

GNU Debugger, GNU Make – 5th Lab

■ GNU Debugger

1. gdb 실행

- Sample code

```
/* test1.c */
#include <stdio.h>

struct
{
    int *i_p;
} s;

void foo()
{
    *(s.i_p) = 2;
    printf("s.i = %d\n", *(s.i_p));
}

main(int argc, char *argv[])
{
    foo();
    //int *i_p;
    /*i_p = 1;
}
```

- Compile with "-g" Option

```
$ gcc -o test test.c -g
$ ./test
세그멘테이션 오류
```

- Run with gdb

➤ gdb 실행시 파일을 지정하는 경우

**4190.102A Computer Programming
(2007 Winter)**

```
$ gdb
(gdb) file test1
(gdb) run 5
Starting program: /home/comp-ta/gdb/test1 5
Program received signal SIGSEGV, Segmentation fault.
0x0804835f in foo () at test1.c:11
11          *(s.i_p) = 2;
```

- gdb안에서 파일을 지정하여 실행하는 경우

```
$ gdb
(gdb) file test1
(gdb) run 5
...
```

- gdb 실행시 파일과 core dump 파일을 지정하는 경우

- core dump : 프로그램이 비정상적으로 종료할 때 만들어지는 종료 시점의 프로그램 상태를 저장한 파일. (process의 memory image)
- 일반 사용자에게는 불필요한 파일이기 때문에 default는 생성되지 않는 것.
- core 파일 생성을 위해서는 사용자 자원 한계를 설정하는 ulimit 명령을 사용.

```
$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
...
$ ulimit -c unlimited
$ ulimit -a
core file size (blocks, -c) unlimited
...
$ ./test1
세그멘테이션 오류 (core dumped)
$ gdb test1 core
...
Core was generated by `./test1'.
Program terminated with signal 11, Segmentation fault.
#0  0x0804835f in foo () at test1.c:11
11          *(s.i_p) = 2;
```

- gdb 안에서 파일과 core dump 파일을 지정하는 경우

```
(after set the core file size to unlimited)
$ gdb
(gdb) file test1
(gdb) core-file core
...
```

2. Debugging Scenario #1 (런타임 에러이나 Segmentation Fault처럼 명시적인 에러)

- 일단 실행시켜 문제되는 부분을 파악

```
(gdb) run 5 (← core dump파일을 이용하면 문제 부분으로 자동으로 이동)
Starting program: /home/comp-ta/gdb/test1 5

Program received signal SIGSEGV, Segmentation fault.
0x0804835f in foo () at test1.c:11
11          *(s.i_p) = 2;
(gdb) list foo
5      {
6          int *i_p;
7      } s;
8
9      void foo()
10     {
11         *(s.i_p) = 2;
12         printf("s.i = %d\n", *(s.i_p));
13     }
14
(gdb) where
#0 0x0804835f in foo () at test1.c:11
#1 0x08048394 in main () at test1.c:18
(gdb) ptype s
type = struct {
    int *i_p;
}
(gdb) ptype s.i_p
type = int *
```

- 함수 foo의 11번 라인에서 문제 발생한다.
- 현재의 Stack Frame은 #0 이며 main함수 아래에 위치하는 것으로 보아 함수 main에서 함수 foo를 호출함을 알 수 있다.
- 문제는 어떤 포인터 변수 s.i_p가 가리키고 있는 곳에 "2"를 넣는 부분에서 발생, ptype이나 whatis 명령을 통해 s.i_p가 int의 포인터 변수임을 알 수 있다.

- 문제되는 곳의 원인 파악

```
(gdb) print s.i_p
$1 = (int *) 0x0
(gdb) print *(s.i_p)
Cannot access memory at address 0x0
```

- 값이 0x0(NULL)이며 따라서 이 값을 Access할 수 없다! (문제 확인!)

4190.102A Computer Programming (2007 Winter)

(참고)

```
(gdb) where
#0 0x0804835f in foo () at test1.c:11
#1 0x0804839b in main () at test1.c:18
(gdb) up
#1 0x0804839b in main () at test1.c:18
18         foo();
(gdb) list
13     }
14
15     main(int argc, char *argv[])
16     {
17         int i=1;
18         foo();
19     }
20
(gdb) print i
$1 = 1
```

- 현재 위치하고 있는 함수 foo를 부르는 함수 main에서 i_p의 값을 확인하고 싶은 경우 up 명령을 통해 현재 Stack Frame의 상위 Frame으로 이동한 후 print 명령을 수행할 수 있다.

