System Programming
숙제 5

인천대학교

시스템 프로그래밍 | 지도교수 박문주

송병준 | 201701562

0.목차

System Programming

숙제 5

송병준
2019년 10월 9일

1. 과제

Computer System, A Programmer's Perspective 5장 연습 문제 풀이
- 3.60
- 3.61
- 3.63

2. 환경

운영체제: macOS 10.14.6 (Darwin Kernel Version 18.7.0 x86_64)
프로세서: Intel Core i5
편집기: vim 8.0.1365
빌드: Apple LLVM version 10.0.1 (clang-1001.0.46.4)

3. 문제 풀이

- **3.60**

다음과 같은 어셈블리 코드가 있다.

```
    long loop(long x, int n)
    x in %rdi, n in %esi
1   loop:
2     movl    %esi, %ecx
3     movl    $1, %edx
4     movl    $0, %eax
5     jmp     .L2
6   .L3:
7     movq    %rdi, %r8
8     andq    %rdx, %r8
9     orq     %r8, %rax
10    salq    %cl, %rdx
11  .L2:
12    testq   %rdx, %rdx
13    jne     .L3
14    rep; ret
```

위 코드는 다음과 같은 형식을 갖는 C 코드를 컴파일해서 생성한 것이다:

```
1  long loop(long x, long n)
2  {
3      long result = _________;
4      long mask;
5      for (mask = _________; mask  _________ ; mask = _________ ) {
6          result |= _________ ;
7      }
8      return result;
9  }
```

해야 할 일은 생성된 어셈블리 코드와 동일한 프로그램이 되도록 C 코드의 빠진 부분을 채우는 것이다. 함수의 결과가 레지스터 %rax에 저장된다는 것을 기억하라. 레지스터와 프로그램 변수들 간의 일정한 매핑을 형성하기 위해 루프 이전, 중간, 이후에 어셈블리 코드를 검토하는 것이 좋다.

A. 어떤 레지스터가 x, n, result, mask 값을 저장하는가?
B. result와 mask의 초기값은 얼마인가?
C. mask의 테스트 조건은 무엇인가?
D. 어떻게 mask가 갱신되는가?
E. 어떻게 result가 갱신되는가?
F. C 코드의 빈칸을 모두 채우시오.

A.
x와 n은 함수 시작시 각각 rdi와 esi에 저장되어 전달된다. 이제 result와 mask가 각각 어디에 저장되는지만 알면 된다.

함수가 시작될 때에 세 번의 대입이 일어난다. 첫번째는 n의 값을 rcx에 저장하는 것, 두번째는 edx에 1을 넣는 것, 세번째는 rax에 0을 넣는 것이다. 이중 하나는 result이다. for 본문 안에 힌트가 있다. 식의 결과와 result 사이에 or 연산을 취해 다시 result에 저장하는데, 어셈블리에서 orq의 destination으로 등장하는 레지스터는 rax뿐이다. 따라서 result는 rax에 저장된다.

mask는 함수 실행시 조건 테스트로 넘어가기 전에 대입이 이루어진다. jump 전 대입이 이루어지는 레지스터는 ecx, edx, eax밖에 없는데, ecx는 n을, eax는 result를 저장한다. 따라서 mask는 edx에 저장된다. 정리하면 다음과 같다.

x	%rdi
n	%esi
result	%rax
mask	%rdx

B.
초기값을 알기 위해서는 해당 변수에 가장 먼저 대입되는 값을 보면 된다. result를 저장하는 rax에 처음으로 대입되는 값은 0이다. mask를 저장하는 rdx에 처음 대입되는 값은 1이다. 따라서 정리하면 다음과 같다.

result	0
mask	1

C.
조건 테스트 부분인 L2에서는 rdx와 rdx를 test한다. 만약 ZERO FLAG이 0이 되면 반복문 안으로 점프한다. x86의 test는 두 operand 사이에 비트 and 연산을 취해 SF, ZF, PF를 설정한다.

rdx와 rdx에 and를 취한 결과는 rdx이다. 즉 rdx가 0이 아닐 때에만 jne에 의해 본문인 L2 안으로 분기가 일어난다. 따라서 테스트 조건문은 다음과 같다.

```
mask != 0
```

D.
mask의 갱신은 for 본문 수행 후에 이루어진다. 주어진 어셈블리에서는 n의 하위 8비트가 담긴 cl의 값만큼 mask에 왼쪽 산술 shift를 취한다. 정리하면 다음과 같다.

```
mask <<= (n & 0xFF)
```

