Development Tools

Tracing and Profiling Tools

- **Gdb**
- **Gdb example**
- **strace**
- **ltrace**
- **valgrind(1) :** 메모리 누수 검출하기
- **valgrind(2) :** 비정상적인 메모리 접근 검출하기

GNU Debugger (GDB)

- **GDB**
  - 다양한 아키텍처와 CPU를 위한 낮은 수준의 종속적인 디버깅 까지도 지원한다.
  - 임베디드 리눅스 개발 환경에서는 교차-디버거로 컴파일된 GDB를 이용한다.
    - 디버거 자체는 개발 호스트 시스템에서 실행되지만, 디버거는 컴파일 시점에 주어진 아키텍처를 위한 바이너리 실행 파일을 인식하도록 되어 있는 것을 의미.

도입

- 디버깅
  - 프로그램의 논리적인 오류를 찾아내는 과정
  - 프로그램 수행 중에 내부적인 동작을 파악
  - 프로그램 수행 중 오류 발생으로 중단된 원인 파악

- **C/C++** 디버깅 도구인 **GDB debugger** 소개
  - 프로그램을 수행하거나 정지 시킬 수 있음
  - 수행 도중 변수 값을 알아 볼 수 있고 프로그램 수행 과정을 추적
  - 프로그램 수행 속도가 늦어 질 수 있음

디버깅을 위한 컴파일

- **GDB를 사용하려면 반드시 gcc 컴파일러에서 –g 옵션을 추가함**

```
gcc -g file1.c file2.c file3.o
```

  - file1.c file2.c는 소스 레벨의 디버깅 가능
  - file3.o는 오브젝트 파일을 만들 때 –g로 컴파일 한 경우에 한해서 소스 레벨의 디버깅 가능

- **최적화 옵션 –O와 동시에 사용 가능**
  - 최적화는 실행 프로그램에 변동을 가져옴
    - 예: 변수 또는 코드가 사라 질 수 있음
  - 추천: 최적화 없이 우선 완전하게 동작하는 프로그램 개발 후 최적화 적용
    - 최적화 사용시 프로그램이 비정상적으로 동작할 수 있음을 명심

GDB 시작하기

- **gdb 명령으로 시작**
  - gdb ex-prog core-dump
  - 실행 파일 'ex-prog' 를 디버깅
  - 프로그램 오류에 의하여 중단된 프로세스의 덤프 파일 'core-dump'를 디버깅

- **gdb 옵션들**

  -d dir : 소스 파일이 있는 디렉토리 지정

  -x file : gdb 시작 후 file에 있는 gdb 명령어 우선 실행

  -c number : 실행 중인 프로세스의 pid가 number인 프로세스를 디버깅

  -batch : -x file로 지정된 gdb 명령 수행 후 gdb 종료

- **gdb 종료**
  - 명령어 모드에서 quit 또는 q

GDB 실행 에러 & 해결 방안

- **libg.a를 찾을 수 없다는 에러가 발생하는 경우**
  - libc 패키지가 설치되어 있지 않을 경우
  - libc 소스 코드를 다운 받아 설치

  - /usr/lib/libg.a가 존재하지 않는 경우
  - /usr/lib/libc.a를 /usr/lib/libg.a로 링크 시켜도 됨

  - 사용자 Path에 /usr/lib/가 포함되어 있지 않을 경우
  - 사용자의 Path 환경에 /usr/lib/를 포함시킴

GDB 명령어 기본

- **gdb** 몇 개만 사용해도 디버깅을 잘 할 수 있음
- 프로그램을 수행하고 정지 시키고 그리고 변수 값을 알아 보는 일이 대부분임
- 명령어를 잘 모른 다면 **help** 명령어 사용
- 명령어 입력 중 **TAB** 키를 사용하면 가능한 입력 나열
  - 만약 가능한 입력이 하나 밖에 없으면 자동으로 명령어를 확장해 줌

GDB 명령어: 파일 내용 출력

- **list** 명령어는 실행하고 있는 부분의 소스를 출력

```
(gdb) list
```

  - 프로그램 실행 전에는 main 함수의 내용을 보여 줌

- 특정 부분을 보려면 소스 프로그램의 줄 번호를 입력함

```
(gdb) list 90
```

- 또는 특정 함수의 소스 코드를 보려면 함수의 이름 입력

```
(gdb) list badfunc
```

GDB 명령어: 프로그램 수행

- **run 명령어: 디버그 할 프로그램 수행**
  - 프로그램에 전달 할 인자 지정 가능

```
(gdb) run -b < invalues > outtable
```

