08장 서브루틴과 제어 추상화

【8.2.2】 사례 연구: MIPS상의 C와 x86상의 파스칼

스택 관리를 좀 더 상세하게 만들기 위해 두 가지 사례 연구를 제시한다. 하나는 단순한 RISC 기계(MIPS)상의 간단한 언어(C)에 대한 것이고, CISC 기계(x86)상의 중첩된 서브루틴을 사용하는 언어(파스칼)에 대한 것이다.

MIPS상의 SGI C

MIPS 구조의 개관을 ©(심화학습에 있는) 5.4.4절에서 발견할 수 있다. 그 절에서 언급했던 것처럼 레지스터 r31(ra라고 알려짐)은 서브루틴 호출(jal – jump and link) 명령어에서 복귀 주소를 받기 위해 하드웨어에 의해 특별한 용도로 쓰인다. 추가적으로 스택 포인터로서 사용을 위한 규정은 레지스터 r29(sp라고 알려짐)를 예약해놓았다. 그리고 있다면 프레임 포인터를 위한 규정이 레지스터 r30(fp로 알려짐)을 예약했다. 여기서 보여준 자세한 사항은 최적화 수준 −O2로 64비트 코드를 만드는 SGI MIPSpro C 컴파일러의 버전 7.3.1.3m에 부합한다.

예 8.60

SGI MIPSpro C의 호출 순서

전형적인 MIPSpro의 스택 프레임은 ©(심화학습에 있는) 그림 8.8에 나타난다. sp는 스택에서 최근에 사용된 위치를 가리킨다(MIPS에 대한 일부를 포함한 대다수 다른 컴파일러가 sp를 사용하지 않은 첫 위치를 가리키는 것을 알아두자). 스택 내에 있는 모든 객체의 크기를 C에서 컴파일 시에 알기 때문에 별개의 프레임 포인터를 엄격하게 필요로 하지는 않고 MIPSpro 컴파일러는 그것 없이 동작한다. 그것은 현재 스택 프레임 내의 모든 것에 대해 sp로부터의 변위 기법 오프셋을 사용한다. 주요 예외는 인자나 로컬 변수가 매우 많아서 변위 주소 지정의 범위를 넘어서는 서브루틴에서 발생한다. 이것을 위해 컴파일러는 fp를 사용한다.

인자 전달 규정: 다음 루틴에 전달하는 것의 과정에서 인자를 프레임의 꼭대기에 모으고 sp로부터의 오프셋을 통해 항상 접근한다. 첫 8개의 인자를 유형에 따라 정수 레지스터 r4~r11이나 부동소수점 레지스터 f12~f19로 전달한다. 추가적인 인자를 스택으로 전달한다. 레코드 인자(struct)를 묵시적으로 64비트 "덩어리"로 나누고 이를 정수처럼 전달한다. 크기가 큰 struct는 일부분을 레지스터에, 일부분을 스택을 통해 전달한다.

