연구일지 part2 정리

어셈블리 문법에 따른 관점

이전에는 컴파일러를 중심으로 분석해보았으나 이 문서는 어셈블에 촛점을 맞추어 보기로 하자.
애시당초 목표는 어셈블 분석단계에서 어째서 서로다른 OS 에서 컴파일이 되지 않는가를 알아보려고 했었다. 그러나 어셈블 코드를 살펴본 결과 서로다른 Os에서 나온 같은 gcc계열의 컴파일러의 결과물은 크게 차이나지는 않았다.
오히려 같은 Windows기반이라고 해도 컴파일러에 따라서 문법의 차이가 상당히 많았다.

일레로 Windows 에서 뽑아낸 gcc와 Visual Studio 의 결과물의 차이를 보도록 하자

```
    .file   "hello_win.c"
    .def    ___main;        .scl    2;      .type   32;     .endef
    .text
LC0:
    .ascii "Hello world!\0"
.globl _main
    .def    _main;  .scl    2;      .type   32;     .endef
_main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    movl    $0, %eax
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
    call    __alloca
    call    ___main
    subl    $12, %esp
    pushl   $LC0
    call    _printf
    addl    $16, %esp
    leave
    ret
    .def    _printf;        .scl    2;      .type   32;     .endef
```

이것이 윈도우에서 Mingw32(gcc 3.4)에서 뽑아낸 결과이고

```
1:      #include <stdio.h>
2:      int main(){
00401010    push        ebp
00401011    mov         ebp,esp
00401013    sub         esp,40h
00401016    push        ebx
00401017    push        esi
00401018    push        edi
00401019    lea         edi,[ebp-40h]
0040101C    mov         ecx,10h
00401021    mov         eax,0CCCCCCCCh
00401026    rep stos    dword ptr [edi]
3:          printf("Hello world");
00401028    push        offset string "Hello world" (0042001c)
0040102D    call        printf (00401060)
00401032    add         esp,4
4:      }
00401035    pop         edi
00401036    pop         esi
00401037    pop         ebx
00401038    add         esp,40h
0040103B    cmp         ebp,esp
0040103D    call        __chkesp (004010e0)
00401042    mov         esp,ebp
00401044    pop         ebp
00401045    ret
```

이것은 Visual Studio 6 에서 뽑아낸 결과이다.
서로 같은 소스를 바탕으로 작업했지만 문법부터 상당한 차이를 보인다.

OS 에 따라서 어셈블리의 차이가 나는게 아닌가?
하지만 두개의 어셈블은 컴파일러가 다를뿐 결과는 같은 결과물을 가져온다.
단지 문법상의 차이일듯한데 이 문법상의 차이는 어디서 오는지 조사했다

참고문서는 RedHat 에서 제공하는 AT&T 어셈블 문법과 인텔 문법의 차이점에 관한 문서를 참조했다.

AT&T Syntax VS Intel Syntax

AT&T 문법과 인텔 문법의 비교
원문 : http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/gnu-assembler/i386-

syntax.html

개략적 해석문
어셈블리를 사용할때 인텔어셈블러의 문법을 설명할때는 intel_syntax라고 명시하고
gcc의 AT&T의 문법을 설명할때에는 att_syntax 라고 명시한다.

일반적으로 책에 서술되어있는형태는 인텔의 8086 문법이다. AT&T의 문법과 인텔의 문법은 다르기 때문에 두 문법간의 차이를 기술한다.