  - 다음 run 명령어에서 인자를 주지 않으면 이전의 인자 그대로 사용
- **set arg 명령을 사용하여 인자를 바꾸거나 설정할 수 있음**

```
(gdb) set args -b < invalues > outtable
```

  - Show args로 설정된 인자를 알아 볼 수 있음

```
(gdb) show args
```

- 실행 디렉토리 변경
  - GDB의 실행 디렉토리를 이어 받으나 cd 명령으로 바꿀 수 있음
- 환경 변수 설정 변경
  - GDB의 환경 변수를 이어 받으나 set environment 명령으로 바꿀 수 있음
- 터미널 입출력
  - GDB의 터미널 입출력을 이어 받으나 tty 명령으로 바꿀 수 있음
- 이미 실행 중인 프로세스
  - Attach, detach 명령으로 이미 실행 중인 프로세스 디버기 가능
- 다중 쓰레드 디버깅
  - Info thread 명령어로 쓰레드를 알아보고 thread number 명령어로 특정 쓰레드 디버깅 가능

GDB 명령어: 프로그램 정지

- **break 명령어로 프로그램 정지 지점 설정**

  `(gdb) break line-number`

  - 소스의 특정 위치인 주어진 줄, line-number에 오면 프로그램 정지

  `(gdb) break func-name`

  - 소스의 특정 함수, func-name에 오면 프로그램 중지

  `(gdb) break line-or-func if condition`

  - 소스의 특정 위치에서 조전, condition을 만족하면 프로그램 정지

- **watch 명령어로 특정 조건을 만족하면 프로그램 정지**

  `(gdb) watch condition`

  - 프로그램의 속도가 상당히 느려짐
  - 일반적으로 watch를 사용하여 프로그램 오류을 대략적인 위치를 알아내고 break를 사용하여 정확한 위치를 추적함

GDB 명령어: 프로그램 상태 파악

- 프로그램을 정지시켜 놓고 상태를 파악하여 오류를 알아냄
  - whatis 명령어로 변수의 타입을 알아 봄

```
(gdb) whatis variable
```

  - print 명령어로 변수의 값을 알아 봄

```
(gdb) print variable
```

- 고급 **print** 명령어
  - Print 명령어의 variable은 C 언어의 구문을 그대로 사용할 수 있음

```
(gdb) print a->member
```

  - 함수의 결과 값도 함수를 호출하여 알아 볼 수 있음

```
(gdb) print function(args,…)
```

    - 전역 변수가 변경 될 수 있음
  - @을 사용하여 배열의 길이를 지정할 수 있음

```
(gdb) print *p@10
```

  - ::를 사용하여 범위(scope)를 지정할 수 있음

```
(gdb) print func::var-x
```

  - /x,u,o,c,etc 를 사용하여 출력 형태를 지정할 수 있음

```
(gdb) print/x var-x
```

- **where** 명령어를 사용하여 프로그램의 호출 상태 파악하기
- **call stack**의 위 아래로 **up, down** 명령어를 사용하여 이동 가능

GDB 명령어: 정지점 관리

- 복수개의 정지점 설정 가능
- 정지점은 **info breakpoints** 를 사용하여 설정된 정지점 알아 볼 수 있음
- **delete break-number** 명령을 사용하여 정지점 제거 가능
  - 인자를 주지 않으면 모든 정지점 지움
- **disable, enable**을 사용하여 정지점을 잠시 유효 및 무효화 시킬 수 있음

GDB 명령어: 계속 실행

- **continue** 명령어를 사용하여 프로그램 계속 수행 가능
- **step** 명령어를 통해서 한 줄씩 수행 가능
  - 함수를 만나면 함수로 들어 가서 한 줄 수행
- **next** 명령어를 통해서 한 줄씩 수행 가능
  - 함수도 한 줄로 인식하여 함수 안으로 들어가지 않음
- **finish** 현재 실행 중인 함수의 끝까지 실행하고 멈춤
- **return value** 은 현재 실행 중인 함수를 취소하고 **value**를 리턴 값으로 사용함