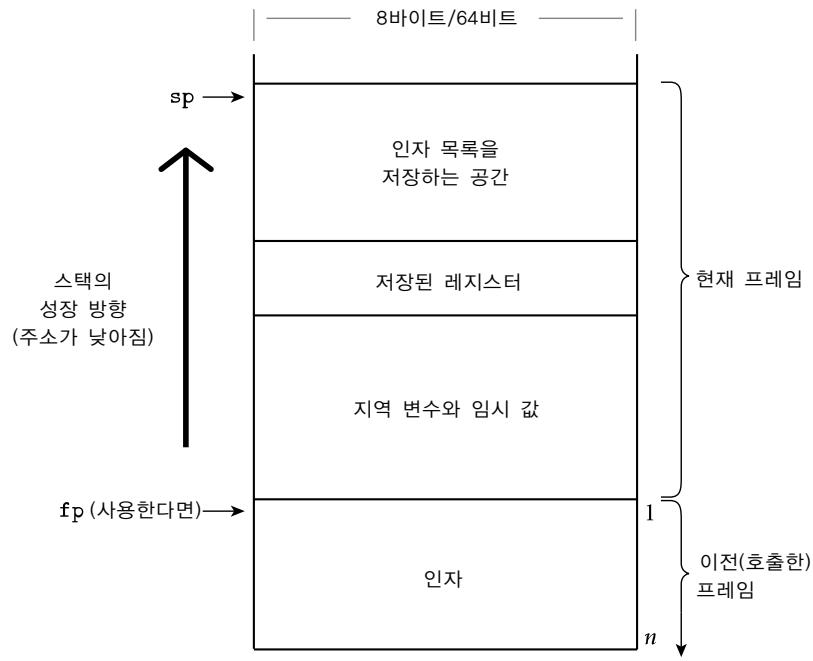

그림 8.8 | 64비트 방식으로 실행하는 SGI MIPSpro C 컴파일러를 위한 서브루틴 호출 스택의 배치. 그림 8.2처럼 더 낮은 주소는 페이지의 위쪽 방향이다.

책에서 언급한 것처럼 레지스터로 전달하든 그렇지 않든 모든 인자를 위한 스택에 공간을 지정한다. 사실 각 서브루틴은 이미 레지스터로 불러온 그 인자의 일부와 메모리에 "한물간" 값을 가진 채로 시작된다. 이것은 상황의 일반적인 상태다. 최적화하는 컴파일러는 가능하면 레지스터 내의 값을 유지한다. 그것은 레지스터가 고갈되거나 메모리 내의 값을 직접적으로 접근하려 할 때(예를 들어 포인터, 참조 매개변수나 중첩된 서브루틴의 활동을 통해) 값을 메모리에 "내보낸다". fp가 있다면 그것은 첫 번째(가장 꼭대기에 있는) 인자를 가리킨다.

어떤 호출된 루틴에라도 전달할 수 있는 인자 목록을 저장하기에 충분히 크게 프레임의 꼭대기에 있는 인자 저장 영역을 설계한다. 이 규정은 특정 경우에 약간의 공간을 낭비할지도 모른다. 하지만 말 그대로 인자를 "밀어 넣을" 필요가 없음을 의미한다. 즉, 인자를 스택 내에 위치시킬 때 sp는 변하지 않는다.

중첩된 서브루틴을 사용하는 언어(물론 C는 그 중 하나가 아니다)에서 MIPS 컴파일러는 일반적으로 정적 체인을 전달하기 위해 레지스터 r2를 사용한다. 모든 언어에서 (유형에 따라) 레지스터 r2와 f0을 함수로부터의 스칼라 값을 반환하기 사용한다. 유형 long double의 값을 레지스터 <f0, f2>으로 반환한다. 128비트 크기인 레코드 값(struct)을 <r2, r3>로 반환한다. 크기가 더 큰 struct를 위해 컴파일러는 반환 값을 위치시켜야 하는 주소의 값을 가진 숨겨진 첫 인자를 r4에 전달한다. 반환 값을 즉시 변수에 대입하면(예를 들어 x = foo()) 호출자는 간단히 변수의 주소를 전달할 수 있다. 값을 다른 서브루틴에 넘겨야 하는 경우 호출자는 자신의 인자 저장 영역 내에 적절한 주소를 전달할 수 있다(반환 값을 이 공간에 쓰는 것은 함수 자신의 인자의 반환을 소실시킬 것이다. 하지만 그것은 훌륭하다. 이 시점에서 더 이상 필요로 하지 않을 것이다). 마지막으로 이 관용구를 가끔 보지 못하지만(그리고 대부분 언어가 그것을 지원하지 않지만) C는 호출자가 한 함수(예를 들어 x = foo().a+y;)의 반환 값에서 직접적으로 어떤 항목을 추출하게 한다. 이 경우 호출자는 스택 프레임의 "지역 변수와 임시 값" 부분 내부의 임시적인 위치의 주소를 전달해야 한다.

