어셈블리 실습 3

Tags

NASM 기반 실습: 스택을 통한 인자 전달 및 계산 문제

이번 실습은 **GCC의 x86-64**기준으로 NASM을 사용하여 진행됨

- 모든 인자 레지스터와 **스택을 혼용해서** 전달되며, NASM 문법을 사용하여 구현됨

x86-64 기본 개념 복습

- **리눅스/유닉스 (System V ABI)**:
  - 첫 6개의 인자는 **레지스터**를 통해 전달됨
  - 이후 인자는 **스택**을 통해 전달됨
- 사용되는 레지스터 - **System V ABI 인자 전달 규칙**:
  - 첫 번째 인자 → `RDI`
  - 두 번째 인자 → `RSI`
  - 세 번째 인자 → `RDX`
  - 네 번째 인자 → `RCX`
  - 다섯 번째 인자 → `R8`
  - 여섯 번째 인자 → `R9`

실습 문제 1: 스택 기반 합 계산 함수 (`SumValsI32_b.asm`)

8개의 32비트 정수(`int32_t`)를 인자로 받아서 이 값들을 모두 더하는 NASM 기반의 어셈블리 함수 `SumValsI32_b` 를 작성하세요. 모든 인자는 **스택을 통해** 전달됩니다.

1. 스택 프레임 및 인자 구조 (GCC 기준)

GCC의 x86-64 호출 규약에서는:

- 첫 6개의 인자는 일반적으로 레지스터(`RDI` , `RSI` , `RDX` , `RCX` , `R8` , `R9`)에 전달되지만, 이번 실습에서는 **모든 인자를 스택을 통해 전달**할 것입니다.
- 인자들은 **RSP**의 상대 위치에서 접근할 수 있습니다.

2. 구현해야 되는 코드 - 아래 참조

```
int Sum(int a, int b, int c, int d, int e, int f, int g, int h) {
    return a + b + c + d + e + f + g + h;
}
```

3. C 테스트 코드 (`test_sum_vals.c`)

```
#include <stdio.h>
#include <stdint.h>// 어셈블리 함수 선언
extern int32_t SumValsI32_b(int32_t a, int32_t b, int32_t c, int32_t
d,
                            int32_t e, int32_t f, int32_t g, int32_t
h);

int main() {
    int32_t a = 1, b = 2, c = 3, d = 4;
    int32_t e = 5, f = 6, g = 7, h = 8;

    int32_t result = SumValsI32_b(a, b, c, d, e, f, g, h);
    printf("Sum of values: %d\n", result);

    return 0;
}
```

4. 컴파일 및 실행 방법

GCC를 사용해서 어셈블리 코드로 변환하고, 인자를 전달하는 방식을 확인하자

```
gcc -S test_sum_vals.c
vim ./test_sum_vals.s
```

5. 어셈블리어 코드 main 부분을 붙여보자

- Question1?
  - 어셈블리 코드 main 부분을 붙여보자.
  - call SumValsI32_b@PLT 이전의 어셈블리어 코드 부분을 설명하시오. 단, .cfi_ 및 endbr64 명령어 제외

6. NASM 코드 작성 (`SumValsI32_b.asm`)

- 조건: rbp 레지스터 활용

```
section .text
global SumValsI32_b
```

```
SumValsI32_b:
    ; 스택 프레임 설정
    push rbp                ; 이전 RBP 값을 스택에 저장
    mov rbp, rsp            ; 현재 RSP 값을 RBP에 복사

    ; 인자 값들을 스택에서 가져와서 합 계산


    ; 리턴 값 설정 및 스택 프레임 해제

    ret
```

7. NASM을 사용하여 컴파일하고 실행합니다.

```
nasm -f elf64 SumValsI32_b.asm -o SumValsI32_b.o
gcc -g test_sum_vals.c SumValsI32_b.o -o test_sum_vals
./test_sum_vals
```

- 여기서 -g 옵션 유의: 디버깅 정보 포함

8. 출력

```
Sum of values: 36
```

9. GDB를 이용한 어셈블리 디버깅 실습 도구

이번 실습에서는 `GDB`를 사용하여 **NASM 어셈블리 함수 호출(`call`) 시의 동작을 확인**하고, 스택 프레임과 레지스터 값을 분석하는 방법을 배워보자

GDB 명령어 설명 테이블

명령어	설명	예제
`list`	현재 소스 코드의 내용을 표시합니다.	`list`
`break`	브레이크포인트를 설정합니다. 특정 함수나 라인에서 프로그램 실행을 멈춥니다.	`break main`
`run`	프로그램을 시작합니다. 브레이크포인트에서 멈추거나 종료 시까지 실행됩니다.	`run`
`disassemble`	어셈블리 코드를 표시합니다. 현재 함수나 특정 주소 범위의 명령어를 보여줍니다.	`disassemble main`
`nexti` (또는 `ni`)	다음 명령어(어셈블리 코드)로 넘어갑니다. **call** 명령어는 한 번에 실행됩니다.	`nexti`