GDB 명령어: 명령어 자동 실행

- command 명령어을 사용하여 정지시 자동으로 실행하는 명령어 설정

  (gdb) command *break-number*

  …list-of-command

  …list-of-command

  End

- display 명령어를 사용하여 프로그램이 멈출 때마다 변수 값 출력

  (gdb) display *variable*

GDB 명령어: 시그널 설정

- Handle 명령어를 사용하여 디버깅 프로그램에 보내는 시그널 제어
  (gdb) handle *signal action-list*
  - signal은 SIGINT와 같은 시스템에서 정한 시그널
  - Action-list는 다음과 같은 의미를 가짐
    - nostop: 시그널을 받았을 때 프로그램에 넘겨주지 않고 정지 시키지도 않음
    - stop: 시그널을 받았을 때 프로그램을 정지 시키고 디버그 모드로 전환
    - print: 시그널을 받았을 때 메시지 출력
    - pass: 프로그램에 시그널을 넘겨 줌
    - nopass: 프로그램에 시그널을 넘겨주지 않음
  - (gdb) handle SIGINT stop print

- Singal 명령을 통하여 프로그램에 시그널 발생
  (gdb) signal 2

GDB 명령어 : 정리

list	현재 위치에서 소스 파일의 내용을 10줄 보여준다 list 2, 15 : 소스 파일의2 ~ 15 까지를 보여준다.
run	프로그램을 시작한다.(break가 있다면 break까지 실행) run arg : 새로운 인수를 가지고 프로그램을 시작한다. arg는 “*”나 “[…]”를 포함할 수도 있다. 쉘의 사용까지도 확장될 수 있다. “<”,“>” , “>>”같은 입출력 방향 재지정기호도 또한 허용된다.
break	특정 라인이나 함수에 정지점을 설정한다. break function : 현재 파일 안의 함수 function에 정지점을 설정한다. break file:function : 파일file안의 function에 정지점을 설정한다. watch : 감시점 설정(감시점은 어떤사건이 일어날 때에만 작동한다) until : 실행중 line까지 실행
clear	특정 라인이나 함수에 있던 정지점을 삭제한다.
delete	몇몇 정지점이나 자동으로 출력되는 표현을 삭제한다.
next	다음 행을 수행한다. 서브루틴을 호출하면서 계속 수행한다. 호출이 발생하지 않으면 step와 같다. next n : 이를 n번 수행하라는 의미

GDB 명령어 : 정리

step	한 줄씩 실행 시킨다. 함수를 포함하고 있으면 함수 내부로 들어가서 한 줄씩 실행시킨다.
print	print expr : 수식의 값을 보여준다.
display	현재 display된 명령의 목록을 보여준다.
bt	프로그램 스택을 보여준다. (backtrace)
kill	디버깅 중인 프로그램의 실행을 취소한다.
file	file program : 디버깅할 프로그램으로서 파일을 사용한다.
cont	continue : 현재 위치에서 프로그램을 계속 실행한다.
help	명령에 관한 정보를 보여주거나 일반적인 정보를 보여준다.
quit	gdb에서 빠져나간다.

Core file 분석하기

- 예기치 못한 일이 발생하여 비정상적인 종료가 발생할 때 운영체제는 디스크에 **Core file**을 남김
- 프로세스의 메모리 이미지를 **dump**한 것
- **Core file**을 **gdb**와 함께 사용하여 프로그램의 상태를 조사하고 실패 원인을 규명할 수 있음
- 메모리에 관한 문제는 **Checker** 패키지를 사용하여 예방 가능함
- 메모리 **fault**를 일으키는 경우에는 메모리가 충돌하면서 파일을 **dump**함
- **Core file** 은 일반적으로 프로세스를 실행시킨 현재 작업 디렉터리에 생성되지만 프로그램 내에서 작업 디렉터리를 바꾸는 경우도 있음

Core file 분석하기 (cont.)

- 일반적으로 Linux는 부팅 시에 Core file 을 만들지 않도록 설정되어 있음
- Core file 생성을 가능케 하려고 한다면 그것을 다시 가능케 하는 셸의 내장 명령을 사용함

- C shell 호환 shell(tcsh)
  - % limit core unlimited

- Bourne shell류(sh , bash , zsh , pdksh)
  - $ ulimit –c unlimited

- 코어파일을 함께 사용하기 위해선 다음과 같이 한다.
  - % gdb program core

실행 중인 프로그램 디버깅하기

- **gdb**는 이미 실행중인 프로그램도 디버깅할 수 있음
- 프로세스 실행을 가로채고 조사한 뒤 다시 원래 상태로 실행하도록 할 수 있음
- **attach**명령을 사용하여 실행중인 프로세서에 **gdb**를 붙임
- **attach** 명령을 사용하기 위해서는 프로세스에 해당하는 실행 프로그램에 허가권을 가지고 있어야 함
- 예를 들어 프로세스 **ID 254**번으로 실행 중인 **pgmseq** 프로그램이 있다면 다음과 같이 입력
  - % gdb pgmseq
  - % attach 254
  - % gdb pgmseq 254