- AT&T immediate operands are preceded by `$`; Intel immediate operands are undelimited (Intel `push 4` is AT&T `pushl $4`). AT&T register operands are preceded by `%`; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by `*`; they are undelimited in Intel syntax.
- AT&T 상수는 $로 표시된다. 인텔의 상수는 아무것도 붙이지 않는다(대략적인 예로써 인텔의 경우 push 4는 AT&T에서는 pushl $4로 표기한다).
  AT&T의 레지스터 오퍼랜드는 %로 펴기한다. 인텔의 경우 표시하지 않는다. AT&T의 절대주소(?)는 (사용하는 머신에따라 달라질수도있음) jump/call 오퍼렌드에는 * 의 접두어가 붙는다. 인텔의 경우 표시하지 않는다.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel `add eax, 4` is `addl $4, %eax`. The `source, dest` convention is maintained for compatibility with previous Unix assemblers. Note that instructions with more than one source operand, such as the `enter` instruction, do *not* have reversed order. Section 21.11 *AT&T Syntax bugs*.
- AT&T와 인텔의 문법은 소스와 목적이 되는 오퍼렌드의 순서가 대립된다.
  인텔은 add eax, 4 로 표기하며 AT&T는 addl $4, %eax로 표기한다.
  소스와 목적의 관례에 따라 이전의 Unix 어셈블러와 호환되게 유지한다.
  AT&T문법중 버그로 인하여 정방향이 아닌 역순으로 해석되는 경우가 있다.
- In AT&T syntax the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of `b`, `w`, `l` and `q` specify byte (8-bit), word (16-bit), long (32-bit) and quadruple word (64-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (*not* the instruction mnemonics) with `byte ptr`, `word ptr`, `dword ptr` and `qword ptr`. Thus, Intel `mov al, byte ptr foo` is `movb foo, %al` in AT&T syntax.
- AT&T문법은 오퍼렌드의 메모리 크기는 명령어문 마지막에 쓰인 글자에 의해 결정된다.
  명령어의 접미사로 b,w,l,q등이 입력된것은 8비트를 의미하고 word는 16비트, long은 32비트 quadruple은 64비트의 메모리를 참조한다.
  인텔 문법은 메모리 오퍼랜드의 접두어로써 결정된다.(명령어가 아니다)

`byte ptr`, `word ptr`, `dword ptr` and `qword ptr` 로 표현된다.
인텔에서 mov al, byte ptr foo 로 표현되었다면 AT&T에서는 movb foo, %al 로 표현된다.

- Immediate form long jumps and calls are `lcall/ljmp $section, $offset` in AT&T syntax; the Intel syntax is `call/jmp far section:offset`. Also, the far return instruction is `lret $stack-adjust` in AT&T syntax; Intel syntax is `ret far stack-adjust`.

- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.
- AT&T의 어셈블러는 섹션을 분할하지 못한다. 유닉스 시스템은 단일 섹션으로 프로그래밍된다.

간단하게 정리하자면 유닉스의 어셈블은 AT&T라는 문법을 사용하고 있어서 이러한 차이를 가져오는것으로 보인다. 한가지 주목할만한 차이점이라면 명령어 니모닉 끝에 각 처리하는 크기를 정하는 단어를 붙인다는것과, 해석의 순서가 인텔의 것과 반대인 정방향이라는것이다.(이문제가 풀리지 않아서 해석당시 상당히 고생했었다)

하지만 아직 남은 문제가 동일한 gcc에서 뽑아낸 결과라 할지라도 윈도우의 어셈블 결과와 리눅스의 어셈블 결과는 차이를 보인다. 각각의 어셈블자료를 추적해보았다.

참조문서는 http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax 를 참조했다.

