링커와 로더

SandeepGrover

sgrover (at) quicklogic.com

도운이

윤인수

isyoon (at) pusan.ac.kr

이 글은 컴파일러, 링커, 로더들이 어떻게 동작하는지를 간략히 보이며, 또한 공유 라이브러리의 이점에 대해서 얘기한다.

고친 과정

고침 0.12002-11-28고친이 쿨링팬

최초 번역

1. 서문

링킹(linking)은 여러가지 코드와 데이터를 묶어 메모리로 로드될 수 있는 하나의 실행가능한 파일을 만드는 작업이다. 링킹은 컴파일-타임때 행해질 수도 있고, 로드-타임(로더에 의해), 혹은 런-타임(응용 프로그램에 의해)때도 행해질 수 있다. 1940년대에는 이러한 링킹작업을 사람이 손수 하였다. 현재에는 공유 라이브러리(shared library)들을 동적으로 링킹시켜주는 등의 복잡한 일을 할 수 있는 링커(linker)라는 것이 있다. 이 문서는 링킹의 모든 과정, 예로들자면 재배치(relocation)와 심볼 해석(symbol resolution)부터 위치 독립적(position independent)인 공유 라이브러리 지원등에 대해 다룬다.문제를 간단하고 이해하기 쉽게 하기 위해, 나는 이 문서를 x86 아키텍처에 기반한 리눅스와 GNU 컴파일러(GCC)와 링커(ld)에 기반한 ELF(executable and linking format) 실행파일에 초점을 맞추었다. 그러나, 기본적인 링킹의 아이디어는 운영체제, 프로세서 또는 오브젝트 파일의 형식에 무관하게 적용될 수 있다.

1.1. 저작권 정보

이 문서는 2002년 11월 26일, Linux Journal (http://www.linuxjournal.com/)에 Sandeep Grover씨가 Linkers and Loaders (http://www.linuxjournal.com/article.php?sid=6463)라는 제목으로 기재하신 글입니다. 원 저자에게 메일로 연락하여 실렸던 잡지의 이름과 원저자가 누군지를 밝히면 번역을 해도 좋다는 동의를 얻었습니다.

1.2. 피드백

이 문서에 대한 발전적인 제안이나 수정사항, 문제점 등에 대한 피드백은 언제든지 환영합니다. 메일을 보내 주십시오.

2. 컴파일러, 링커, 로더들의 동작: 기본 사항

a.c와 b.c 두 개의 프로그램이 있다고 가정하고, 쉘 프롬프트에서 a.c와 b.c를 gcc를 이용하여 아래와 같은 명령을 수행하면 다음과 같은 일들이 순서대로 수행된다.

gcc a.c b.c

- a.c에 대하여 전처리기(preprocessor)를 수행시키고, 그 결과를 전처리된 임시파일에 저장한다.

  cpp [other-command-line options] a.c `/tmp/a.i`

- a.i에 대하여 컴파일러를 수행시키고, a.s라는 어셈블러 코드를 생성한다.

  cc1 [other-command-line options] `/tmp/a.i` -o `/tmp/a.s`

- a.s에 대하여 어셈블러를 수행시키고, a.o라는 오브젝트 파일을 생성한다.

  as [other-command-line options] `/tmp/a.s` -o `/tmp/a.o`

cpp, cc1, as는 GNU의 전처리기, 컴파일러, 어셈블러를 각각 나타내며, GCC 배포본 안에 들어있다.

위와 같은 스텝은 b.c에도 똑같이 적용되어 b.o라는 오브젝트 파일을 하나 더 생성하게 된다. 그러면 링커의 작업은 이러한 두 개의 오브젝트 파일들(a.o, b.o)을 입력으로 받아서 최종적으로 실행가능한 파일을 만드는 것이다.

ld [other-command-line options] `/tmp/a.o /tmp/b.o` -o a.out

최종적으로 만들어진 실행파일(a.out)은 이제 로드될 준비가 되었다. 이것을 실행시키기 위해서 우리는 쉘 프롬프트상에서 아래와 같이 타이핑한다.

