14장 실행 가능한 프로그램 작성

14.7 동적 링킹

동적 링킹을 지원하기 위해 라이브러리는 (1) 자신을 사용하는 모든 프로그램에서 동일한 주소에 위치하거나 (2) 세그먼트의 내용이 주소에 의존적이지 않도록 자신의 코드 세그먼트에 재배치 가능한 단어를 두지 않아야 한다. 첫 번째 접근 방법은 직관적이지만 제한적이다. 이를 위해서는 일반적으로 공유 가능한 모든 라이브러리에 유일한 주소를 할당해야 한다. 그렇지 않으면 새로 생성된 프로그램이 서로 겹치는 주소 범위를 부여 받은 두 개의 라이브러리를 사용할 수도 있는 위험을 감수해야 한다. 유일-주소 방식을 채택한 유닉스 시스템 V R3에서 공유 라이브러리는 시스템 관리자만이 설치할 수 있었으며, 이러한 요구 사항 때문에 동적 링킹을 비교적 적은 수의 널리 쓰이는 라이브러리로 제한되는 경향이 있었다. 공유 라이브러리가 어느 주소에나 링킹될 수 있는 두 번째 접근 방법은 사용자가 원할 때마다 동적 링킹을 할 수 있게 해준다.

【14.7.1】 위치 독립적 코드

재배치 가능한 단어를 포함하지 않는 코드 세그먼트를 **위치 독립적 코드**(PIC, position-independent code)라고 한다. PIC를 생성하기 위해 컴파일러는 다음의 규칙들을 따라야 한다.

1. 모든 내부 분기에 대해 절대 주소로의 건너뛰기 대신 PC-상대적 주소 지정을 사용한다.

2. 규칙 1과 비슷하게 어떤 표준 기준 레지스터에 대한 변위 주소 지정을 사용함으로써 정적으로 할당된 자료에 대한 절대 참조를 피한다. 코드와 자료 세그먼트가 서로 알려진 오프셋에 위치한다는 것이 보장되면 공유 라이브러리로의 진입 지점은 PC를 사용해서 적절한 기준 레지스터 값을 계산할 수 있다. 그렇지 않은 경우 호출자는 호출 순서의 일부로서 기준 레지스터를 설정해야 한다.

3. PIC 세그먼트로부터의 모든 제어 전송과 이에 대응되는 자료 세그먼트 외부의 정적 메모리 저장이나 불러오기에 대해 부가적인 수준의 간접 지정을 사용해야 한다. (PIC가 아닌) 목표 주소는 간접 지정을 통해 각 프로그램 인스턴스의 전용인 자료 세그먼트에 유지될 수 있다.

예 14.19

MIPS/IRIX 하의 PIC

정확한 세부 사항은 프로세서, 판매 업체, 운영체제에 따라 달라진다. IRIX 6.2 버전 유닉스 하의 MIPS 구조를 위한 SGI의 컴파일러에 대한 규약은 ©(심화학습에 있는) 그림 14.12와 같다. 각 공유 코드 세그먼트는 정적 오프셋에서는 비공유 링크 테이블을 동반하며 임의의 오프셋에서는 비공유 자료 세그먼트를 동반한다. 링크 테이블은 코드 세그먼트에서 참조하는 모든 외부 기호의 주소를 담고 있다.