먼저 윈도우에서 뽑아낸 어셈블 결과부터 살펴보자

```
#       .file   "hello_win.c"
#       .def    ___main;        .scl    2;      .type   32;     .endef
#         .file 이나 .def부분은 디버깅을 위해 필요한 부분이다. 현재는 필요하지 않다.
#         .file .def .ascii 와 같은 것들은 어셈블리 지시어들이다.
#         어셈블을 어떻게 할것인가 알려주는 역할을 한다.

#       .text
#         .text 는 코드의 섹션을 시작을 정의한다. 이것을 통해 섹션의 네임을 지정할 수 있다.

LC0:
        .ascii "Hello world!\0"
#         이부분은 이 코드에서 사용할 ascii 텍스트 코드가 어디에 있는지 말해준다.
```

```
#       LC0는 이름표인데 나중에 출력당시 이부분을 이용해 문자열이 어디에 위치하는지 찾아낸다

.globl _main
#       다른부분에서 참조할수 있도록 어셈블러에게 _main 이 전역함수라는것을 설명해주는 부분이다.
#       이 경우에는 링커가 _main 부분을 볼수 있도록 코드의 시작에 _main 부분을 불러온다.
#       실제적으로 아래의 _main 부분의 아래부터 명령어가 기록된다

#     .def    _main;  .scl    2;      .type   32;     .endef
_main:
      pushl   %ebp
#       epb의 값을 스택의 맨 윗쪽으로 밀어넣는다

      movl    %esp, %ebp
#       esp의 값을 ebp로 넣는다          * 퍼센트(%)는 레지스터를 의미한다

      subl    $8, %esp
#       달러($) 표시는 상수를 의미한다. 이경우 8을 esp값과  뺐다
#       위의 3작업은 저장할수 있는 메모리를 확보하기위한 전형적인 작업이다.



#     andl    $-16, %esp
#       이 코드는 esp와 -16을 AND논리연산한다.
#       이 기능은 주소를 정렬하는 기능인데 이 프로그램에서는 실제로 쓸모는 없다.

#     movl    $0, %eax
#       eax에 0을 집어넣는다

#     movl    %eax, -4(%ebp)
#     movl    -4(%ebp), %eax
#       위의 세 명령어는 eax의 값을 ebp-4에 덮어쓰고, 다시 ebp-4의 값을 eax에 복사하고 있다.
#       왜 컴퓨터가 이러한 부분을 만들었는지 알수 없지만 최적화 되지 않은 필요없는 부분이다

      call    __alloca
      call    ___main
#       위의 두 코드는 C 라이브러리를 불러오기위한 부분이다.
#       C라이브러리 안에 있는 함수를 calling 했다면 이 부분이 필요할 것이다.
#       이 부분이 플랫폼의 종류와  GNU 툴의 버젼에 가장 민감한 부분이다.

#     subl    $12, %esp
#       esp에 있는 값을 12로 빼주는 역할인데 이 프로그램에선 전혀 필요가 없다

      pushl   $LC0
      call    _printf
#       위의 두줄이 문자를 출력하는 부분이다.
#       ascii 문자열을 스택의 맨 꼭대기로 밀어준다. 그 다음 C라이브러리에서 _printf서브루틴을 끌어온다.
```

```
#       addl    $16, %esp
#         esp에 16을 더하는 명령. 이 프로그램에서 필요없음

        leave
#         서브루틴의 끝을 의미한다.사용했던 ebp와 esp 를 반환하고 저장된 epb의 값을 ebp로 밀어준다(?)
#         This line, typically found at the end of subroutines, frees the space saved on the stack by copying
EBP into ESP, then popping the saved value of EBP back to EBP.

        ret
#         프로시져를 불러오기위해 스택에 저장된 IP를 따라간다
#       .def    _printf;        .scl    2;      .type   32;     .endef
```

길게도 썼지만 막상 필요없는 주석부분만 빼면 다음과 같다.

```
LC0:
        .ascii "Hello world!\0"
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        call    __alloca
        call    ___main
        pushl   $LC0
        call    _printf
        leave
        ret
```

기계적 어셈블러가 실제로 비효율적이라는것을 알 수 있다.
이것이 윈도우에서 사용하는 어셈블운용방법이다.

다음은 리눅스상에서 gcc를 통해 어셈블한 코드이다.

```
        .file   "hello_linux.c"
        .section        .rodata
.LC0:
        .string "Hello world!"
```

```
        .text
.globl  main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)"
        .section        .note.GNU-stack,"",@progbits
```