./a.out

그러면 쉘은 로더를 불러 a.out의 코드와 데이터를 메모리로 복사하고, 프로그램내의 제일 처음으로 제어권을 넘긴다. 여기서 말하는 로더는 execve라는 것으로 실행가능한 오브젝트 파일의 코드와 데이터를 메모리로 로드하고 그 프로그램의 첫번째 명령어가 저장된 주소로 점프함으로써 프로그램을 수행하게 한다.

a.out이라는 명칭은 a.out 오브젝트 파일들안에 있는 어셈블러의 출력물에서 그 유래를 찾을 수 있다. 그 이후로 오브젝트 형식은 다양하게 바뀌어 왔지만, 그 이름은 계속 사용되어지고 있다.

3. 링커와 로더

링커와 로더는 많은 부분이 연관되어 수행되지만 개념적으로는 다른 작업들을 수행한다.

- 프로그램 로딩(Program Loading). 이것은 프로그램을 실행가능한 상태로 만들기 위해 하드 디스크로부터 프로그램 이미지를 읽어서 메인 메모리로 복사하는 것을 말한다. 어떤 경우에는 프로그램 로딩이 저장(storage)공간을 할당하거나 가상주소를 디스크 페이지로 매핑하는 일도 한다.
- 재배치(Relocation). 컴파일러와 어셈블러는 각각의 입력 파일들로부터 시작주소가 제로인 오브젝트 코드를 생성한다. 재배치라는 것은 프로그램의 각기 다른 부분들(코드와 데이터)에 대해 로드되는 주소를 할당하는 것이다. 이러한 작업은 같은 타입(코드 혹은 데이터)으로 정의된 모든 구간들을 하나의 구간으로 합치고, 이러한 구간들이 런-타임때 올바른 주소를 가리킬 수 있도록 조정하는 것을 말한다.
- 심볼 해석(Symbol Resolution). 프로그램은 다양한 하위 프로그램(subprogram)들도 구성된다; 하나의 상위 프로그램이 다른 하위 프로그램을 참조하는 것은 심볼이라는 것을 통해 이루어진다. 링커의 작업은 이러한 심볼의 위치를 알아내어 상위 프로그램의 오브젝트 코드에 하위 프로그램의 주소를 기입하여 참조를 해석하도록 한다.

링커와 로더사이에는 중첩되는 일들과 각각 차이나는 일들도 있는데, 이렇게 생각하도록 하자: 로더는 프로그램이 로딩되도록 하며; 링커는 심볼을 해석하며; 링커와 로더, 둘 다 재배치를 할 수 있다.

4. 오브젝트 파일들

오브젝트 파일들은 세가지로 분류될 수 있다.

- 재배치 가능한 오브젝트 파일(Relocatable object file). 이것은 바이너리 코드와 데이터를 가지고 있으며, 실행가능한 오브젝트 파일을 만들기 위해 컴파일-타임때 재배치 가능한 다른 오브젝트 파일들과 결합될 수 있는 것을 가리킨다.
- 실행가능한 오브젝트 파일(Executable object file). 이것은 바이너리 코드와 데이터를 가지고 있으며, 메모리로 직접 로드되어 실행될 수 있는 것을 가리킨다.
- 공유 오브젝트 파일(Shared object file). 이것은 재배치 가능한 오브젝트 파일의 특별한 타입으로, 로드-타임이나 런-타임때 동적으로 메모리로 로드되고 링킹될 수 있는 것을 가리킨다.

컴파일러와 어셈블러는 재배치 가능한 오브젝트 파일을 생성한다(공유 오브젝트 파일도 또한 생성한다). 링커는 이러한 오브젝트 파일들을 합쳐 실행가능한 오브젝트 파일들을 생성한다.

오브젝트 파일들은 시스템에 따라 그 형식이 틀리다. 최초의 유닉스 시스템은 a.out 포맷을 사용하였다. System V의 초기 버전에서는 COFF(Common object file format)라는 것을 사용하

였고, 윈도우즈 NT는 COFF의 변형인 PE(portable executable)라는 형식을 사용한다; IBM은 독자적인 IBM 360 형식을 사용한다. 리눅스와 솔라리스와 같은 현대적인 유닉스 시스템들은 유닉스 ELF(executable and linking format)포맷을 사용한다. 이 문서는 주로 ELF에 대해 다룬다.