실행 중인 프로그램 디버깅하기 *(cont.)*

- 일단 **gdb**가 실행 중인 프로세스에 부착되면 프로그램을 일시 중지 시키고 **gdb**명령을 사용할 수 있도록 제어권을 가져온 옴
- **break**를 사용하여 중지점을 사용할 수 있고 중지점에 이를 때까지 실행하도록 **continue** 명령을 사용할 수 있음
- **detach**명령을 사용하여 **gdb**를 실행 중인 프로세스에서 떼어 냄
- 필요에 따라 다른 프로세스에 대하여 **attach**명령을 사용할 수 있음

GDB debugger 실습

- **% gdb**
  - gdb를 먼저 실행 후 file이라는 명령으로 program을 부름
- **% gdb program**
  - 일반적인 방법
- **% gdb program core**
  - 코어파일을 사용할 때 동시에 인자로 줌
- **% gdb program 1234**
  - 실행중인 프로세스를 디버그 하려면 프로세스 ID를 두 번째 인자로 주면 된다. 이 명령은 gdb를 (‘1234’ 란 이름의 파일이 없다면) 프로세스 1234에 접속시킨다.(gdb는 core파일을 먼저 찾는다.)

GDB debugger 실습

- 실행절차
  - % gcc -g test.c -o test
  - % gdb test

```
[root@esp lib]# gdb
GNU gdb Red Hat Linux (6.6-45.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

실습 예제 1

- % vim test.c

```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5         int i;
 6         double j;
 7
 8         for(i=0;i<5;i++) {
 9                 j=i/2+i;
10                 printf("j is %f \n", j);
11         }
12         return 0;
13 }
```

- % gcc –g test.c –o test
- % ./test

실습 예제 *1(cont.)*

- **% gdb test**
- **(gdb) list** // **list**는 소스 내용을 **10**줄씩 보여준다.

```
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               int i;
6               double j;
7
8               for(i=0;i<5;i++) {
9                       j=i/2+i;
10                      printf("j is %f \n", j);
(gdb)
```

- **(gdb) b 8** // **break 8 : line 8**에 **breakpoint**를 잡는다.

```
(gdb) b 8
Breakpoint 1 at 0x80483d5: file test.c, line 8.
```

실습 예제 *1(cont.)*

- **(gdb) r** **// breakpoint** 전까지 프로그램 실행

```
(gdb) r
Starting program: /root/program/test

Breakpoint 1, main () at test.c:8
8                for(i=0;i<5;i++) {
Missing separate debuginfos, use: debuginfo-install glibc.i686
```

- **(gdb) s** **// step :** 한 줄 실행시킨다**.**

```
(gdb) s
9                        j=i/2+i;
```

- **(gdb) s**

```
(gdb) s
10                       printf("j is %f \n", j);
```

- **(gdb) p j** **// print j : j**의 값을 본다**.**

```
(gdb) p j
$1 = 0
```

실습 예제 *1(cont.)*

- **(gdb) n // next**
- **(gdb) display i // 변수 i를 display 목록에 추가**
- **(gdb) display j //변수 j를 display 목록에 추가**
- **(gdb) n // next**
- **(gdb) quit //종료**

```
(gdb) n
j is 0.000000
8               for(i=0;i<5;i++) {
(gdb) display i
1: i = 0
(gdb) display j
2: j = 0
(gdb) n
9                       j=i/2+i;
2: j = 0
1: i = 1
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

실습 예제 2

- % vim hab.c

```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5         int a, b, dab;
 6         printf("input num a, b : ");
 7         scanf("%d %d", &a, &b);
 8         dab = hab(a,b);
 9         printf("\n %d + %d = %d \n", a, b, dab);
10 }
11
12 int hab(int x, int y)
13 {
14         return (x+y);
15 }
```

- % gcc –g hab.c –o hab

실습 예제 *2 (cont.)*

- 이 예제는 여러 곳에서 호출되는 함수 안에서 충돌이 일어날 경우 *함수가 어디로부터 호출되었는지 그리고 어떤 상황에서 충돌이 일어났는지 파악하고자 함*
- **backtrace (bt)** 명령을 이용하면 충돌이 일어난 시점에서 프로그램의 현재 호출 스택**(call stack)** 상태를 볼 수 있음**.**
- 호출 스택은 현재 함수까지 이르는 호출 목록이다**.** 함수를 호출할 때마다 보관된 레지스터 값**,** 함수 전달 인수**,** 지역 변수 등의 자료를 스택에 **push**한다**.** 이렇게 해서 각 함수들은 스택상에 일정 공간을 차지한다**.**
- 특정함수에 대하여 스택에서 사용되고 있는 메모리 부분을 스택프레임**(frame)**이라 부르며 호출 스택은 이러한 스택 프레임을 순서대로 정렬한 목록이다**.**

실습 예제 2 (cont.)