이때, rcx나 ecx 대신 cl을 사용한 의미를 살리기 위해 (n & 0xFF)를 사용했지만 어차피 255비트 shift는 불가능하다. 이는 컴파일러 최적화로 보인다. 32비트 정수 n에 취할 수 있는 shift는 최대 31로, 이를 나타내기 위해서는 다섯 비트밖에 필요하지 않다. 5비트 이상인 가장 작은 레지스터 접근 단위는 8비트이기 때문에 하위 8비트 접근 방식을 채택한 것으로 보인다.

즉, 컴파일러는 shift 연산자의 operand로 정수가 들어오면 이중 하위 8비트만 사용할 것이다. 따라서 아래 두 코드는 같은 동작을 한다.

```
mask <<= (n & 0xFF)
mask <<= n
```

단, 이는 operand가 31 이하일 경우에만 해당된다. 그렇지 않은 경우의 동작은 정의되지 않았다.

E.
result에 or 연산을 취할 operand는 x와 mask 사이에 and 연산을 취함으로써 만들어진다. 따라서 정리하면 다음과 같다.

```
result |= (x & mask);
```

F.
주어진 코드를 완성하면 다음과 같다.

```
long loop(long x, int n) {
	long result = 0;
	long mask;
	for (mask = 1; mask != 0; mask <<= n) {
		result |= (x & mask);
	}
	return result;
}
```

- **3.61**

3.6.6절에서 다음 코드를 조건부 데이터 이동 목적으로 사용하는 가능성을 조사하였다.

```
long cread(long *xp) {
    return (xp ? *xp : 0);
}
```

조건부 move 인스트럭션을 사용하여 시험적으로 구현하였지만, null 주소로부터 읽기를 할 수 없었기 때문에 이것은 실효성이 없다고 결론지었다.

조건부 데이터 이동을 사용하기 위해 컴파일될 수 있다는 점을 제외하고 cread와 동일한 C 함수 cread_alt를 작성하라. 컴파일될 때 생성된 코드는 jump 인스트럭션이 아닌 조건부 move 인스트럭션을 사용해야 한다.

주어진 cread 함수가 조건부 데이터 이동 (cmove)을 통해 구현된다면, 조건과 관계없이 xp를 역참조해야 하기 때문에 *xp를 반환하거나 동작을 멈출 것이다. 그렇다면 순서를 조금만 바꾸어 보자.

먼저 결과를 0으로 놓은 후, xp가 0이 아니면 이를 참조하여 결과에 대입하는 것이다.

```
long cread_alt(long *xp) {
       return (!xp ? 0 : *xp);
}
```

LLVM Clang x86_64 환경에서 cmove를 사용하도록 하는 것은 실패했다. 하지만 cread가 위와 같이 cmove를 사용해 구현되는 환경이라면 cread_alt 또한 그럴 것이다.

- **3.63**

이 문제는 역어셈블한 기계어 코드에서 swich 문장을 역엔지니어링 reverse engineer할 수 있는 기회를 제공한다. 다음 프로시저에서 swich문의 본체를 생략하였다:

```
1  long switch_prob(long x, long n) {
2      long result = x;
3      switch(n) {
4          /* Fill in code here */
5
6      }
7      return result;
8  }
```

그림 3.53은 이 프로시저에 대한 역어셈블한 기계어 코드를 보여준다.

점프 테이블은 메모리의 다른 영역에 위치한다. 5번 줄의 간접 점프로부터 점프 테이블이 주소 0x4006f8에서 시작된다는 것을 알 수 있다. GDB 디버거를 사용하여 점프 테이블을 구성하는 메모리의 6개 8바이트 워드를 명령 x/6gx 0x4006f8을 사용해서 조사하였다. GDB가 다음과 같은 출력을 나타냈다:

```
    long switch_prob(long x, long n)
    x in %rdi, n in %rsi
1   0000000000400590 <switch_prob>:
2     400590:  48 83 ee 3c           sub    $0x3c,%rsi
3     400594:  48 83 fe 05           cmp    $0x5,%rsi
4     400598:  77 29                 ja     4005c3 <switch_prob+0x33>
5     40059a:  ff 24 f5 f8 06 40 00  jmpq   *0x4006f8(,%rsi,8)
6     4005a1:  48 8d 04 fd 00 00 00  lea    0x0(,%rdi,8),%rax
7     4005a8:  00
8     4005a9:  c3                    retq
9     4005aa:  48 89 f8              mov    %rdi,%rax
10    4005ad:  48 c1 f8 03           sar    $0x3,%rax
11    4005b1:  c3                    retq
12    4005b2:  48 89 f8              mov    %rdi,%rax
13    4005b5:  48 c1 e0 04           shl    $0x4,%rax
14    4005b9:  48 29 f8              sub    %rdi,%rax
15    4005bc:  48 89 c7              mov    %rax,%rdi
16    4005bf:  48 0f af ff           imul   %rdi,%rdi
17    4005c3:  48 8d 47 4b           lea    0x4b(%rdi),%rax
18    4005c7:  c3                    retq
```

