펭귄 스터디 내용03

작성자 : 류성수
작성일자 : 24.06.30

url : https://www.youtube.com/watch?v=ajoQujYfVI0&list=PLz-ENLG_8TMdMJIwyqDIpcEOysvNoonf&index=4

사용 리눅스 OS이름 : Ubuntu

● GDB 디버거

1. 이걸 도대체 왜 쓰는가?

만약, ARM 아키텍쳐처럼, MCU 자체가 너무 협소해서, OS를 깔고, 그 위에 컴파일러를 깔아 실행파일을 만들 수 없는 경우를 가정한다.
이러한 경우에서는 특별한 컴파일러의 개념이 나오는데, 바로 크로스 컴파일러 개념이다.
크로스 컴파일러는 데스크탑에서 다른 시스템에서 동작할 것을 타겟으로 하여 실행파일을 생성하는 과정을 통틀어서 의미한다.
이전에, Atmega 1281에 , microchip studio를 이용하여 펌웨어 시스템을 만들어 봤던 경험이 있다. 해당 MCU는 1 clk당 받아올 수 있는 비트수가 8bit 였기에, 데스크탑의 64bit에서의 코드 exe 파일을 그대로 갖고 가면, 정상적으로 동작이 되지 않게 된다.

그러나, 크로스 컴파일러 자체는 해당 MCU를 만들었던 회사에서 제공이 되지만, 버그를 어떻게 잡아낼것인지가 문제이다.
위에서 언급하였듯이, 크로스 컴파일링은 환경이 전혀 다르기에, 현재 데스크탑에서 오류가 발생하지 않았더라도, 타겟에서는 문제가 발생할 수 있다.
그렇기에, 그 환경을 가상적으로나마 구현해서 디버거역할을 하는 프로그램이 gdb 디버거이다.
(VScode, Microchip Studio 등 여러곳에서 gdb의 이름을 심심치 않게 보았기 때문에 완전 낯설지는 않았다.)

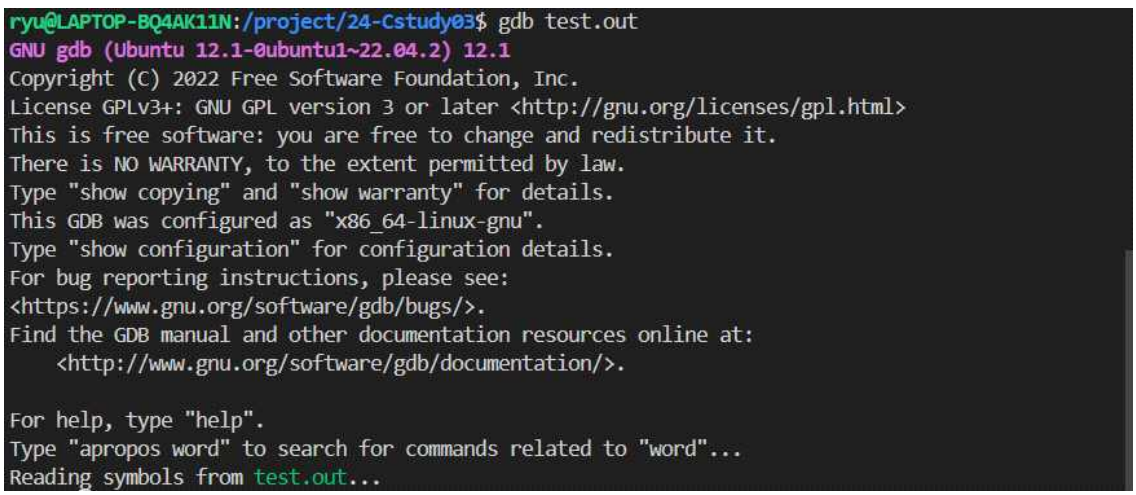
1. 1. 간단한 사용법

일단 gdb를 실행하기 전에 필요한 절차가 있다.
제일먼저, gcc 혹은 g++ 컴파일러를 이용해, 소스코드를 .out 형태의 실행파일로 바꿔주어야 하는데, 여기서 컴파일러에게 옵션을 부여해 주어야 한다.

ex) gcc -g 파일이름.c -o 출력이름.out

여기서 -g와 -o는 컴파일러에게 옵션을 부여하는 것이다. -g는 컴파일된 실행파일이 디버깅에 필요한 정보를 갖게하여 gdb가 확인 할 수 있게 한다.