- % gdb hab
- (gdb) b 8
- (gdb) r // 숫자 3 4 입력

```
(gdb) b 8
Breakpoint 1 at 0x804842b: file hab.c, line 8.
(gdb) r
Starting program: /root/program/hab
input num a, b : 3 4

Breakpoint 1, main () at hab.c:8
8               dab = hab(a,b);
Missing separate debuginfos, use: debuginfo-install glibc.i686
```

- (gdb) bt // 현재 스택에 main이 있다.

```
(gdb) bt
#0  main () at hab.c:8
```

실습 예제 *2 (cont.)*

- **(gdb) s**

```
(gdb) s
hab (x=3, y=4) at hab.c:14
14              return (x+y);
```

- **(gdb) bt // 지금은 스택에 hab이 있다.**

```
(gdb) bt
#0  hab (x=3, y=4) at hab.c:14
#1  0x0804843d in main () at hab.c:8
```

- **(gdb) frame 0 // hab**의 상태를 점검하기 위해서 스택 프레임**0**번으로 이동

```
(gdb) frame 0
#0  hab (x=3, y=4) at hab.c:14
14              return (x+y);
```

실습 예제 *2 (cont.)*

- **(gdb) up // hab**이 어떻게 호출되었는가를 보기 위하여 상위 스택프레임으로 이동

```
(gdb) up
#1  0x0804843d in main () at hab.c:8
8               dab = hab(a,b);
```

- **(gdb) info program // 프로그램의 실행 상태를 보여 준다.**

```
(gdb) info program
        Using the running image of child process 3192.
Program stopped at 0x804846d.
It stopped after being stepped.
```

- **(gdb) info locals // 현재 함수 내에서 모든 지역 변수 이름과 값을 출력한다.**

```
(gdb) info locals
a = 3
b = 4
dab = 2230592
```

실습 예제 2 (cont.)

- **(gdb) info variables //** 소스파일 순서대로 프로그램 내에 알려져 있는 모든 변수를 출력한다.

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x08048544  _fp_hw
0x08048548  _IO_stdin_used
0x0804854c  __dso_handle
0x080485ec  __FRAME_END__
0x080495f0  __CTOR_LIST__
0x080495f0  __init_array_end
0x080495f0  __init_array_start
0x080495f4  __CTOR_END__
0x080495f8  __DTOR_LIST__
0x080495fc  __DTOR_END__
0x08049600  __JCR_END__
0x08049600  __JCR_LIST__
0x08049604  _DYNAMIC
0x080496d0  _GLOBAL_OFFSET_TABLE_
0x080496ec  __data_start
0x080496ec  data_start
0x080496f0  dtor_idx.5647
---Type <return> to continue, or q <return> to quit---
```

실습 예제 *2 (cont.)*

- **(gdb) info frame // 현재 프레임 정보를 보여 준다.**

```
(gdb) info frame
Stack level 1, frame at 0xbfd7d550:
 eip = 0x804843d in main (hab.c:8); saved eip 0x4c12390
 caller of frame at 0xbfd7d520
 source language c.
 Arglist at 0xbfd7d548, args:
 Locals at 0xbfd7d548, Previous frame's sp at 0xbfd7d544
 Saved registers:
  ebp at 0xbfd7d548, eip at 0xbfd7d54c
```

실습 예제 3

- **% vim coredebug.c**

```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5         char *bug=NULL;
 6
 7         strcpy(bug,"debug");
 8         printf("bug is %s \n", bug);
 9
10         return 0;
11 }
```

- **% gcc –g coredebug.c –o coredebug**
- **% coredebug**
  - //세그먼트 오류가 발생된 후 core.xxxx 파일이 생성됨

실습 예제 *3 (cont.)*

- **% gdb coredebug**
- **(gdb) b 7**
- **(gdb) r**
- **(gdb) p bug**

```
(gdb) b 7
Breakpoint 1 at 0x80483dc: file coredebug.c, line 7.
(gdb) r
Starting program: /root/program/coredebug

Breakpoint 1, main () at coredebug.c:7
7               strcpy(bug,"debug");
Missing separate debuginfos, use: debuginfo-install glibc.i686
(gdb) p bug
$1 = 0x0
```

- **gdb** 에서 **0x0**는 **NULL**이다**.** 즉 번지가 없다**.**

실습 예제 *3 (cont.)*

- **(gdb) s**

