개발 조직이 프로그램 코드 전체를 만든다고 생각할 수도 있지만, 실제로 그런 경우는 많지 않다. 대신 개발자는 다른 개발자나 외부에서 작성된 라이브러리를 사용하는 경우가 많이 존재한다. 이런 라이브러리는 디스크에 파일로 존재하며, 여러 프로그램에서 재사용할 수 있는 함수의 집합으로 구성돼 있다. 결국 개발자는 항상 프로그램 전체를 만들기보다는 이미 빌드된 라이브러리를 자신이 구현한 부분과 결합해 하나의 프로그램을 만든다.

많은 프로그래밍 언어에서 라이브러리 함수는 언어의 확장인 것처럼 생각되며, 사용자가 직접 만든 함수들과 동일한 방법으로 사용된다. 예를 들면 C언어에서는 화면에 문자를 출력하기 위해서 printf 라이브러리 함수를 사용한다. 자바에서는 println 메소드를 사용한다.

위에서 설명한 두 에제에서는 개발자는 다른 사람이 이미 만들어 놓은 함수나 메소드를 사용했고, 빌드 프로세스의 링크 과정을 통해 손쉽게 하나의 실행 파일을 생성했다.

대부분의 운영체제는 파일과 네트워크 I/O, 수학, 사용자 인터페이스, 데이터 베이스에 접근하기 위한 함수를 라이브러리에 포함해 설치 파일과 함께 배포한다. 또한 개발자는 인터넷으로부터 서드파티 라이브러리를 다운로드할 수 있으며, 자신이 직접 만든 라이브러리를 배포할 수도 있다.

빌드 시스템에서 라이브러리와 관련된 두 가지 주요한 작업은 다음과 같다.

새로운 라이브러리 생성

자신만의 라이브러리를 만들기 위한 첫 단계는 소스파일을 컴파일해 오브젝트 파일을 생성하는 일이다. 생성된 오브젝트 파일은 링커나 아카이브를 사용해 하나의 라이브러리 파일로 만든다. 마지막으로 이 라이브러리에 포함된 함수의 인덱스를 생성한다.

라이브러리 링크해 사용하기

실행 프로그램을 생성할 때 빌드 시스템은 라이브러리 리스트를 제공해야 한다. 임의의 함수가 소스코드에서 사용됐지만, 개발자가 명시적으로 선언하지 않았다면 빌드 시스템은 라이브러리 리스트를 검색해 해당 함수를 찾는다. 그 함수를 찾았다면 해당 오브젝트 파일을 실행 프로그램안에 복사해줘야 한다.

라이브러리의 링킹 방법

정적 링크

정적 링크 방식에서의 라이브러리는 각 오브젝트 파일의 묶음이다. 빌드 프로세스에서 링커는 사용된 함수가 오브젝트 파일을 라이브러리에서 찾아 실행 파일안에 복사한다. 이 과정에

서 라이브러리 안에 있는 오브젝트 파일은 사용자가 직접 만든 오브젝트 파일과 동일하게 다뤄진다.

정적 라이브러리와 직접 만든 코드가 링크되는 과정은 빌드 프로세스에서 이뤄지며, 최종적으로 생성된 실행 프로그램은 타겟 머신의 메모리에 로드돼 실행된다. 그러므로 정적 라이브러리는 런타임보다는 빌드와 관련이 있다고 할 수 있다. 또한 실행 프로그램이 만들어진 이후에는 라이브러리와 분리될 수 없다.

동적 링크

static link와는 달리 dynamic link방식은 오브젝트 파일을 실행 파일로 복사하는 대신, 실행 시점에 필요한 라이브러리를 실행 파일 안에 기록해 두고 프로그램이 실행되면 라이브러리는 프로그램과는 별도로 메모리에 로드돼 메인 프로그램과 연결되는 과정을 거친다. 이때 다이나믹 링커는 프로그램이 사용하는 함수와 이 함수를 제공하는 라이브러리를 연결한다.

빌드 시스템에서 동적 라이브러리는 오브젝트 파일을 결합해 만든 파일이며, 릴리스 패키지에 담아 타겟 머신에 설치된다. 이 과정을 거친 후 타겟 머신의 메모리에 로드된다.

다이나믹 라이브러리는 static 라이브러리에 비해 다소 복잡한 방법이기는 하지만 두 가지 큰 장점이 있다.

첫 번째로 동적 라이브러리는 버그나 기능 추가로 발생하는 새로운 버전으로의 업그레이드가 필요하면 실행 프로그램을 재 컴파일하지 않아도 된다.