그리고 다음의 사진처럼 터미널창에 작성해주면,



아까 -g로 전달한 정보(=심볼)들을 gdb가 읽는 것을 맨 아랫줄에서 확인이 가능하다.

여기서, gdb 프로세스에게 명령을 내려줄 수 있는 창이 등장한다. 이곳에 'run' 을 입력시,

```
(gdb) run
Starting program: /project/24-Cstudy03/test.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello Ubuntu!
[Inferior 1 (process 458) exited normally]
(gdb)
```

해당 파일이 실행되는 것을 확인 할 수 있다.

코드를 보고싶다면, 'l' 을 입력하면 된다. 그 뒤에 숫자를 입력시 원하는 코드행 수 까지 출력해준다.

```
(gdb) l 30
Line number 25 out of range; test.cpp has 8 lines.
(gdb) l 8
3       using namespace std;
4       int  main()
5       {
6               cout << "Hello Ubuntu!" << endl;
7               return 0;
8       }
```

이는 숫자를 입력하지 않고, 함수 자체를 출력시키면, 해당함수 (main 은 전체를 출력) 다음과 같이 나온다.

```
(gdb) l main
1       #include <iostream>
2
3       using namespace std;
4       int  main()
5       {
6               cout << "Hello Ubuntu!" << endl;
7               return 0;
8       }
```

1. 2. breaking point

디버거에서 가장 중요한 것은 중단점이라고 생각한다. 이 기능을 사용하여, 어디부분에서 오류가 나는지를 확인할 수 있고, 그에 적절한 조치를 취해 실행이 정상적으로 동작하게 만들 수 있기 때문이다.

중단점은 'b' 를 입력하고 뒤에 코드 행 숫자 또는 함수를 입력하여 중단점을 걸 수 있다.

```
1       #include <iostream>
2
3       using namespace std;
4       int  main()
5       {
6               cout << "Hello Ubuntu!" << endl;
7               return 0;
8       }
(gdb) b 4
Breakpoint 1 at 0x5555555551b1: file test.cpp, line 6.
```

1. 3. 디어셈블리 모드

gdb 명령창에 'disas'를 입력하고 뒤에 함수 혹은 코드 행 숫자를 입력하면 해당부분들의 어셈블리 코드를 볼 수 있다.
매우 발전한 데스크탑 시스템에서는 굳이 어셈블리 코드를 확인할 필요가 없지만, MCU와 같은 곳에서는 아키텍쳐가 다르기 때문에 어셈블리 코드를 확인하여 어느정도 MCU에서 실행파일 혹은 소스코드가 어떻게 작동되는지를 알아야 하기 때문에 본다고 이전에 교수님이 설명하셨던 것을 기억한다.

```
(gdb) disas main
Dump of assembler code for function main():
   0x000055555555551a9 <+0>:     endbr64
   0x000055555555551ad <+4>:     push   %rbp
   0x000055555555551ae <+5>:     mov    %rsp,%rbp
   0x000055555555551b1 <+8>:     lea    0xe4c(%rip),%rax        # 0x555555556004
   0x000055555555551b8 <+15>:    mov    %rax,%rsi
   0x000055555555551bb <+18>:    lea    0x2e7e(%rip),%rax        # 0x555555558040 <_ZSt4cout@GLIBCXX_3.4>
   0x000055555555551c2 <+25>:    mov    %rax,%rdi
   0x000055555555551c5 <+28>:    call   0x555555555090 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
   0x000055555555551ca <+33>:    mov    0x2dff(%rip),%rdx        # 0x555555557fd0
   0x000055555555551d1 <+40>:    mov    %rdx,%rsi
   0x000055555555551d4 <+43>:    mov    %rax,%rdi
   0x000055555555551d7 <+46>:    call   0x5555555550a0 <_ZNSolsEPFRSoS_E@plt>
   0x000055555555551dc <+51>:    mov    $0x0,%eax
   0x000055555555551e1 <+56>:    pop    %rbp
   0x000055555555551e2 <+57>:    ret
End of assembler dump.
```