호출 순서의 상세 사항: MIPSpro 스택을 유지하기 위한 호출 순서는 다음과 같다. 호출자는

1. (프레임의 "지역 변수와 임시 값" 부분에) 여전히 필요한 값을 가진 호출자 저장 레지스터 모두를 저장한다.
2. 8개의 스칼라 인자(혹은 구조체의 "덩어리")를 레지스터에 넣는다.
3. 현재 프레임의 꼭대기에 남아있는 인자를 인자 저장 영역에 넣는다.
4. 레지스터 ra에 반환 주소를 넣고 그리고 목적 주소로 건너뛰는 jal 명령어를 수행한다.[1]

호출자 저장 레지스터는 r2~r15, r24, r25와 f0~f23으로 구성된다. 중첩된 서브루틴을 사용하는 언어에서 호출자는 jal을 수행하기 전에 정적 연결을 즉시 레지스터 r2에 위치시킬 것이다.

그것의 프롤로그에서 피호출자는

1. 프레임 크기(©심화학습에 있는 그림 8.8에서 첫 인자와 sp 간의 거리)를 뺀다.
2. 프레임 포인터를 사용한다고 하면 그것의 값을 이용 가능한 임시 레지스터(일반적으로 r2)에 복사하고 나서 프레임 크기를 sp에 더한다. 그 결과 값을 fp에 넣는다 (이는 효과적으로 이전 sp를 fp로 이동시킨다. add는 move 만큼 빨라서 sp를 처음 갱신하는 데 어떠한 해로움도 없음을 기억하자).

주1. MIPS의 모든 분기 명령어와 마찬가지로 jal은 구조적으로 명백한 분기 지연 공간을 가진다. ISA의 MIPS II 버전에서 불러오기 지연 공간을 제거했다. 최근 MIPS 프로세서 모두는 완전히 서로 맞물린다.

3. sp나 이용 가능하다면 fp를 변위 방식 주소 지정의 기저로서 이용해 필요한 레지스터를 모두 새롭게 할당된 프레임의 중앙에 저장한다.

저장된 레지스터는 (a) 복귀하기 전에 변경될 수도 있는 값을 가진 피호출자 저장 임시 레지스터(r16~r23과 f24~f31)와 (b) 현재 루틴이 리프가 아니거나 그것이 ra를 추가적인 임시 레지스터로 사용하면 ra, 그리고 (c) 2단계에서 이전 fp를 포함하는 임시 레지스터나 현재 루틴이 프레임 포인터를 필요로 하지 않지만 추가적인 임시 레지스터로 fp를 사용하면 fp 자체도 포함한다.

에필로그에서 복귀하기 직전에 피호출자는

1. (있다면) 함수 반환 값을 r2, r3, f0, f2나 메모리에 적당히 넣는다.
2. (있다면) 저장된 레지스터를 sp나 이용 가능하다면 fp를 변위 방식 주소 지정의 기저로서 이용해 복구한다.
3. fp를 sp에 이동시키거나 sp에 프레임 크기를 더해 프레임을 해제한다.
4. (있다면) 2단계의 임시 레지스터의 값을 fp로 옮긴다.
5. jr ra 명령어를 수행한다(레지스터 ra에 있는 주소로 건너뛴다).

마지막으로 적합하다면 호출자는 반환 값을 필요한 어느 곳에라도 옮긴다. 호출자 저장 레지스터를 필요할 때까지 시간상으로 게으르게 복구한다.

gdb와 dbx와 같은 심볼릭 디버거의 사용을 지원하기 위해 컴파일러는 정보를 객체 파일 심볼 테이블에 위치시키는 다양한 어셈블러 의사 op를 생성한다. 각 서브루틴에 대해 이 정보는 시작하고 끝나는 주소, 스택 프레임의 크기, 어떤 레지스터(보통 sp나 fp)가 지역 객체에 대한 기저인지에 대한 표시, 어떤 레지스터(있다면 보통 ra)가 복귀 주소를 가지고 있는지에 대한 표시, 어떤 레지스터를 저장했는지에 대한 목록 등을 포함한다.