`stepi` (또는 `si`)	다음 명령어로 들어갑니다. **call** 명령어인 경우, 호출된 함수 내부로 진입합니다.	`stepi`
`info registers`	현재 레지스터의 값을 표시합니다.	`info registers`
`info frame`	현재 스택 프레임의 정보 (예: `RBP`, `RSP` 위치, 인자 값) 를 표시합니다.	`info frame`
`x`	메모리의 내용을 표시합니다. 다양한 형식으로 메모리를 확인할 수 있습니다.	`x/10wx $rbp`
`bt` (또는 `backtrace`)	현재 콜 스택의 호출 흐름을 보여줍니다. 함수 호출의 연쇄를 확인할 수 있습니다.	`bt`
`print` (또는 `p`)	변수나 메모리 주소의 값을 출력합니다.	`print $rax`
`continue` (또는 `c`)	브레이크포인트까지 프로그램 실행을 계속합니다.	`continue`
`info locals`	현재 함수의 로컬 변수를 표시합니다.	`info locals`
`info args`	현재 함수의 인자 값을 표시합니다.	`info args`
`frame`	스택 프레임을 변경합니다. 특정 프레임으로 이동하여 그 상태를 확인할 수 있습니다.	`frame 1`
`finish`	현재 함수의 실행을 완료하고, 호출한 위치로 돌아갑니다.	`finish`
`set`	레지스터나 변수의 값을 변경합니다.	`set $eax = 10`
`info functions`	디버깅 중인 프로그램의 모든 함수 목록을 표시합니다.	`info functions`
`quit` (또는 `q`)	GDB를 종료합니다.	`quit`

디스플레이 예제: `x/10wx $rbp`

- `x`: 메모리를 검사합니다.
- `/10`: **10개의 4바이트 메모리 블록**을 표시함
- `w`: 각 메모리 블록의 크기는 **4바이트 (32비트)임**.
- `x`: 출력 형식은 **16진수 (hexadecimal)임**
- `$rbp`: 기준 주소는 `rbp` **레지스터**의 값입니다 (즉, 현재 스택 프레임의 기준 주소).

9.1. GDB 시작

```
gdb ./test_sum_vals
```

9.2. 주요 GDB 명령어

1. **소스 코드 확인:**

   ```
   list
   ```

2. **브레이크포인트 설정 (`main` 함수의 `call SumValsI32_b` 직전):**

```
break main
```

3. **프로그램 실행:**

```
run
```

4. **어셈블리 코드 보기 (`main` 함수):**

```
disassemble main
```

-

5. **다음 명령어 실행**

```
nexti
```

- **셈블리 명령어 단위로 디버깅**할 때 자주 사용하는 두 가지 명령어가 있습니다: `stepi (si)`와 `nexti (ni)`입니다.
- **`stepi (si)`와 `nexti (ni)` 비교**

명령어	동작	함수 호출 시 행동	사용 상황
`stepi (si)`	어셈블리 명령어 한 줄 실행	함수 내부로 진입	함수 내부를 한 줄씩 분석할 때
`nexti (ni)`	어셈블리 명령어 한 줄 실행	함수 호출을 건너뜀	함수 내부 분석이 필요 없을 때

- 사용 예시: step (si)

```
(gdb) stepi
```

또는

```
(gdb) si
```

6. **레지스터 상태 확인 (인자 값 확인):**

- **`print` 명령어 사용 -** `print` 명령어(`p`로도 사용 가능)를 사용하면 특정 레지스터의 값을 출력할 수 있습니다.

예시:

```
(gdb) print $rax
```

출력 예시:

```
$1 = 42
```

- 여기서 `$1`은 GDB의 내부 변수 번호이며, `RAX` 레지스터의 값이 `42`임을 나타냅니다.

- **`info registers` 명령어 사용**

`info registers` 명령어를 사용하면 **모든 레지스터**의 값을 출력하거나, 특정 레지스터만 출력할 수도 있습니다.

예시 (모든 레지스터 출력):

```
(gdb) info registers
```

출력 예시:

```
rax            0x2a    42
rbx            0x0     0
rcx            0x1     1
rdx            0x3     3
rsi            0x5     5
rdi            0x7     7
...
```

- `RAX` 값이 `0x2a`(16진수)이며, 이는 10진수로 `42` 가 됨

예시 (특정 레지스터만 출력):

```
(gdb) info registers rax
```

7. **스택 프레임 확인:**

```
info frame
```

제출물

- 64비트로 컴파일된 메인 함수의 어셈블리어 코드를 붙이고, SumValsI32_b@PLT 이전의 코드 부분에 대한 설명
- **NASM 코드 작성 (`SumValsI32_b.asm`)**
- GDB를 사용하여 `SumValsI32_b` 함수 호출 시 반환 값(`eax`)이 점차 커지는 과정을 캡처함
  - 반환 값이 어떻게 증가하는지 보여주는 스크린샷을 포함함
- **32비트로 `main` 함수 컴파일하는 명령어 정리**
  - 64비트 환경에서 C 프로그램(`main` 함수 포함)을 컴파일하려면 **GCC의 `-m32` 옵션**을 사용할 수 있음
  - 예시

```
gcc -m32 -S test_sum_vals.c
```

- 32비트로 컴파일했을 때, main 함수 부분 어셈블리어 코드를 붙이고, 다시 분석해보자
  - 64비트와 비교했을 때, 인자 전달하는 방식이 어떻게 다른지 결정적인 차이를 한문장으로 적으시오.
- 바뀐 메인 어셈블리어 코드를 바탕으로 32비트용 어셈블리코드를 작성하시오. **(`SumValsI32_c.asm`)**
- 다음 명령어를 사용하여 컴파일하고 실행해보고, 결과를 캡쳐하시오.

```
nasm -f elf32 SumValsI32_c.asm -o SumValsI32_c.o
gcc -m32 -g test_sum_vals.c SumValsI32_c.o -o test_sum_vals
./test_sum_vals
```