표 **1.** 전형적인 재배치 가능한 **ELF** 오브젝트 파일의 형식

ELF Header
.text
.rodata
.data
.bss
.symtab
.rel.text
.rel.data
.debug
.line
.strtab

ELF 헤더는 4-byte magic문자열(177ELF)로 시작한다. ELF 재배치 가능한 오브젝트 파일의 각 구간의 의미는 아래와 같다.

- .text, 컴파일된 코드의 머신 코드가 들어있다.
- .rodata, read-only 데이터가 들어있다, printf문의 문자열등이 이에 해당한다.
- .data, 초기화된 전역 변수들이 들어있다.
- .bss, 초기화되지 않은 전역 변수들이 들어있다. BSS는 block storage start의 이니셜이고, 이 구간은 실제적으로 오브젝트 파일에서 공간을 차지하지 않고 단지 공간을 확보하는 역할만 한다.
- .symtab, 프로그램에서 정의된 전역 변수들과 함수들에 대한 참조 정보를 가지고 있다. 이 테이블은 지역 변수에 대한 것은 담고 있지 않다; 지역 변수들은 스택에 의해 유지된다.
- .rel.text, .text에 들어있는 각 머신 코드의 위치를 나타낸다. 이것들은 나중에 링커가 이 오브젝트 파일을 다른 오브젝트 파일들과 연결시킬때 필요하다.
- .rel.data, 현재의 파일에서는 정의되어 있지 않지만 참조되는 전역 변수에 대한 재배치 정보를 담고 있다.
- .debug, 지역, 전역 변수들에 대한 디버깅 심볼들이 들어있다. 이 구간은 컴파일러가 -g 옵션과 함께 수행될 때 생성된다.
- .line, .text에 들어있는 머신 코드와 실제 C 코드의 라인 넘버에 대한 메핑 정보가 들어있다. 디버거 프로그램이 이 정보를 필요로 한다.

- .strtab, .symtab, .debug 구간에 있는 심볼 테이블에 들어있는 스트링들에 대한 테이블이다.

5. 심볼들과 심볼 해석

모든 재배치 가능한 오브젝트 파일들은 심볼 테이블과 그와 관련된 심볼들을 가지고 있다. 링커의 관점에서 볼 때 심볼들을 다음과 같이 분류할 수 있다.

- 현재의 파일에서 정의되고, 다른 파일들에서 참조되는 전역 심볼. 모든 non-static 함수들과 전역 변수들이 이 분류에 해당한다.
- 현재의 파일에서 참조는 되나, 다른 곳에서 정의된 전역 심볼. extern으로 정의된 모든 함수들과 변수들이 이 분류에 해당한다.
- 현재의 파일에서만 정의되고 참조되는 지역 심볼. 모든 static 함수들과 변수들이 이 분류에 해당한다.

링커는 심볼의 참조를 해석할 때, 입력으로 주어지는 재배치 가능한 오브젝트 파일의 심볼 테이블로부터 꼭 하나만 존재하는 심볼의 정의를 참조하여 심볼 참조를 해석한다. 지역 심볼(local symbol)은 그에 대한 다중 정의(multiple definitions)를 심볼 테이블이 가질 수 없으므로 쉽게 해석된다. 그러나 전역 심볼의 해석은 약간의 트릭이 요구된다. 컴파일 타임때, 컴파일러는 전역 심볼들을 strong 혹은 weak한 것으로 만드는데, 함수들과 초기화된 전역 변수들은 strong하게, 초기화되지 않은 변수들은 weak하게 만든다. 그러면 링커는 아래의 룰을 적용하여 심볼들을 해석하게 된다.

1. 다중 strong 심볼들은 허가되지 않는다.
2. 하나의 strong 심볼과 여러개의 weak 심볼들이 있으면, strong 심볼을 선택한다.
3. 여러개의 weak 심볼들이 있으면, 그것들중 아무거나 선택한다.