x86상에서의 GNU 파스칼

CISC와 RISC 기계 간의 차이점을 설명하기 위해 두 번째 사례 연구는 여전히 세계에서 가장 인기 있는 명령어 집합 구조인 x86을 고려한다(프로세서에 대한 개관은 ©심화학습에 있는 5.4.4절에 있다). 중첩된 서브루틴과 클로저의 처리를 설명하기 위해 파스칼 컴파일러, 즉 GNU 파스칼 컴파일러의 3.2.2 버전인 gpc를 설명한다(에이다 컴파일러[예를 들어 GNU의 gnat]은 이 특징을 비슷한 방법으로 다룬다. 하지만 에이다의 많은 추가적 기능은 이 사례 연구를 훨씬 더 복잡하게 만들 것이다).

x86의 현대 구현에서 일반적인 store 명령어는 push를 사용해서 가능한 것보다 파이프라인을 더 잘 사용할지도 모른다. 그러므로 (gpc가 기반하는) gcc를 포함한 대부분

의 x86을 위한 현대 컴파일러는 앞선 사례 연구의 것과 비슷한 인자 저장 영역을 이용한다. 기본적으로 gpc와 gcc는 부분적으로는 다른 구조와 언어와의 일관성을 목적으로(gcc는 아주 이식성이 뛰어나다) 하고 부분적으로는 공간을 현재 스택 프레임 내에 동적으로 할당하는 라이브러리 메커니즘의 구현을 단순화하기 위해 여전히 별개의 프레임 포인터를 사용한다(©심화학습에 있는 연습문제 8.35를 보라).

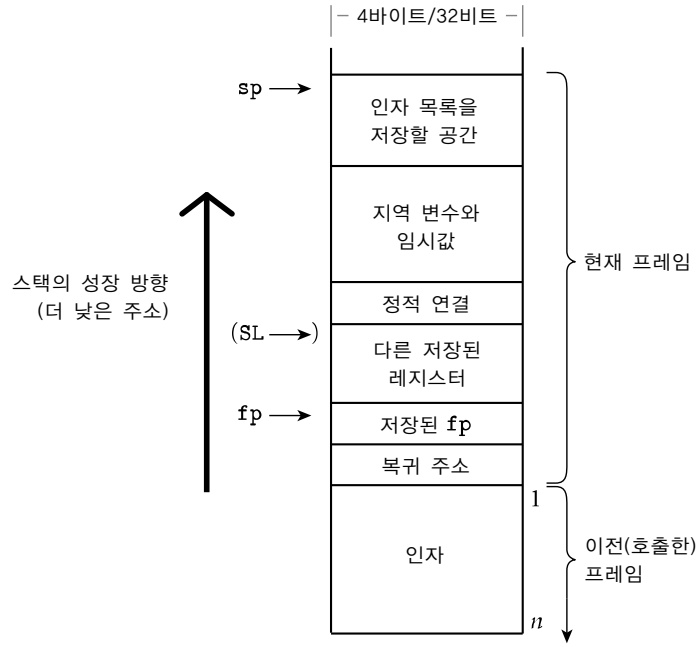

그림 8.9 | GNU 파스칼 컴파일러, gpc를 위한 서브루틴 호출 스택의 배치. 복귀 주소와 저장된 fp는 모든 프레임에서 존재한다. 프레임의 다른 부분은 모두 선택적이다. 즉, 그것은 현재 서브루틴이 필요로 할 때만 존재한다. x86의 용어로 하면 sp를 esp로 이름 붙이고 fp는 ebp(extended base pointer)다. SL은 이 서브루틴 안에 바로 중첩된 서브루틴의 정적 연결이 참조할 것인 위치를 표시한다.