```
(gdb) s

Program received signal SIGSEGV, Segmentation fault.
0x080483df in main () at coredebug.c:7
7               strcpy(bug,"debug");
```

- **line 7**번 **strcpy**에서 **segmentation fault**가 발생한 것을 알 수 있다**.**
- **bug**에 번지를 할당하면 에러를 해결할 수 있다**.**

실습 예제 *3 (cont.)*

- **% gdb coredebug core.xxxx**

```
[root@esp program]# gdb coredebug core.4253
GNU gdb Red Hat Linux (6.6-45.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".

warning: core file may not match specified executable file.

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./core'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483df in main () at coredebug.c:7
7               strcpy(bug,"debug");
```

실습 예제 *3 (cont.)*

- **gdb**는 **signal 11** 번과 함께 **core file**일이 생성되었음을 알려 준다.
- 여기서는 허가 받지 않은 메모리 공간에 읽기, 쓰기를 시도했기 때문에 커널이 프로세스에게 **signal 11**을 보냈다.
- 이 시그널로 인해 프로세스는 종료하면서 코어 파일을 덤프한다.

그 외 기능

- 프롬프트 변경
  - (gdb) set prompt haha: //haha:로 프롬프트 변경
- 리스트 라인 수 변경
  - (gdb) set listsize 20 // list 출력 라인 수를 20으로 변경
- 특정한 부분으로 이동
  - (gdb) | main // 문자 패턴 main으로 이동
  - (gdb) | 25// 25라인으로 이동
  - (gdb) | file.c:func // 파일명 file.c의 func이란 함수로 이동

그 외 기능 *(cont.)*

- 브레이크 포인트 추가
  - (gdb) b 19 // line 19에 breakpoint 추가
  - (gdb) b main // 문자패턴 main에 breakpoint 추가
  - (gdb) b file.c:10 // file.c의 10번째 줄에 breakpoint 추가
  - (gdb) b -2 // 현재 기준 -2번째 줄에 breakpoint 추가
  - (gdb) b 10 if tmp ==0 // tmp 값이 0일 때 10번째 줄에 breakpoint 추가..
- 브레이크 포인트 삭제
  - (gdb) cl hello
  - (gdb) cl 10
  - (gdb) cl file.c:20