● 여담

이전 강의 영상에서 다양한 자료형을 이용하여 변수들을 출력해보고 시스템에서 어떻게 작동하는지를 잠깐 배웠었다.
이번 강의 영상에서는 그 코드를 gdb의 디어셈블기능을 이용해 어셈블리 코드를 다 뜯어보는 시간을 가졌다.

```
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     sub    $0x40,%rsp
   0x0000000000001155 <+12>:    movl   $0x0,-0x34(%rbp)
   0x000000000000115c <+19>:    movl   $0x1,-0x30(%rbp)
   0x0000000000001163 <+26>:    movb   $0x2,-0x3a(%rbp)
   0x0000000000001167 <+30>:    movb   $0x3,-0x39(%rbp)
   0x000000000000116b <+34>:    movw   $0x5,-0x38(%rbp)
   0x0000000000001171 <+40>:    movw   $0x6,-0x36(%rbp)
   0x0000000000001177 <+46>:    movq   $0x7,-0x28(%rbp)
   0x000000000000117f <+54>:    movq   $0x8,-0x20(%rbp)
   0x0000000000001187 <+62>:    movss  0xf61(%rip),%xmm0        # 0x20f0
   0x000000000000118f <+70>:    movss  %xmm0,-0x2c(%rbp)
   0x0000000000001194 <+75>:    movsd  0xf5c(%rip),%xmm0        # 0x20f8
   0x000000000000119c <+83>:    movsd  %xmm0,-0x18(%rbp)
   0x00000000000011a1 <+88>:    movq   $0xd,-0x10(%rbp)
   0x00000000000011a9 <+96>:    movq   $0xe,-0x8(%rbp)
   0x00000000000011b1 <+104>:   mov    -0x34(%rbp),%eax
   0x00000000000011b4 <+107>:   mov    %eax,%esi
   0x00000000000011b6 <+109>:   lea    0xe4b(%rip),%rax        # 0x2008
   0x00000000000011bd <+116>:   mov    %rax,%rdi
   0x00000000000011c0 <+119>:   mov    $0x0,%eax
   0x00000000000011c5 <+124>:   call   0x1050 <printf@plt>
   0x00000000000011ca <+129>:   mov    -0x30(%rbp),%eax
   0x00000000000011cd <+132>:   mov    %eax,%esi
   0x00000000000011cf <+134>:   lea    0xe3e(%rip),%rax        # 0x2014
   0x00000000000011d6 <+141>:   mov    %rax,%rdi
   0x00000000000011d9 <+144>:   mov    $0x0,%eax
   0x00000000000011de <+149>:   call   0x1050 <printf@plt>
   0x00000000000011e3 <+154>:   movsbl -0x3a(%rbp),%eax
   0x00000000000011e7 <+158>:   mov    %eax,%esi
   0x00000000000011e9 <+160>:   lea    0xe3a(%rip),%rax        # 0x202a
   0x00000000000011f0 <+167>:   mov    %rax,%rdi
   0x00000000000011f3 <+170>:   mov    $0x0,%eax
   0x00000000000011f8 <+175>:   call   0x1050 <printf@plt>
   0x00000000000011fd <+180>:   movzbl -0x39(%rbp),%eax
   0x0000000000001201 <+184>:   mov    %eax,%esi
```

```
#include <stdio.h>

void main()
{
        int i=0;
        unsigned int ui=1;
        char c=2;
        unsigned char uc=3;
        short s=5;
        unsigned short us = 6;
        long l = 7;
        unsigned long ul = 8;
        float f = 9.;
        //unsigned float uf = 10.0;
        double d = 11.0;
        //unsigned double ud = 12.0;
        long long ll = 13;
        unsigned long long ull = 14;

        printf("int i = %d\n", i);
        printf("unsigned int ui = %u\n", ui);
        printf("char c = %d\n", c);  // %d 사용
        printf("unsigned char uc = %u\n", uc);  // %u 사용
        printf("short s = %d\n", s);
        printf("unsigned short us = %u\n", us);
        printf("long l = %ld\n", l);
        printf("unsigned long ul = %lu\n", ul);
        printf("float f = %f\n", f);
        printf("double d = %f\n", d);
        printf("long long ll = %lld\n", ll);
        printf("unsigned long long ull = %llu\n", ull);

        //unsigned float은 안된다. 자료형으로 선언 자체가 안됨
}
~
~
~
~
~
```

어셈블리 코드를 하나하나 다 따지지 않고, 대충 변수는 어셈블리에서 이러한 과정을 거친다는 것을 보여주시기 위해 일부러 설명하신 것 같다는 생각이 들었다.