예로, 다음과 같은 두 프로그램의 링킹은 링크-타임 에러를 낸다.

```
/* foo.c */

int foo() {
return 0;
}

/* bar.c */

int foo() {
return 1;
}

int main() {
foo();
```

}

foo (전역 함수로써 strong 심볼이다)가 두 번 정의 되었으므로, 링커는 아래와 같은 에러 메세지를 낸다.

gcc foo.c bar.c

/tmp/ccM1DKre.o: In function 'foo':

/tmp/ccM1DKre.o(.text+0x0): multiple definition of 'foo'

/tmp/ccIhvEMn.o(.text+0x0): first defined here

collect2: ld returned 1 exit status

collec2는 GCC에 의해 호출되는 링커 ld의 wrapper이다.

6. 정적 라이브러리의 링킹

정적 라이브러리는 비슷한 형을 지닌 오브젝트 파일들의 집합이다. 이러한 라이브러리들은 디스크에 아카이브(archive) 형식으로 저장된다. 아카이브는 라이브러리를 구성하고 있는 것들을 좀 더 빠르게 검색하기 위해 디렉토리 정보를 또한 가지고 있다. 각각의 ELF 아카이브는 !<arch>\n (\n은 뉴라인을 뜻한다)의 8자로 구성된 magic 문자열로 시작한다.

정적 라이브러리들은 링커에게 인자 (arguments)로써 전달된다. 그러면 링커는 프로그램에서 참조되는 오브젝트 모듈들만을 복사한다. 유닉스 시스템에서 libc.a는 모든 C 라이브러리 함수들 (printf나 fopen등과 같은)을 담고 있다.

gcc foo.o bar.o /usr/lib/libc.a /usr/lib/libm.a

libm.a는 유닉스 시스템에서 sqrt, sin, cos과 같은 수학관련 함수들을 담고 있는 라이브러리이다.

정적 라이브러리를 이용할 때, 심볼 해석과정이 어떻게 이루어지나 보면, 링커는 커맨드 라인에서 입력으로 받은 재배치 가능한 오브젝트 파일들과 아카이브들을 왼쪽에서 오른쪽으로 스캔한다. 이러한 스캔 과정중에, 링커는 세가지의 집합을 유지한다. 먼저, 재배치 가능한 오브젝트 파일들이 실행 가능한 파일의 상태로 들어간 집합 O; 아직 해석되지 않은 심볼들을 담고 있는 집합 U, 이전의 입력 파일에서 정의된 심볼을 담고 있는 집합 D가 그것이다. 이러한 집합들은 초기에 비워진 상태이다.

- 커맨드 라인의 각각의 입력 파일에 대해, 링커는 그것이 오브젝트 파일인지 아카이브인지를 먼저 결정한다. 만약 입력이 재배치 가능한 오브젝트 파일이면, 링커는 그것을 집합 O에 추가하고, 집합 U와 D를 업데이트한 후, 다음 입력 파일로 넘어간다.
- 만약 입력 파일이 아카이브라면, 링커는 현재의 집합 U에 들어있는 아직 해석이 안된 심볼들을 해석하기 위해 아카이브를 구성하고 있는 멤버 모듈들을 스캔하여 풀어나간다. 만약 아카이브를 구성하고 있는 멤버 모듈 자체에도 해석이 안된 심볼들이 있으면, 그 멤버는 집합 O에 추가되고 집합 U와 D도 업데이트 된다. 이러한 과정은 아카이브를 구성하고 있는 모든 오브젝트 파일 모듈들에 대해 행해진다.
- 만약 커맨드 라인에 주어진 모든 입력 파일들에 위의 두 스텝을 행하였는데도, 집합 U가 아직 비어 있지않다면, 링커는 에러 메세지를 내고 종료한다. 집합 U가 비어 있다면, 링커는 집합 O에 들어있는 모든 오브젝트 파일들을 병합하고 재배치시켜 실행가능한 출력 파일을 만들어 낸다.

위의 일련의 순서때문에 커맨드 라인에서 정적 라이브러리가 끝에 온다. 또한 라이브러리들 사이에 발생할 수 있는 순환적인 의존성도 주의깊게 살펴야한다. 입력으로 주어지는 라이브러리들은 순서대로 주어져서 아카이브의 멤버들이 참조할 수 있도록 해야하며, 정의된 하나의 심볼은 뒤따르는 커맨드 라인의 입력에 의해 참조되어야 한다. 만약 해석이 안된 심볼이 있고, 그 심볼이 여러 정적 라이브러리들내에서 정의되어 있으면, 커맨드 라인에서 처음에 주어진 라이브러리에 정의된 것을 받아들인다.