서브루틴 호출에 대한 특별한 명령어는 CISC 기계마다 완전히 다르다. 오늘날 x86에서 가장 빈번히 사용하는 것은 상대적으로 간단하다. `call` 명령어는 복귀 주소를 스택에 푸쉬하고 sp를 갱신한 후 호출된 루틴으로 분기한다. `ret` 명령어는 스택에서 복귀 주소를 꺼내고 sp를 갱신한 다음 호출자로 돌아간다. 현대 컴파일러는 후진 호환성을 위해 남겨진 일부 추가적인 명령어를 생성하지 않는다. 그것을 명시적인 디스플레이를 사용하고 인자 저장 영역이 없는 호출 순서를 위해 설계했기 때문이거나 더 간단한 명령어의 동일한 순서뿐만 아니라 파이프라인을 하지 않기 때문이다.

예 8.61

GNU 파스칼의 x86 호출 순서

인자 전달 규정: ©(심화학습에 있는) 그림 8.9는 x86을 위한 스택 프레임을 보여준다. 이전 사례 연구처럼 sp는 스택상에서 가장 마지막에 사용한 위치를 가리킨다. 다른 루틴에 전달하는 과정에서 sp로부터의 오프셋을 통해 인자에 접근한다. 그 밖의 모든 것은 fp로부터의 오프셋을 통해 접근한다. 모든 인자를 스택으로 전달한다. 정적 연결

을 전달하기 위해 레지스터 rcx를 사용한다. 그 연결은 어휘적으로 둘러싸인 루틴의 프레임에서 마지막으로 저장한, 즉 그 루틴 자신의 정적 연결이 있다면 그것 바로 아래의 레지스터(다른 것이 없다면 저장된 fp)를 가리킨다.

함수는 레지스터 eax에 정수나 포인터 값을 반환한다. 부동소수점 값을 부동소수점 레지스터의 첫 번째 것인 st(0)에 반환한다. 만들어진 유형(레코드, 배열이나 집합)의 값을 반환하는 함수에 대해 컴파일러는 반환 값을 위치시켜야 하는 주소의 값을 가진 숨겨진 첫 인자를 스택으로 전달한다.

호출 순서의 상세 사항 gpc 스택을 유지하기 위한 호출 순서는 다음과 같다. 호출자는

1. 여전히 필요한 값을 가진 호출자 저장 레지스터를 (그 프레임의 "지역 변수와 임시 값" 부분에) 저장한다.
2. 현재 프레임의 꼭대기의 저장 영역에 인자를 넣는다.
3. 레지스터 ecx에 정적 연결을 위치시킨다.
4. call 명령어를 실행한다.

호출자 저장 레지스터는 eax, edx와 ecx로 구성된다. 이 중 어느 것도 나중에 필요한 값을 포함하고 있지 않다면 1단계를 넘어간다. 서브루틴이 매개변수를 가지고 있지 않으면 2단계를 넘어간다. 어휘적 중첩의 가장 외부 수준에 서브루틴을 선언한다면 3단계를 넘어간다. call 명령어는 복귀 주소를 푸쉬하고 그 서브루틴으로 건너뛴다.

프롤로그에 피호출자는

1. 묵시적으로 sp를 4(1 워드)씩 감소시키면서 fp를 스택에 푸쉬한다.
2. sp를 fp에 복사해 현재 프레임에 대한 프레임 포인터를 정한다.
3. 현재 루틴이 덮어쓰는 피호출자 저장 레지스터를 푸쉬한다.
4. 리프가 아니면 정적 연결(ecx)을 푸쉬한다.
5. sp에서 프레임 크기의 나머지를 뺀다.

피호출자 저장 레지스터는 ebx, esi와 edi다. 1단계와 2단계는 레지스터 esp와 ebp (각기 sp와 fp)를 저장한다. 이 단계의 일부를 위한 명령어를 컴파일러의 코드 개선기는 동일한 순서로 대체하고 명령어 스케줄러는 서브루틴의 나머지와 섞는다. 특히 4단계에서 sp를 뺀 값을 충분히 크게 만들어서 피호출자 저장 레지스터를 수용한다면 3단계의 push를 4단계 뒤로 옮기고 fp와 연관된 저장하기로 대체시킬 수도 있다.

