System Programming
숙제 4

인천대학교

시스템 프로그래밍 | 지도교수 박문주

송병준 | 201701562

0.목차

System Programming

숙제 4

송병준
2019년 10월 3일

1. 과제

Computer System, A Programmer's Perspective 3장 연습 문제 풀이
- 3.58
- 3.59

3.58은 C언어 코드 작성 후 컴파일, assembly 코드 생성하여 제출.

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

- **3.58**

다음과 같은 정의를 갖는 함수에 대해

```
long decode2(long x, long y, long z);
```

GCC는 다음의 어셈블리 코드를 생성하였다.

```
1   decode2:
2     subq    %rdx, %rsi
3     imulq   %rsi, %rdi
4     movq    %rsi, %rax
5     salq    $63, %rax
6     sarq    $63, %rax
7     xorq    %rdi, %rax
8     ret
```

매개변수 x, y, z는 레지스터 %rdi, %rsi, %rdx로 전달된다. 코드는 레지스터 %rax에 리턴 값을 저장한다.

위 어셈블리 코드와 동일한 효과를 갖는 decode2를 C로 작성하시오.

어셈블리 코드의 각 줄을 해석해본다.

2번 줄에서는 rsi 레지스터의 값에서 rdx 레지스터의 값을 뺀 결과를 다시 rsi 레지스터에 저장하였다. rdx는 매개변수 z, rsi는 매개변수 y로 전달되므로 해당 연산을 C 코드로 표현하면 y = y − z일 것이다.

3번 줄에서는 rdi에 rsi를 곱해 이를 다시 rdi에 집어넣었다. 이때 rdi는 초반 x의 값을 그대로 유지하고 있지만 rsi에는 위에서 z를 뺀 y의 값이 저장되어 있다. 이 연산은 x = x * z로 나타낼 수 있다.

4번 줄에서는 rsi에서 rax로 가는 대입 연산을 취한다. rax는 이때 처음 사용되었으므로, 새로운 long 타입 임시 변수가 생겼음을 알 수 있다. 이렇게 나타낼 수 있다. long temp = z;

5번줄에서는 위에서 생긴 temp 변수에 좌측 산술 쉬프트를 63번 취한다. 이는 temp = temp 〈〈 63으로 나타내어진다.

6번 줄에서는 temp 변수에 우측 산술 쉬프트를 63 취한다. 산술 쉬프트로 명시된 것을 보니 temp는 signed 타입인 것을 어느 정도 유추할 수 있다. C 코드로는 temp = temp 〉〉 63으로 나타내어진다.ㄱ

7번 줄에서는 rax와 rdi 사이에 xor 연산을 취해 이를 다시 rax에 저장한다. C 코드로는 temp = temp ^ x;이다.

마지막 줄에서는 함수를 끝내고 반환한다. 이때 마지막 연산 결과는 rax 레지스터에 저장되어 있어 반환값으로 사용된다.

다행히 인자로 넘어온 rdi, rsi, rdx의 용도가 바뀌지 않아 분석이 어렵지는 않았다. 위에서 도출해낸 C 코드를 기반으로 해당 함수를 다시 작성해보면 다음과 같다.

```
long decode2(long x, long y, long z) {
	long temp;

	y = y - z;
	x = x * y;
	temp = y;
	temp = temp << 63;
	temp = temp >> 63;
	temp = temp ^ x;

	return temp;
}
```

이 함수가 위의 decode2와 같은 행동을 하는지 확인하기 위해 어셈블리를 추출해낸다.

```
$ gcc -o chapter3.o -c chapter3.c
$ objdump -d chapter3.o
```

결과는 다음과 같다.

```
obj:   file format Mach-O 64-bit x86-64

Disassembly of section __TEXT,__text:
_decode2:
       0:    55            pushq  %rbp
       1:    48 89 e5      movq   %rsp, %rbp
       4:    48 89 7d f8   movq   %rdi, -8(%rbp)
       8:    48 89 75 f0   movq   %rsi, -16(%rbp)
       c:    48 89 55 e8   movq   %rdx, -24(%rbp)
      10:    48 8b 55 f0   movq   -16(%rbp), %rdx
      14:    48 2b 55 e8   subq   -24(%rbp), %rdx
      18:    48 89 55 f0   movq   %rdx, -16(%rbp)
      1c:    48 8b 55 f8   movq   -8(%rbp), %rdx
      20:    48 0f af 55   f0     imulq  -16(%rbp), %rdx
      25:    48 89 55 f8   movq   %rdx, -8(%rbp)
      29:    48 8b 55 f0   movq   -16(%rbp), %rdx
      2d:    48 89 55 e0   movq   %rdx, -32(%rbp)
      31:    48 8b 55 e0   movq   -32(%rbp), %rdx
      35:    48 c1 e2 3f   shlq   $63, %rdx
      39:    48 89 55 e0   movq   %rdx, -32(%rbp)
      3d:    48 8b 55 e0   movq   -32(%rbp), %rdx
      41:    48 c1 fa 3f   sarq   $63, %rdx
      45:    48 89 55 e0   movq   %rdx, -32(%rbp)
      49:    48 8b 55 e0   movq   -32(%rbp), %rdx
      4d:    48 33 55 f8   xorq   -8(%rbp), %rdx
      51:    48 89 55 e0   movq   %rdx, -32(%rbp)
      55:    48 8b 45 e0   movq   -32(%rbp), %rax
      59:    5d            popq   %rbp
      5a:    c3            retq
```