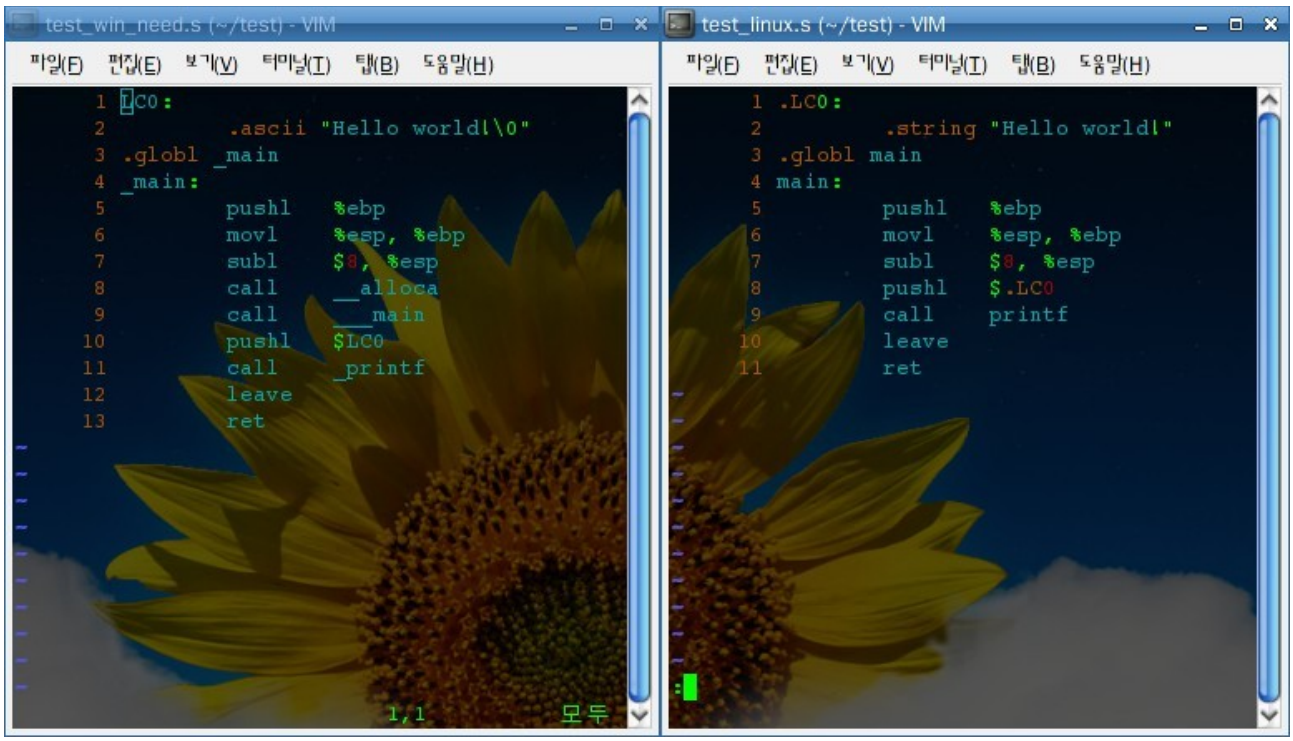
혹시나 해서 윈도우상에서 gcc가 어셈블한 결과중 쓰레기 코드와 같은것이 있어 찾아 지워보았다.
위에서의 경험을 토대로 다른 부분과 연결되지 않을것같은 부분을 전부 지워보았다.

```
.LC0:
        .string "Hello world!"
.globl  main
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        pushl   $.LC0

        call    printf
        leave
        ret
```

지우고 나서 다음과 같은 코드가 완성되었는데, 컴파일 역시 성공적이었다.

그런데, 놀랍게도 윈도우에서 뽑아낸 어셈블 파일을 정리한것과 리눅스에서 뽑아낸 어셈블 파일을 정리한것을 비교해봤을때 거의 차이가 없었다.



[좌측은 윈도우 에서 뽑아낸것이고 우측은 리눅스에서 뽑아낸 결과물이다]

Windows	*Linux*
.ascii 로 문자열을 표현한다. 널문자 까지 포함됨	.string 으로 문자열을 표현한다. 널문자 포함되지 않음
__alloca 와 ___main을 call 한다	그런부분 없음
LC0 라고 표시하고 _printf 를 호출한다	.LC0 라고 표시하고 printf 를 호출한다

위의 주석문에서 말했듯이 __alloca와 ___main은 플랫폼에 종류와 C 라이브러리의 차이라고 말했다.
또한 라이브러리를 호출하는데 동일한 printf함수임에 불구하고 한쪽은 _printf라고 쓰였다.
이것은 라이브러리 안에 서로 다른 형태로 저장되어있음을 시사하는것이라고 볼 수 있다.

완벽하게 동일한 형태의 어셈블 파일이라고 가정하고 이 두가지가 호환되지 않는 이유를 C라이브러리에서 찾아볼수도 있을것 같다.