두 번째로 하나의 라이브러리를 메모리에 로드한 후 같은 라이브러리를 사용하는 프로그램 간에 라이브러리를 공유하는 방식이기 때문에 운영체제의 메모리를 더욱 효율적으로 사용할 수 있다. 이런 장점들은 정적 링크 방식과 거리가 멀다.

설정 파일과 데이터 파일

모든 프로그램은 데이터 타입을 사용한다. 단순히 두 숫자를 더할 때에도 숫자 타입이 사용된다. 초기화된 숫자 배열과 같이 실행 프로그램에 직접 링크되는 데이터도 있지만, 규모가 어느 정도 있는 프로그램은 운영체제로 하여금 데이터를 메모리로 로드하게 요청한다.

데이터를 사용하는 데 특별한 제약 사항은 없지만, 예를 들어 다음과 같은 다양한 형태의 데이터가 사용될 수 있다.

- 화면에 표시되는 비트맵 형식의 그래픽 이미지
- 웨이브 형태로 디지털화돼 저장된 사운드 파일
- 프로그램의 동작을 조정하는 설정 파일
- 도움말 정보를 갖고 있는 문서 파일
- 이름과 주소를 저장하고 있는 데이터베이스

빌드 시스템은 이와 같은 파일들이 프로그램의 릴리스 패키지에 포함돼 타겟 머신에 설치 될 수 있게 해야 한다. 다른 방법으로는 데이터 파일을 인터넷에 저장해 두고 원격에서 접근해 사용하는 방법되 있으며, 이때는 데이터르 ㄹ릴리스 패키지에 포함시킬 필요가 없어진다.

데이터 파일은 아무 수정 없이 그 자체로 릴리스 패키지에 포함되는 것이 일반적이지만, 빌드 프로세스에 의해 생성되거나 수정돼 포함되는 일도 있다.

분산형 프로그램

프로그램이 시스템의 여러 부분으로 분산되는 환경이다.

최근 대부분의 운영체제는 동시에 여러 프로그램이 실행될 수 있기 때문에 여러 프로그램이 서로 통신하는 방식으로 하나의 소프트웨어를 구성할 수 있다. 더욱이 멀티프로세스 방식을 넘어 지리적으로 멀리 떨어진 컴퓨터 간에 네트워크로 연결돼 마치 단일 소프트웨어 프로그램처럼 동작할 수도 있다.

단순히 단일 디스크에 저장되는 하나의 실행 파일을 생성할 때와는 달리 분산형 프로그램 환경에서 릴리스 패키지를 생성하는 일은 더욱 중요해진다. 빌드 시스템은 각 실행 파일을 빌드하고 패키지에 포함해야하며, 각 실행 파일이 정상적으로 실행할 수 있게 여러 개의 설정 파일과 스크립트를 준비해야 한다.

여러 다른 컴퓨터에서 실행되는 각 클라이언트와 단일 컴퓨터에서 실행되는 하나의 서버로 구성된 서버/클라이언트 모델을 생각해보자. 이때 빌드 시스템은 두 개의 릴리스 패키지를 별도로 생성해 서버 프로그램과 클라이언트 프로그램을 각기 다른 사용자가 설치하게 할 수도 있고 단일 패키지로 제공해 두 개의 프로그램을 사용자가 선택할 수 있게 할 수도 있다.

빌드 시스템은 여러 프로그램을 지원하기 위해 몇 가지 추가적인 기능이 필요하다. 예를 들면 소프트웨어를 빌드할 때마다 모든 프로그램의 소스 파일을 재빌드하고 패키지를 생성하기보다는 빌드할 프로그램을 지정할 수 있어야 한다. 실제로 개발자는 한 번에 하나의 프로그램에서 작업하는 시간이 더 많아 이런 기능은 개발 생산성에 직접 영향을 미친다.

또한 빌드 시스템은 여러 개의 프로그램 간에 공통으로 사용되는 API를 지원해야 한다. 서로 다른 두 개의 프로그램이 네트워크 통신을 한다고 생각해보자. 두 프로그램은 여러 개의 데이터 구조체를 공유하며 네트워크로 이 데이터 구조체를 전송한다면 빌드 시스템은 두 프로그램이 다른 데이터 정의를 사용해 데이터 타입의 불일치가 발생하는 문제를 피할 수 있게 지원해야 한다.

분산 시스템에서 소프트웨어가 커지고 복잡해짐에 따라 빌드 시스템 역시 복잡해진다는 점이다.