음…
대부분의 공간을 차지하는 것이 %rbp에다가 무언가를 대입하는 코드이다. 오프셋을 8바이트씩 하여 rdi, rsi, rdx를 대입하는 것을 보니 인자를 스택에다가 쌓아두고 처리하는 것으로 보인다. 왜 레지스터를 놓아두고 굳이 그걸 다시 스택에 집어넣어서 쓰는 지 모르겠다. 최적화 옵션을 2로 주어 다시 컴파일 해보았다.

```
Disassembly of section __TEXT,__text:
_decode2:
       0:    55            pushq  %rbp
       1:    48 89 e5      movq   %rsp, %rbp
       4:    48 89 f0      movq   %rsi, %rax
       7:    48 29 d0      subq   %rdx, %rax
       a:    48 0f af f8   imulq  %rax, %rdi
       e:    83 e0 01      andl   $1, %eax
      11:    48 f7 d8      negq   %rax
      14:    48 31 f8      xorq   %rdi, %rax
      17:    5d            popq   %rbp
      18:    c3            retq
```

이제 조금 깨끗한 코드가 나왔다.
첫번째와 두번째 줄에서는 호출자가 사용하던 stack base pointer를 잠시 저장해둔 뒤, 현재의 stack pointer를 base pointer로 사용하기 위해 rbp에 rsp를 대입해준다. 이렇게 하여 새로운 스택 프레임을 사용할 준비가 끝난다. 인자 x ,y ,z는 각각 rdi, rsi, rdx에 담긴다(System V ABI). 세번째 줄에서 처음으로 인자를 사용하는데, y로 넘어온 값을 rax에 대입한다. 네 번째 줄에서 y - z를 수행한다. 이때까지 rax에는 초기의 y에서 초기의 z를 뺀 값이 들어있다. 그 다음 다섯번째 줄에서는 위의 결과에 x의 값을 곱한 결과를 x가 있던 rdi에 대입한다.

이 다음부터는 주어진 어셈블리와 조금 다른 모양을 띤다.
일곱번째 줄에서는 y가 담겨있는 rax의 하위 4바이트에 & 0x01을 적용한다. 그런 다음 여덟번째 줄에서는 rax의 부호를 바꾼다.

어떤 숫자에 우측 쉬프트 63번 후 좌측 쉬프트 63번을 취한 결과는 해당 숫자의 LSB가 1이면 -1, 그렇지 않으면 0일 것이다. 즉 해당 숫자의 LBS * -1을 하면 쉬프트 없이 같은 결과가 나온다. 위의 일곱번째 줄과 여덟번째 줄에서 이를 수행한다.

아홉번째 줄에서는 현재까지의 결과가 담긴 y에 x와의 xor 연산 결과를 저장한다. 그리고 그 다음줄부터는 호출자의 stack base pointer를 복구하고 함수를 반환한다.

- **3.59**

다음의 코드는 두 개의 64비트 부호형 값 x와 y의 128비트 곱을 계산해서 그 결과를 메모리에 저장한다.

```
1   typedef __int128 int128_t;
2
3   void store_prod(int128_t *dest, int64_t x, int64_t y) {
4       *dest = x * (int128_t) y;
5   }
```

GCC는 계산을 구현하기 위해 다음과 같은 어셈블리 코드를 생성하였다:

```
1   store_prod:
2     movq    %rdx, %rax
3     cqto
4     movq    %rsi, %rcx
5     sarq    $63, %rcx
6     imulq   %rax, %rcx
7     imulq   %rsi, %rdx
8     addq    %rdx, %rcx
9     mulq    %rsi
10    addq    %rcx, %rdx
11    movq    %rax, (%rdi)
12    movq    %rdx, 8(%rdi)
13    ret
```

이 코드는 64비트 머신에서 128비트 산술연산을 구현하기 위해 요구된 다중 정밀도 산술연산을 위해 세 개의 곱셈을 이용한다. 곱을 계산하기 위해 사용한 알고리즘을 설명하고, 어셈블리 코드에 주석을 달아서 이 알고리즘이 어떻게 구현되었는지 보이시오. 힌트: x와 y의 인자를 128비트로 확장할 때, 이들은 $x = 2^{64} \cdot x_h + x_l$, $y = 2^{64} \cdot y_h + y_l$로 재작성될 수 있으며, 여기서 x_h, x_l, y_h, y_l은 64비트 값들이다. 유사하게 128비트 곱은 $p = 2^{64} \cdot p_h + p_l$로 작성할 수 있으며, 여기서 p_h와 p_l은 64비트 값들이다. 어떻게 이 코드가 p_h와 p_l을 x_h, x_l, y_h, y_l로 계산할 수 있는지를 보이시오.