  - (gdb) d // 모든 breakpoint 삭제

그 외 기능 *(cont.)*

- 브레이크 포인트 확인
  - (gdb) info breakpoints // 현재 breakpoint 확인
- 브레이크 포인트 활성화/비활성화
  - (gdb) enable 2
  - (gdb) disable 2
- 프로그램 수행 명령
  - (gdb) r (run) // 프로그램 수행
  - (gdb) r arg1 arg2 // 프로그램 수행 시 arg1과 arg2를 전달
  - (gdb) k (kill) // 프로그램 수행 종료
  - (gdb) s (step) // 현재 행 수행 후 정지, 함수 호출 시 함수 내부로 들어감
  - (gdb) n (next) // 현재 행 수행 후 정지, 함수 호출 시 함수 수행 후 다음 행으로 감

그 외 기능 *(cont.)*

- (gdb) c (continue) // 브레이크 포인트를 만날 때까지 계속 진행
- (gdb) u (until) // 현재 루프를 빠져나감
- (gdb) finish // 현재 함수를 수행하고 빠져나감
- (gdb) return // 현재 함수를 수행하지 않고 빠져나감
- (gdb) return arg1 // 현재 함수를 수행하지 않고 빠져나가는데... arg1을 리턴
- (gdb) si // 현재의 instruction을 수행, 함수 호출 시 함수 내부로 들어감
- (gdb) ni // 현재의 instruction을 수행, 함수 호출 시 함수 내부로 들어가지 않음

그 외 기능 *(cont.)*

- 와치 포인트
  - 변수 값 이 바뀔 때마다 브레이크가 걸리면서 변수의 이전 값(old value)과 현재값(new value)을 출력한다.
  - (gdb) watch i // 변수 i에 와치 포인트를 설정.
- 변수와 레지스터 값 출력
  - (gdb) info locals // 현재 상태에서의 지역변수 출력
  - (gdb) info variables // 현재 상태에서의 전역변수 리스트 출력
  - (gdb) p (printf) // 개별 변수 값 출력
  - (gdb) p *pointer // pointer 변수 값 출력
  - (gdb) p *pointer@4 // pointer 구조체 배열 값 출력
    @ 다음에 오는 숫자는 배열의 크기
  - (gdb) p $eax // 레지스터 eax 값 출력

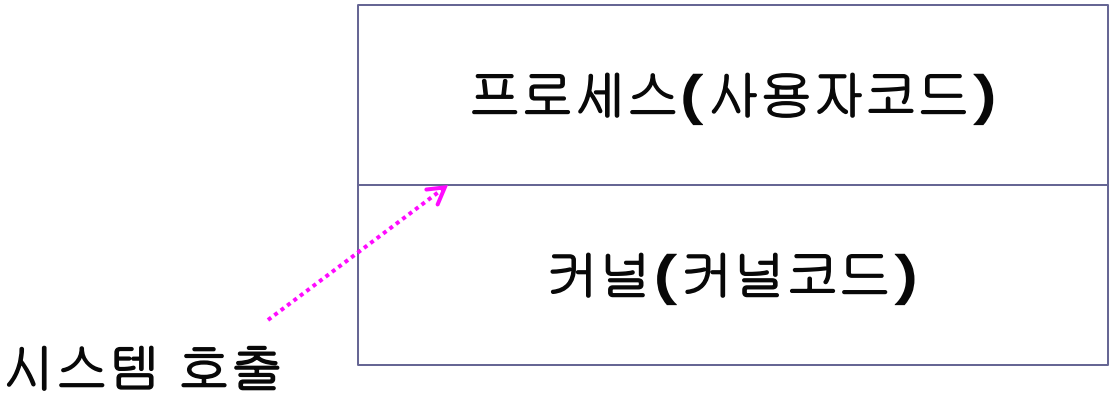
그 외 기능 *(cont.)*

- **display를 이용한 변수 값 출력**
  - 특정 변수 값을 display 목록에 추가하게 되면 p 명령어로 매번 확인하는 불편함 없이 s 명령어로 자동으로 확인
  - 변수의 범위를 벗어나면 자동으로 출력 중지
  - (gdb) display i // 변수 i의 값을 display 목록에 추가
  - (gdb) undisplay i // 변수 i의 값을 display 목록에서 삭제
  - (gdb) enable display 3 // 디스플레이 번호 3번을 활성화