©(심화학습에 있는) 8.2.2절에서 설명한 바와 같이 리프가 아닌 모든 서브루틴은 `ra`(복귀 주소) 레지스터의 값을 유지하기 위해 자신의 스택 프레임에 공간을 할당하고 이 레지스터를 자신의 프롤로그와 에필로그에 저장하고 복구해야 한다. 이와 마찬가지로 동적으로 링킹된 공유 라이브러리로 호출할 수 있는 모든 서브루틴은 `gp`(전역 포인터) 레지스터를 프롤로그에 저장하고 동적으로 링킹된 공유 라이브러리로의 매 호출 후마다 이를 복구해야 한다. 코드 생성 시 컴파일러는 이러한 라이브러리에 어느 외부 기호가 위치하는지를 알아야 한다. 동적으로 링킹되는 공유 라이브러리 중 하나로 호출하는 경우 보통의 `jal`(건너뛰기 후 연결, jump-and-link) 명령어는 명령어 세 개의 나열로 대체된다. 첫 번째 명령어는 `gp`-상대적 주소 지정을 사용해서 링크 테이블로부터 레지스터 `t9`를 불러오는 것이다. 두 번째는 `t9`로부터 목표 주소를 가져오는 `jalr`(건너뛰기 후 레지스터 링킹, jump-and-link-register) 명령어다. (복구 후에 실행될) 세 번째 명령어는 `gp`를 복구한다. 비슷한 맥락에서 동적으로 링킹된 공유 라이브러리에 위치한 자료의 저장과 불러오기는 2-명령어 나열을 사용해야 한다. 첫 번째 명령어는 `gp`-상대적 주소 지정을 사용해서 링크 테이블로부터 자료의 주소를 불러온다. 두 번째는 자료 자체를 저장하거나 불러온다.

동적으로 링킹되는 공유 라이브러리로의 진입 역할을 하는 서브루틴 `foo`의 프롤로그는 새로운 `gp`를 수립해야 한다. 이를 위해 서브루틴은 `t9`의 값(즉, `foo`의 주소)을 취하고 코드 세그먼트 내에 있는 `foo`의 오프셋과 코드와 링크 테이블 간의 거리 사이의 (정적으로 알려진) 부호가 있는 차이를 더한다.