7. 재배치(Relocation)

링커가 모든 심볼을 해석하고 나면, 심볼 참조는 오직 하나의 심볼 정의만을 가지게 된다. 그때, 링커는 아래 두 스텝으로 구성된 재배치 작업을 하게된다.

- 섹션과 심볼정의들을 재배치한다. 링커는 같은 타입의 모든 섹션들을 새로운 하나의 섹션으로 통합한다. 예로 들면, 링커는 입력으로 받은 모든 재배치 가능한 오브젝트 파일들의 .data섹션을 합쳐 하나의 .data섹션을 만든다. 같은 과정이 .code에 대해서도 행해진다. 그런 후에 링커는 병합된 새로운 섹션과, 병합된 새로운 섹션내의 각 섹션, 그리고 모든 심볼들에 대해 런-타임 메모리 주소를 할당한다. 이러한 작업후에는 프로그램의 모든 코드와 전역 변수들은 고유한 로드-타임 주소를 가지게 된다.
- 섹션들안에 있는 심볼의 참조를 재배치한다. 이 과정에서, 링커는 코드와 데이터 섹션에 있는 모든 심볼 참조를 수정하여, 그것들이 올바른 로드-타임 주소를 가지게 한다.

어셈블러가 해석안된 심볼들을 만날 때마다, 어셈블러는 오브젝트 파일의 .rel.text/.rel.data 섹션에 해석안된 심볼들을 위한 재배치 항목을 생성한다. 이러한 재배치 항목은 해석안된 심볼들이 어떻게 해석되어야 하는지에 대한 정보들을 담고 있다. 전형적인 ELF 재배치 항목은 다음과 같은 멤버들로 구성된다.

- 옵셋, 재배치되어질 필요가 있는 심볼 참조의 섹션내애서의 옵셋을 나타내며, 혹은 디스크의 저장공간이 오브젝트 파일내에서 재배치되어질 필요가 있을 시, 이 값은 재배치될 필요가 있는 디스크 섹션의 처음부터 바이트단위로 얼마만큼 떨어져 있는가를 나타낸다.

- 심볼, 이것은 심볼 테이블에서의 인덱스로서, 아직 해석이 안된 심볼이 심볼 테이블에서 몇 번째 위치에 있는가를 나타낸다.
- 타입, 재배치 타입, 일반적으로 R_386_PC32, 이는 PC-relative 주소지정방식을 나타내며, R_386_32는 절대주소지정방식을 나타낸다.

링커는 재배치 가능한 오브젝트 모듈들 안에 있는 모든 재배치 엔트리에 대해 이 작업을 반복하고 그것들의 타입에 따라 해석안된 심볼들을 재배치한다. R_386_PC32는 재배치 주소를 S+A-P로 계산하며, R_386_32는 S+A로 계산한다. 이 계산에서, S는 재배치 항목의 심볼항목에 들어있는 값을 가리키며, P는 섹션 옵셋 혹은 재배치되는 저장장치의 주소를 나타낸다 (재배치 항목의 옵셋값으로부터 계산된다). 그리고 A는 재배치 가능한 필드를 계산하는데 필요한 주소이다.

8. 동적 링킹: 공유 라이브러리

위의 정적 라이브러리는 몇가지 중요한 단점들을 지니고 있다; 예로, printf나 scanf와 같은 함수들을 고려해보자. 이러한 함수들은 거의 모든 응용 프로그램에서 사용된다. 만약 시스템이 50~100개의 프로세스를 동작시키고 있다면, 각 프로세스는 각각 printf와 scanf의 실행 가능한 코드의 복사본을 가지고 동작하게 된다. 이것은 메모리 공간의 중대한 낭비를 초래한다. 공유 라이브러리는 이러한 정적 라이브러리의 단점을 해결한다. 공유 라이브러리는 런-타임때 메모리의 임의의 위치로 로드될 수 있는 오브젝트 모듈이다. 그리고 그것은 메모리에서 프로그램과 링킹될 수 있다. 공유 라이브러리는 종종 공유 오브젝트라고도 불리운다. 대부분의 유닉스 시스템에서는 .so로 공유 라이브러리 파일명이 끝나며; HP-UX에서는 .sl로 끝나고 마이크로 소프트사는 DLL(dynamic link libraries)로 부른다.