그림 3.53 연습문제 3.63을 위한 역어셈블한 코드

```
0x4006f8:    0x00000000004005a1    0x00000000004005c3
0x400708:    0x00000000004005a1    0x00000000004005aa
0x400718:    0x00000000004005b2    0x00000000004005bf
```

기계어 코드와 동일한 동작을 갖는 C 코드의 swith 문장 본체를 채우시오.

함수 시작 초기에는 switch문으로의 jump가 일어나기도 전에 rsi를 비교해서 마지막으로 보내버린다. 이는 점프 테이블 참조가 필요하지 않을 정도로 확실하다는 뜻이다. 이렇게 확실한 경우는 n이 점프테이블 인덱스의 범위를 벗어나는 경우밖에 없다.

어셈블리를 보면 $5 < (n - 60)$인 경우에 default case로 jump가 일어난다는 것을 알 수 있다. 즉 모든 case는 **65 이하**이다. 또한 $n - 60$을 점프 테이블의 인덱스로 직접 사용하는 것으로 보아 $n - 60$이 0 이상의 값을 가질 것으로 기대된다고—컴파일러에 의해— 볼 수 있다. 즉 모든 case는 **60 이상**이다.

문제에는 점프 테이블의 원소가 6개라고 명시되어 있다. 테이블의 각 행은 n이 각각 60, 61, 62, 63, 64, 65인 경우와 대응된다. 테이블 각 행에 대응되는 n값과 주소, 그리고 주소가 가리키는 코드를 정리하면 다음과 같다.

n	address	content
60	0x4005a1	`result = x * 8; break;`
61	0x4005c3	`result = x + 75; break`
62	0x4005a1	`result = x * 8; break;`
63	0x4005aa	`result = x >> 3; break`
64	0x4005b2	`result = (x << 4) - x;` `x = (result -= x);`
65	0x4005bf	`x = x * x;`

이때 60 이상 65 이하인 모든 n에 대해 case가 정의되어 있는 것은 아니다. 점프 테이블의 두번째 행은 switch 문의 마지막, 즉 default case를 가리킨다.

n이 60인 경우와 n이 62인 경우는 동작이 같다. 둘은 같은 동작을 수행하지만 해당 코드는 하나만 존재한다(같은 주소). 이는 60인 case의 본문이 비어있고 break 없이 바로 그 다음 62인 case로 이어지거나, 62인 case의 본문이 비어있고 바로 다음 60인 case로 이어지는 경우이다. 이때 default case를 제외하면 점프 테이블의 모든 행은 가리키는 주소에 대해 오름차순으로 정렬되어 있기 때문에 60 다음에 62 case가 위치하도록 하여 60의 본문을 비우는 것이 자연스럽다.

64와 65 case의 본문에는 break에 해당하는 retq가 없다. 따라서 64 case로 진입하면 65 case를 지나 default case까지 도달한다. 이와 마찬가지로 65 case로 진입하면 default case까지 도달한다.

위의 어셈블리와 같은 동작을 하는 코드를 작성하면 다음과 같다.

```
long switch_prob(long x, long n) {
	long result = x;

	switch(n) {
		case 60:

		case 62:
			result = x * 8;
			break;

		case 63:
			result = x >> 3;
			break;

		case 64:
			x = (result = x << 4) - x;

		case 65:
			x = x * x;

		default:
			result = x + 75;
			break;
	}
	return result;
}
```

이때 각 case의 순서가 매우 중요하다. case 값들은 정렬되어있지 않아도 되지만 case의 순서가 바뀐다면 동작이 바뀔 수 있다.