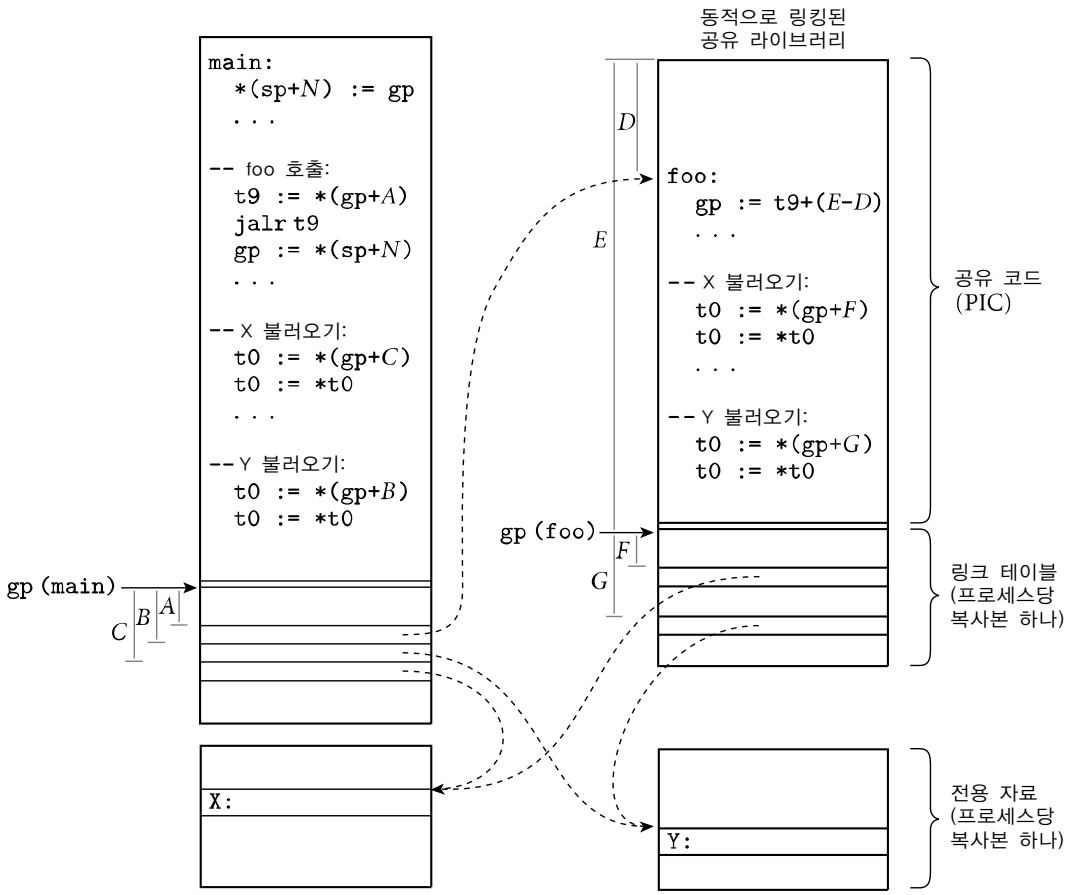

그림 14.12 | 동적으로 링킹된 공유 라이브러리. main이 라이브러리에 위치한 foo를 호출하기 때문에 main의 프롤로그와 에필로그는 ra(여기서 보이지 않음)와 gp를 모두 저장하고 복구해야 한다. foo에 대한 호출은 main의 링크 테이블에 저장된 주소를 사용해서 간접적으로 이뤄진다. 이와 마찬가지로 전역에서 볼 수 있는 변수 X와 Y에 대한 참조는 간접 지정을 사용해야 한다. foo의 프롤로그에서 gp는 t9의 값을 사용해서 foo의 링크 테이블을 가리키도록 설정된다. main의 호출 순서는 foo가 복귀할 때 이전의 gp를 복구한다.

【14.7.2】 완전히 동적인(게으른) 링킹

공유 라이브러리가 내보내는 모든 기호나 대부분의 기호가 부모 프로그램에 의해 참조되는 경우 로딩 시점에 라이브러리를 온전히 링킹하는 것이 타당하다. 그러나 임의로 주어진 프로그램 실행에는 실제로 사용되지 않는 라이브러리에 대한 참조가 있을 수 있는데, 이는 참조가 나오는 코드 경로를 따르는 실행을 입력 자료가 유발하지 않기 때문이다. 이러한 "잠재적으로 불필요한" 참조가 매우 많으면 라이브러리를 요구에 따라 게으르게 링킹함으로써 상당한 양의 작업을 피할 수 있다. 더욱이 기호를 모두

사용하는 프로그램에서조차 점진적인 게으른 링킹은 프로그램이 실행을 더 빠르게 시작할 수 있게 함으로써 시스템의 상호 응답성을 향상시킬 수 있다. 끝으로 프로그램 구성 요소의 동적 생성을 허용하는 언어 시스템(예를 들어 커먼 리스프나 자바)은 컴파일된 구성 요소에 있는 외부 참조의 결정(resolution)을 지연하기 위해 게으른 링킹을 사용해야 한다.

예 14.20

MIPS/IRIX 하의 동적 링킹

게으른 링킹을 위한 실행 시간 자료 구조는 (심화학습에 있는) 그림 14.20의 자료 구조와 거의 동일하지만 점진적으로 생성된다. 로딩 시점에 프로그램은 주 코드 세그먼트와 링크 테이블, 링크 테이블에 나와야 하는 주소에 대한 모든 자료 세그먼트 등을 가지고 시작한다. 앞서 살펴본 예에서는 X(main에 속한 변수)와 Y(foo에 속한 변수)의 주소가 모두 주 링크 테이블에 나와야 하기 때문에 main과 foo 모두의 자료 세그먼트를 로딩해야 한다. 그러나 foo의 주소가 링크 테이블에 나와야 함에도 불구하고 foo의 코드 세그먼트나 링크 테이블은 로딩하지 않는다. 대신 해당 링크 테이블 항목이 스터브 루틴을 참조하도록 초기화한다. 스터브 루틴은 컴파일러가 생성해 주 코드 세그먼트에 포함된다. 스터브 코드는 다음과 같을 수 있다.

```
t9 := *(gp+k)          -- 게으른 링커 진입 지점
t7 := ra
t8 := n                -- 스터브의 색인
call *lt9              -- ra 덮어쓰기
```