에필로그에서 피호출자는

1. 반환 값을 저장한다.

2. 피호출자 저장 레지스터를 복구한다.

3. fp를 sp로 복사하고 프레임을 해제한다.

4. fp를 스택에서 팝한다.

5. 복귀한다.

마지막으로 이전 사례 연구와 마찬가지로 호출자는 반환 값이 레지스터 안에 있다면 그것을 필요로 하는 모든 곳에 이동시킨다. 그것은 시간이 흐르면서 게으르게 호출자 저장 레지스터를 복구한다.

예 8.62

서브루틴 클로저 트램펄린

파스칼이 서브루틴이 중첩되게 하기 때문에 3.5.1절에서 설명한 것처럼 매개변수로서 P에서 Q로 전달되는 서브루틴 S를 클로저로 표현해야 한다. 대다수의 컴파일러에서 클로저는 S를 호출할 때 사용해야 하는 S의 주소와 정적 연결을 포함하는 자료 구조다. 하지만 gpc에서 클로저는 **트램펄린**(trampoline)이라 하는 x86 코드 순서를 포함한다. 이는 일반적으로 적절한 정적 연결을 ecx로 불러와서 S의 도입부로 건너뛰기 위한 2개의 명령어다. 트램펄린은 P의 활성화 레코드의 "지역 변수와 임시 값" 부분에 위치한다. 그것의 주소를 Q로 전달한다. 실행 시간에 그 클로저를 '해석하지" 않고 Q는 사실상 그것을 호출한다. 이 방법의 이점 중 하나는 매개변수로 전달된 C 함수가 코드 주소인 gcc와의 상호호환성이다. 사실 S를 가장 외부의 어휘적 중첩 수준에 선언한다면 gpc는 역시 보통의 코드 주소를 전달할 수 있다. 이 경우 트램펄린이 필요하지 않다.

설계와 구현

스택 내 코드를 실행하기

트램펄린 기반 클로저의 단점은 스택 내 코드를 실행할 필요다. 많은 기계와 운영체제는 적어도 두 중요한 이유 때문에 그런 실행을 허가하지 않는다. 첫째, 5.1절에서 언급한 것처럼 현대 마이크로프로세서는 일반적으로 빠른 병행적 접근을 위해 별개의 명령어 캐시와 자료 캐시를 갖는다. 한 프로세서가 메모리의 동일한 영역에 쓰고 실행하게 하는 것은 이 캐시를 상호간에 모순이 없게 유지해야 함, 즉 중대한 하드웨어 복잡성을 도입할 어떤 작업을 의미한다. 둘째, 대다수의 컴퓨터 보안의 틈은 소위 버퍼 오버플로우 공격을 포함한다. 여기서 침입자는 배열 범위 검사의 결여를 이용해 스택에 코드를 써넣는다. 그러면 현재 서브루틴이 복귀할 때 그것을 실행할 것이다. 그런 공격은 쓸 수 있는 자료도 역시 실행할 수 있는 기계에서만 가능하다.

✓ 확인문제

52. 한 개나 두 개의 사례 연구에 대해 하드웨어가 호출 순서와 스택 배치의 어떤 면을 알려주는지, 그리고 소프트웨어 규정의 문제가 무엇인지를 설명하라.

53. MIPS상에서 일부 컴파일러는 `sp`가 스택에서 가장 마지막으로 사용된 워드를 가리키게 한다. 하지만 다른 컴파일러는 그것이 첫 사용되지 않는 워드를 가리키게 한다. x86에서는 모든 컴파일러는 그것이 가장 마지막에 사용된 워드를 가리키게 한다. 이 차이점은 왜 그런가?

54. 왜 MIPSpro 컴파일러와 `gpc`는 호출자 저장 레지스터를 호출이 끝난 직후에 복구하지 않는가?

55. 서브루틴 호출 트램폴린이란 무엇인가? 3.5.1절에서 설명한 클로저의 일반적인 구현과는 어떻게 다른가? 두 방법의 상대적인 이점은 무엇인가?