- vi등의 편집기를 이용하여 코드 수정

```
#include <stdlib.h> // added
...
void foo()
{
    s.i_p = (int*)malloc(sizeof(int)); // added
    *(s.i_p) = 2;
    printf("s.i = %d\n", *(s.i_p));
    free(s.i_p); // added
}
```

3. Debugging Scenario #2 (런타임 에러이나 코드의 Logic이 잘못된 경우; Logical Error)

- Sample code

**4190.102A Computer Programming
(2007 Winter)**

```
/* test2.c */
#include <stdio.h>

main()
{
    double i;
    printf("An example of the infinite loop!\n");

    for(i = 0; i != 1; i += 0.1)
    {
        printf("current i=%f\n", i);
        sleep(0);
    }
}
```

➤ 실행하면 무한루프가 발생한다!

● 문제확인

- 발생하는 문제는 첫번째 예와는 달리 논리적인 문제라서 실행 시 어떠한 종류의 에러인지 알지 못한다.
- 따라서 개발자는 프로그램에서 문제가 발생할 것 같은 부분을 짐작한 후 그 부분부터 Tracing해야 할 필요가 있다.
- 짐작한 위치에서 프로그램이 멈추도록 break명령을 이용하여 breakpoint를 설정한다.

```
$ gdb test2
...
(gdb) list main
1      /* test2.c */
2      #include <stdio.h>
3
4      main()
5      {
6          double i;
7          printf("An example of the infinite loop!\n");
8
9          for(i = 0; i != 1; i += 0.1)
10         {
(gdb) list
11             printf("current i=%f\n", i);
12             sleep(0);
13         }
14     }
(gdb) break 11
Breakpoint 1 at 0x80483e8: file test2.c, line 11.
(gdb) info breakpoints
Num Type          Disp Enb Address      What
1  breakpoint      keep y   0x080483e8 in main at test2.c:11
```

➤ 프로그램을 실행시켜 지정한 breakpoint로 이동하기 위해서 run명령을 사용한다.

**4190.102A Computer Programming
(2007 Winter)**

```
(gdb) run
Starting program: /home/comp-ta/gdb/test2
Failed to read a valid object file image from memory.
An example of the infinite loop!

Breakpoint 1, main () at test2.c:11
11          printf("current i=%f\n", i);
```

➤ 다음 Statement로 이동하기 위해서 next나 step명령을 사용한다.

```
(gdb) next
current i=0.000000
12          sleep(0);
(gdb) next
9          for(i = 0; i != 1; i += 0.1)
(gdb) next

Breakpoint 1, main () at test2.c:11
11          printf("current i=%f\n", i);
```

- i의 값이 for문의 조건을 만족하지 못할 경우 무한루프가 발생하게 되며, 따라서 i 값의 변화를 관찰할 필요가 있다.
- 변수의 내용을 확인하기 위해서 print명령을 사용, 다음 Breakpoint로 이동하기 위해서는 continue 명령을 사용한다.

```
(gdb) print i
$1 = 0.100000000000000001
(gdb) continue
Continuing.
current i=0.100000

Breakpoint 1, main () at test2.c:11
11          printf("current i=%f\n", i);
(gdb) print i
$2 = 0.200000000000000001
```

➤ 반복적으로 print 명령을 사용하여 변수의 내용을 확인하는 불편을 줄이기 위해 watch 명령을 사용하여 Watchpoint 설정한다.

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb) info watchpoints
Num Type          Disp Enb Address      What
 1 breakpoint      keep y  0x080483e8 in main at test2.c:11
   breakpoint already hit 3 times
 2 hw watchpoint  keep y          i
```

➤ watch명령을 사용하여 Watchpoint를 설정한 경우 지정된 변수가 변경되면 Tracing이 자동으로 멈춘다. 따라서 이 경우 한 루프에서 2번의 멈춤이 발생하게 된다. (line 8, 10)

**4190.102A Computer Programming
(2007 Winter)**

- 이러한 중복된 멈춤을 방지하기 위해서 disable 명령을 사용하여 잠시 line 8에 대한 breakpoint를 해제한다.

```
(gdb) disable 1
(gdb) info breakpoints
Num Type           Disp Enb Address      What
 1 breakpoint      keep n  0x080483e8 in main at test2.c:11
   breakpoint already hit 3 times
 2 hw watchpoint   keep y                i
```

- continue나 next, step 명령어를 이용하여 변수 i의 값을 Tracing한다.