이전 시스템 프로그래밍 과목에서 이러한 어셈블리어를 열어보는것들을 확인하였고, 배웠던 대로 어셈블리 코드가 작성된 것을 확인 할 수 있었다.

굉장히 특이한 부분을 발견할 수 있었는데, gdb로 실행파일을 어셈블리 코드로 열었을 때의 컴파일 단계를 유추해볼 수 있었다.
현재, printf를 호출할 때, 0x1050 <printf@plt> 이러한 형식으로 작성된 것을 보이는데, 이는 컴파일의 과정 중, 오직 전처리과정과

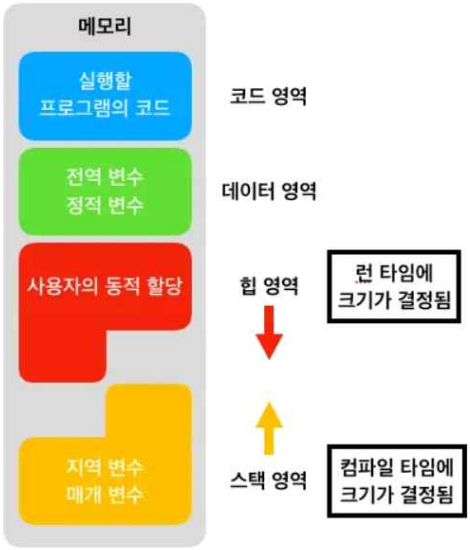
컴파일 과정만을 거친 형태라고 가정해 볼 수 있겠다.

그 뒤의 어셈블 과정은 기계어로 번역되고, 그 후의 링킹 과정은 해당 <printf@plt> 부분에 실제 printf 함수의 주소를 붙여넣어 jmp 기능으로 바뀌기 때문이다.

● 상수와 메모리 구조

이전까지 상수 자체는 런타임에서 구성되는 것으로 알고 있었다. 굳이 자세히 들어가면, 파이썬 코드에서는 성능의 향상을 위해 메모리 구조의 특정한 영역에 1부터 200 조금 넘는 수들을 저장하는 것으로 알고있었는데, C에서도 성능의 향상을 위해 특정한 메모리 영역에 상수 자체를 저장한다는 것을 알게되었다.

엽의 사진을 보면, a.out 파일을 실행하였을 때, 메모리에 어떻게 위치되는지를 대략적으로 나타낸 사진이다.
상수는 코드영역의 끄트머리에 저장된다는 것을 새로 배웠다.
또한, 상수는 컴파일 단계에서 작성&결정 되고 수정이 불가능하다.
그리고 unique한 특징을 가진다. 만약, 코드에서 상수를 call 하는 구조의 코드일 때, 모든 코드영역에서 상수에 접근이 가능하다.



1. 상수와 const 차이

C에서 const 변수를 미리 선언하고, 그를 이용해 배열 선언이 불가능하다.

이게 왜 그렇냐면, const로 선언된 변수여도 일단 존재 자체가 변수이기 때문에, 기본적으로 코드영역에 저장되지 않기 때문이다.
단지 const는 문법상에서만 변환이 되지 않게 억제할 뿐이다.

그러나 C++에서는 되긴 한다.

```
ryu@LAPTOP-BQ4AK11N:/project/24-Cstudy03$ sudo g++ ConstWithArr.cpp -o ConstWithArr.out
ryu@LAPTOP-BQ4AK11N:/project/24-Cstudy03$ ls
ConstWithArr.cpp  ConstWithArr.out  asmtest.out  test.cpp  test.out  var03.c
ryu@LAPTOP-BQ4AK11N:/project/24-Cstudy03$ ./ConstWithArr.out
ryu@LAPTOP-BQ4AK11N:/project/24-Cstudy03$ cat ConstWithArr.cpp
using namespace std;

int main()
{
        const int num = 20;
        int Arr[num] = {0};

        return 0;
}
```

이처럼, c++ 코드는 g++에서 컴파일을 할 수 있는데, 이는 다음으로 해석할 수 있다.
gcc와 g++의 컴파일 순서와 규칙이 다르기에 const로 선언된 변수가 정말 말그대로 상수화가 되기 때문이다.
지금까지, gcc와 g++의 차이점을 제대로 알 수가 없었는데, 이러한 디테일을 알 수 있다는 건 놀라운 사실이다.