공유 오브젝트를 만들기 위해, 컴파일러는 다음과 같은 옵션을 가지고 호출된다.

gcc -shared -fPIC -o libfoo.so a.o b.o

위의 명령어는 컴파일러가 a.o, b.o라는 두 개의 오브젝트 모듈들로부터 libfoo.so라는 공유 라이브러리를 만들도록 한다. -fPIC 옵션은 컴파일러에게 위치 독립적인 코드(position independent code)를 만들도록 한다.

만약 bar.o라는 오브젝트 모듈이 a.o, b.o와 의존성이 존재한다고 가정하면, 링커는 다음처럼 불리워진다.

gcc bar.o ./libfoo.so

이 명령어는 로드-타임때 링크될 수 있는 a.out이라는 실행파일을 만든다. 여기서 a.out은 정적 라이브러리를 사용할 때는 포함되었던 a.o와 b.o 오브젝트 모듈을 포함하고 있지 않다. 이

실행파일은 런-타임때 libfoo.so와 함께 해석될 수 있는 재배치와 심볼 테이블만을 포함하고 있다. 따라서, a.out은 libfoo.so와의 의존성이 존재하는 부분적으로 실행가능한 파일(partially executable file)이다. 이 실행가능한 파일은 .interp라는 섹션을 가지고 있는데, 이 섹션은 동적 링커(dynamic linker)의 이름을 가리키고 있다. 동적링커 자체도 리눅스에서는 ld-linux.so라는 공유 오브젝트이다. 그래서 실행파일이 메모리로 적재될 때, 로더는 제어권을 동적링커로 넘긴다. 동적링커는 공유 라이브러리들과 해당 프로그램의 주소공간을 매핑시킬 수 있는 start-up 코드를 가지고 있고 다음과 같이 동작한다.

- libfoo.so의 코드와 데이터를 메모리 속으로 재배치한다.
- a.out에 있는 참조들을 libfoo.so에 정의되어 있는 것으로 재배치한다.

마지막으로, 동적링커는 제어권을 응용프로그램으로 넘긴다. 이때부터 공유 오브젝트는 메모리에 고정되게 된다.

9. 응용 프로그램으로부터 동적 라이브러리 로딩

동적 라이브러리들은 응용 프로그램이 실행되고 있는 중에도 로드될 수 있다. 응용 프로그램은 공유 라이브러리들을 자신과 링킹하지 않아도, 동적링커에게 요청하여 공유 라이브러리들을 로드하고 링킹시킬 수 있다. 리눅스와 솔라리스 그리고 다른 시스템들에서는 이를 위해 동적으로 공유 오브젝트들을 로드할 수 있는 몇가지 종류의 함수들을 제공한다. 리눅스에서는 공유 오브젝트를 열 수 있는 dlopen; 공유 오브젝트의 심볼 테이블을 볼 수 있는 dlsym, 공유 오브젝트를 닫을 수 있는 dlclose와 같은 시스템 콜을 제공하며, 윈도우즈에서는 LoadLibrary와 GetProcAddress와 같은 함수들을 제공한다.

오브젝트 파일들을 조작할 수 있는 툴들

여기에 오브젝트 파일들과 실행파일들을 조사할 수 있는 툴들의 목록이 있다.

- ar: 정적 라이브러리들을 만든다.
- objdump: 가장 중요한 바이너리 툴; 바이너리 형식 오브젝트 파일의 모든 정보를 보여준다.
- strings: 바이너리 파일의 출력가능한 모든 문자열들을 보여준다.
- nm: 오브젝트 파일의 심볼 테이블에 정의된 심볼들의 리스트를 보여준다.
- ldd: 오브젝트 바이너리 파일이 의존하고 있는 공유 라이브러리들의 목록을 보여준다.
- strip: 심볼 테이블 정보를 지운다.

10. 읽어볼 것들

Linkers and Loaders (http://www.iecc.com/linker) by John Levine

Linkers and Libraries Guide from Sun (http://docs.sun.com/db?p=/doc/816-1386)