두 64비트 부호 정수를 받아 이들 간에 128비트 곱셈을 수행하는 함수이다.
2의 보수 표기법에서 음수의 덧셈은 쉽게 처리할 수 있지만 곱셈은 조금 고민해보아야 할 문제이다.

음수로 주어진 어떤 수 $-n$과 $-m$이 있을 때, 이들은 2의 보수 표기법에서 $x = 2^{64} - n$과 $y = 2^{64} - m$으로 표시된다. 따라서 이들을 곱한 결과는 $xy = (2^{64} - n)(2^{64} - m) = 2^{128} - 2^{64}n - 2^{64}m + nm$으로 나타내어지며, 우리가 기대하던 답인 $(n \cdot m)$과는 거리가 멀다. 이는 결과 중 64비트만 사용할 때에는 문제가 되지 않는다.

하지만 결과를 128비트로 확장하여 상위 64비트도 사용하고 싶을 때에는 문제가 된다. 이를 극복하기 위해 X86_64 명령어 세트는 부호 있는 64bit * 64bit 곱셈 명령인 imulq S를 지원하지만 이 문제에서는 사용되지 않았다. 따라서 해당 보정 알고리즘을 직접 구성해야 한다.

위 식을 n과 m에 대한 식으로 다시 쓰면 다음과 같다:
$nm = xy - 2^{128} + 2^{64}n + 2^{64}m = xy - 2^{128} - 2^{64}x - 2^{64}y$

x와 y가 각각 음수/양수, 양수/음수인 경우는 다음과 같다.
$-nm = xy - 2^{64}m = xy - 2^{64}y$
$-nm = xy - 2^{64}n = xy - 2^{64}x$

즉, 원하는 곱셈 결과를 얻기 위해서는 부호를 고려하지 않고 계산된 곱셈의 결과에서 x가 음수이면 $2^{64}y$를, y가 음수이면 $2^{64}x$를 빼주면 된다. 이때 -2^{128}은 범위 밖이므로 고려하지 않는다. 곱셈의 결과를 하위 64비트와 상위 64비트로 나누어 저장한다면 이는 더 쉬워진다. 상위 64비트에서 x가 음수이면 y를, y가 음수이면 x를 빼주면 된다. 정리하면 다음과 같다.

$(hi, lo) = unsigned(x * y)$
$hi -= ((x < 0)?y : 0) + ((y < 0)?x : 0)$

주어진 어셈블리에는 사실 특별한 점이 없다. 음수 곱셈에서의 보정만 고려하면 된다. 어셈블리에 등장하는 cqto는 rdx와 rax를 각각 상위/하위 64비트씩 하여 128비트로 확장하는 instruction이다. 이때 rdx는 rax의 부호 비트로 채워진다. 어셈블리 각 줄의 기능은 다음과 같다.

```
movq        %rdx, %rax          ; copy y to rax.
cqto                            ; fill rdx with MSB of rax.
movq        %rsi, %rcx          ; copy given x to rcx.
sarq        $63, %rcx           ; sign extention of rcx.
imulq       %rax, %rcx          ; sign_ext(x) * y
imulq       %rsi, %rdx          ; sign_ext(y) * x
addq        %rcx, %rdx          ; combine two biases.
movq        %rax, (%rdi)        ; copy low 64bits to *%rdi (littel endian).
movq        %rdx, 8(%rdi)       ; copy high 64bits to *(%rdi + 8).
```

이와 같은 동작을 하는 C 소스코드는 다음과 같다.

```c
void store_prod(int128_t *dest, int64_t x, int64_t y) {
    int128_t result;

    int64_t  temp;
    int64_t  correction;

    temp = y;
    y >>= 63;                           // cqto, sign extension of y.

    correction = x;
    correction >>= 63;                  // sign extension of x.

    correction *= temp;                 // -y if x is under zero, otherwise zero.
    y *= x;                             // -x if y is under zero, otherwise zero.

    correction += y;                    // sign_ext(x)*y + sign_ext(y)*x.

    result = result * x;
    result = ((result & 0xFFFFFFFF00000000) + (correction << 64)) +
             (result & 0x00000000FFFFFFFF);

    *dest = result;
}
```

4. 참고

- https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/
- https://stackoverflow.com/questions/2535989/what-are-the-calling-conventions-for-unix-linux-system-calls-on-i386-and-x86-6
- https://stackoverflow.com/questions/33789230/how-does-this-128-bit-integer-multiplication-work-in-assembly-x86-64