게으른 링커 자체는 (게으르지 않은) 로딩 시점에 프로그램에 링킹되는 공유 라이브러리에 상주한다(여기서는 게으른 링커의 주소가 링크 테이블에서 오프셋 k만큼의 위치에 있다고 가정했다).

게으른 링커로 분기한 후 제어는 스터브로 절대 복귀하지 않는다. 링커는 대신 프로그램 목적 파일의 들여오기 테이블에 색인하기 위해 상수 n을 사용한다. 그리고 링커는 프로그램의 목적 파일에서 미결정 참조의 이름과 라이브러리 모두를 식별하는 데 필요한 정보를 찾는다. 그 후 링커는 라이브러리의 코드 세그먼트가 아직 메모리에 없는 경우 해당 코드 세그먼트를 메모리로 로딩한다. 이 시점에서 링커는 링크 테이블 항목이 라이브러리 루틴을 가리키도록 스터브가 호출되었던 링크 테이블 항목을 변경("수정")할 수 있다. 라이브러리의 코드 세그먼트를 로딩할 필요가 있다면 링커는 라이브러리의 링크 테이블 복사본도 생성해야 한다. 링커는 세그먼트(의 복사본)가 이전 링킹 연산 중에 이미 로딩되지 않았다면 테이블의 모든 자료 항목들이 참조하는 세그먼트를 로딩하면서 이 자료 항목들을 모두 초기화한다. 라이브러리의 링크 테이블에 있는 각 서브루틴 항목에 대해 링커는 적절한 코드 세그먼트가 이미 로딩되었는지 확인한다. 이미 로딩되었다면 서브루틴 항목을 서브루틴 주소로 초기화하며 그렇지 않다면 서브루틴의 스터브 주소로 초기화한다. 끝으로 링커는 t7을 ra로 복사하고 새로 링킹된 라이브러리 루틴으로 건너뛴다. 이 시점에서 모든 것은 마치 호출이 일반적인 방식으로 일어난 것처럼 보이게 된다.

실행이 진행되면서 아직 로딩되지 않은 기호에 대한 참조는 프로그램의 "프런티어"를 확장한다. 링커 호출은 자료 참조가 아니라 서브루틴 호출 시에 일어나기 때문에 현재 프런티어는 항상 코드 세그먼트들과 이 코드 세그먼트들이 참조하는 자료 세그먼트를 포함한다. 각 링킹 연산은 새로운 코드 세그먼트 하나와 이 세그먼트가 참조하는 부가적인 자료 세그먼트 전부를 가져온다. 페이지 폴트를 차단하고자 한다면 아직 로딩되지 않은 자료에 대한 참조 시에 링커로 진입하게 할 수도 있다. 이러한 접근 방법을 통해 실제로는 절대 사용되지 않는 자료 세그먼트의 로딩을 피할 수 있지만 폴트의 오버헤드가 실행 시간을 급격하게 증가시킬 수 있다.

✓ 확인문제

28. 동적 링킹 시스템이 직면하고 있는 주소 지정의 어려움을 설명하라.
29. 위치 독립적 코드란 무엇인가? 이는 어디에 유용한가? 이를 생성하기 위해 컴파일러가 따라야 하는 특별한 주의 사항은 무엇인가?
30. 동적 링킹을 사용하는 프로그램에서 gp(전역 포인터) 레지스터가 가지는 의미를 설명하라.
31. 링크 세그먼트의 목적은 무엇인가?
32. 게으른 동적 링킹이란 무엇이며 그 목적은 무엇인가? 이는 어떻게 동작하는가?