```
(gdb) continue
Continuing.
current i=0.200000
Hardware watchpoint 2: i

Old value = 0.200000000000000001
New value = 0.300000000000000004
0x08048415 in main () at test2.c:9
9          for(i = 0; i != 1; i += 0.1)
(gdb) continue
Continuing.
current i=0.300000
Hardware watchpoint 2: i

Old value = 0.300000000000000004
New value = 0.400000000000000002
0x08048415 in main () at test2.c:9
9          for(i = 0; i != 1; i += 0.1)
...
(gdb) continue
Continuing.
current i=0.900000
Hardware watchpoint 2: i

Old value = 0.899999999999999991
New value = 0.99999999999999989          1)
0x08048415 in main () at test2.c:9
9          for(i = 0; i != 1; i += 0.1)
(gdb) continue
Continuing.
current i=1.000000
Hardware watchpoint 2: i

Old value = 0.99999999999999989
New value = 1.09999999999999999          2)
0x08048415 in main () at test2.c:9
9          for(i = 0; i != 1; i += 0.1)
```

- 주어진 code에서 for문은 "i != 1"의 조건일 때 계속 반복된다. 1)에서 값을 살펴보면 0.99...으로 정확히1이 안됨을 알 수 있다.따라서 주어진 조건을 계속 만족하게 되어

4190.102A Computer Programming (2007 Winter)

다음 루프로 들어가며 이때부터 i 의 값은 1을 넘어가게 되어 주어진 조건을 영원히 만족하게 된다. 따라서 프로그램은 무한루프에 빠지게 된다.

- 문제 발생 원인

- “ $i += 0.1$ ” 에서 실수 0.1에 대해 살펴보면,

- 컴퓨터에서 숫자는 2진수로 표시된다. (e.g., $5 = 101(2)$, $11 = 1011(2)$)
- 실수도 마찬가지로 2진수로 표시되나, 표현 가능한 비트 수의 제한으로 인해 모든 실수에 대해 정확한 값을 표현할 수 없다. (precision이 존재하여 오차 발생할 수 있음)
- 따라서 “ $i += 0.1$ ” 의 문장에서 i 의 값이 올바르게(개발자가 의도한대로) 더해지지 않는다.

- vi등의 편집기를 이용하여 코드 수정

```
for(i = 0; i < 1; i += 0.1)
{
    printf("current i=%f\n", i);
    sleep(0);
}
```

■ GNU Make

1. make

- 복잡한 컴파일 작업을 자동화 해주는 유틸리티
- 불필요하게 다시 컴파일 해야 할 일을 줄여주므로 컴파일 시간을 절약
 - 여러 모듈로 이루어진 프로그램을 다시 컴파일 시 필요한 부분만 재 컴파일
- 프로그램에 알맞은 옵션을 주어 컴파일 하거나 최신 버전의 프로그램모듈, 라이브러리와 링크 가능

2. Makefile

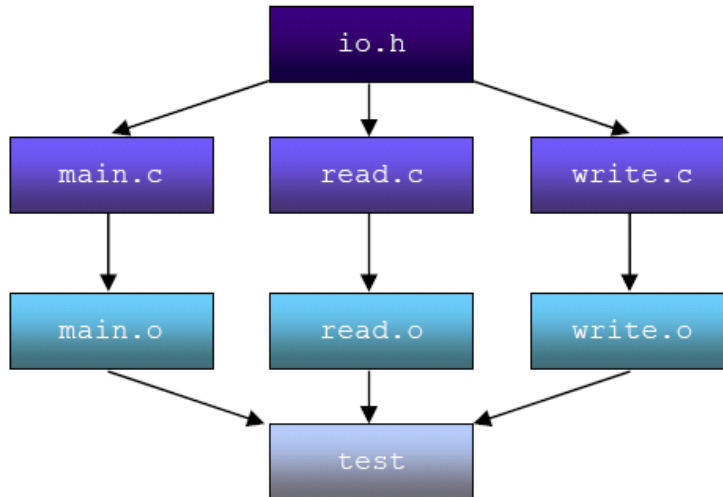
- 컴파일 시 필요한 규칙을 명시한 일종의 스크립트 파일
- Makefile의 내부구조

```
target ... : dependencies ...  
[TAB] command  
...  
...
```

- target : command가 수행되어서 나온 결과파일명 (or 수행할 작업 명)
 - 목적 파일(object file)이나 실행 파일
- dependencies: target을 만들기 위해 필요한 파일.
 - B가 변경되었을 때 A도 바뀐다면, A는 B에 dependent하다.
 - ex) test.o : test.h test.c
 - test.o는 소스 파일 test.c와 test.h에 dependent하다.
 - 따라서 test.c나 test.h가 변경되면 test.o도 변경되어야 한다.
- command: target 파일을 만들기 위하여 수행할 (컴파일) 명령어
 - 반드시 TAB으로 시작하여야 함
 - 일반적으로 셸에서 쓸 수 있는 모든 명령어들을 사용할 수 있다.
- 간단한 Makefile

```
test : read.o write.o main.o  
      gcc -o test read.o write.o main.o  
  
main.o : io.h main.c  
       gcc -c main.c  
  
read.o : io.h read.c  
       gcc -c read.c  
  
write.o: io.h write.c  
       gcc -c write.c
```

- Dependency graph



- 매크로의 사용
 - 명령을 단순화하기 위하여 매크로 지정 가능

```
# read.o write.o main.o를 OBJECTS라는 매크로로 지정  
OBJECTS = read.o write.o main.o
```

```
# 매크로를 호출할 시에는 $기호를 써주고 소괄호나 중괄호로 둘러싸야 함  
test : $(OBJECTS)  
      gcc -o test $(OBJECTS)
```

- 레이블의 사용
 - target-name을 label로 지정하여 해당 명령을 수행
 - dependency는 명시 안 해도 됨

```
clean :  
      rm $(OBJECTS)
```

3. Abbreviations (Internal Macro)

- \$* : 확장자가 없는 현재의 목표 파일 (Target)

```
main.o : main.c io.h  
      gcc -c $*.c
```

- @\$: 현재의 목표 파일 (Target)

4190.102A Computer Programming (2007 Winter)

```
test : $(OBJS)
      gcc -o $@ $*.c
```

- \$< : 현재 target보다 최근에 변경된 dependency 항목 리스트

```
main.o : io.h main.c
        gcc -c $<
```

- \$^ : 현재 target의 dependency 항목 리스트

```
test : main.o read.o write.o
      gcc -o $@ $^
```

4. Suffix Rule

- make가 확장자를 보고, 그에 따라 적절한 연산을 수행하는 규칙
- .SUFFIXES : .c .o
 - .c, .o 확장자를 가진 파일들을 규칙에 의거하여 처리하게 해주는 매크로

```
# 확장자 리스트에 .c .o를 추가한다
.SUFFIXES : .c .o
(생략)
# make가 .c를 찾아서 컴파일하고 .o를 build하는 규칙을 정의
.c.o :
      gcc -c -o $@ $*.c (또는 gcc -c -o $@ $<)
```

5. 실습

- 컴파일 할 소스파일 복사

```
$cp /home/comp-ta/make/*.c ./
```

- Makefile (Version 1)

```
$ vi makefile
diary : memo.o calendar.o main.o
      gcc -o diary memo.o calendar.o main.o

memo.o : diary.h memo.c
      gcc -c -o memo.o memo.c

calendar.o : diary.h calendar.c
      gcc -c -o calendar.o calendar.c

main.o : diary.h main.c
      gcc -c -o main.o main.c

clean :
      rm -rf *.o diary
```

4190.102A Computer Programming (2007 Winter)

- make 이용 컴파일 한 후 실행파일 실행

```
$ make
$ ./diary
```

- 소스코드 수정 후 컴파일

```
$ vi diary.h
(DIARY_NAME 변경 후 저장)
$ make
$ vi memo.c
(printf에서 출력부분 일부 변경 후 저장)
$ make
```

- Makefile 상에서 diary.h는 dependency graph 상에서 최상위에 위치한다. 다시 말하면, diary.h가 바뀌게 되면 모든 소스파일들이 재 컴파일 / Build되어야 하는 것이다.
- 하지만, memo.c만을 수정했을 경우 memo.c에 의존하고 있는 memo.o만이 재 컴파일 / Build된다.

- 필요없는 object 파일 삭제

- 정의한 레이블 clean 사용

```
$ make clean
```

- Makefile (Version 2)

```
# 확장자 규칙 사용
.SUFFIXES: .c .o
.c.o :
    gcc -c -o $@ $<

diary : memo.o calendar.o main.o
    gcc -o diary memo.o calendar.o main.o

memo.o : diary.h memo.c
calendar.o : diary.h calendar.c
main.o : diary.h main.c

clean :
    rm -rf *.o diary
```

6. 참고자료

- http://wiki.kldp.org/KoreanDoc/html/gcc_and_make/gcc_and_make.html
- <http://wiki.kldp.org/KoreanDoc/html/GNU-Make/GNU-Make.html>