  - (gdb) disable display 4 // 디스플레이 번호 4번을 비활성화
- **인스트럭션 레벨 디버깅**
  - (gdb) disass play //play함수에 대한 디스어셈블리.
  - (gdb) display $pc //현재의 instruction을 보여줌
  - $pc는 gdb내부 변수로서 현재 instruction의 위치를 가리키는 Program Counter(PC) 레지스터

그 외 기능 *(cont.)*

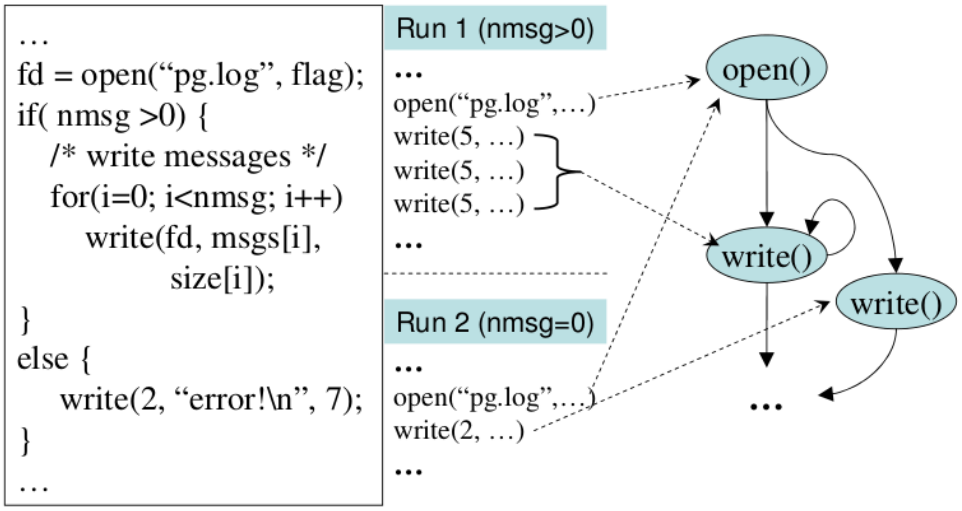
- 스택 상태 검사
  - (gdb) bt (backtrace) // 전체 스택 출력
  - (gdb) info f (frame) // 스택의 정보 출력
  - (gdb) info args // 함수가 호출 될 때 인자를 출력
  - (gdb) info locals // 함수의 지역변수를 출력
  - (gdb) frame 3 // 3번 스택 프레임 확인
  - (gdb) up // 한 단계 상위 스택 프레임으로 이동
  - (gdb) down// 한 단계 하위 스택 프레임으로 이동

- **strace**
  - Strace는 리눅스 어플리케이션 프로그램이 호출하는 커널 시스템 콜에 대한 정보를 캡처하여 표시해 주는 일을 한다.
  - 시스템 콜에 대한 정확한 정보를 확인할 수 있다.



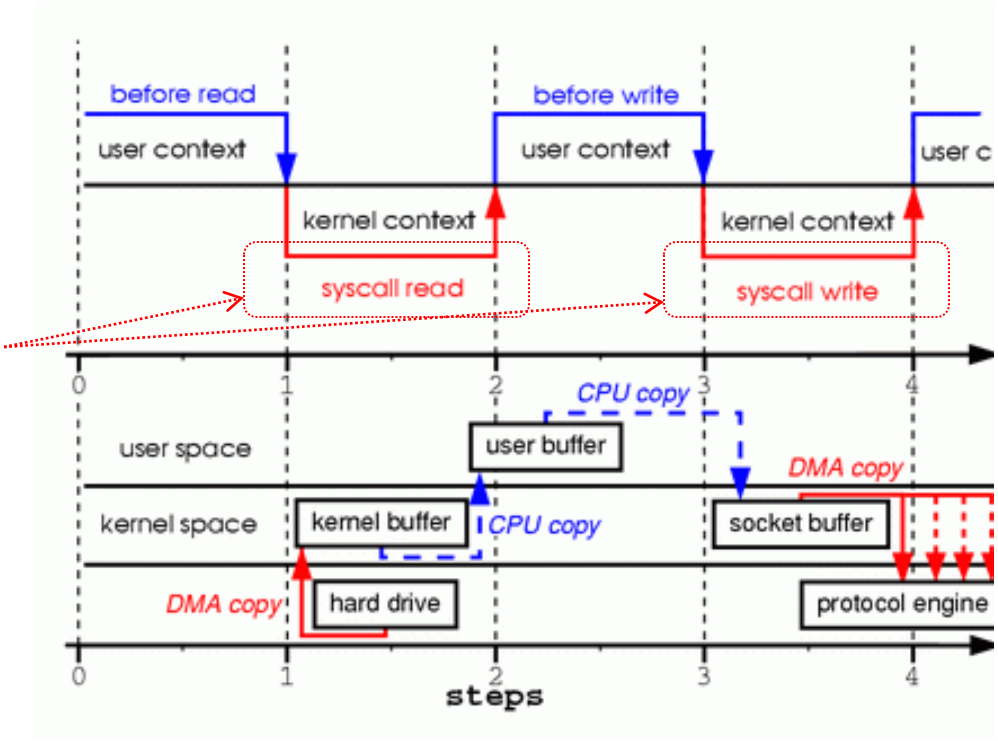
시스템 호출은 사용자와 커널 사이에 존재하는 층이다.

- 시스템 호출이란.. ?
  - 시스템 호출은 커널 내에서 실행되는 특수 함수이다.
  - 프로세스는 시스템 호출을 통해 디스크, 네트워크, 메모리와 같은 자원에 공평하고 안전하게 접근할 수 있다.
  - IPC와 같은 커널 서비스를 말한다.
    - 하드웨어 플랫폼에 따라 다르지만, 시스템 호출은 게이트 명령, 트랩명령, 소프트웨어 인터럽트 명령, 또는 기타 메커니즘을 사용하여 사용자 코드에서 커널로 전환한다. ***(함수 호출과 시스템 호출을 비교해 보면 이해하기 쉬움)***



- 시스템 호출 **vs** 함수 호출
  - 함수호출은 커널이 필요하지 않다.
  - 시스템호출은 커널 모드 전환이 필요하다.

시스템 **call** 모드
커널 모드이다.



- **strace 실습.**
  - 다음 예제는 간단한 프로그램을 사용하여 strace 사용법을 보여준다.
  - 프로그램은 **"읽기 전용"**인 파일을 열지 못하고 종료한다.

시스템 호출

```
1  #include<sys/types.h>
2  #include<sys/stat.h>
3  #include<fcntl.h>
4
5  int main()
6  {
7      int fd;
8      int i = 0;
9
10     fd = open("./temp/foo",O_RDONLY);
11     if (fd < 0)
12         i = 5;
13     else
14         i = 2;
15
16     return i;
17 }
18
```

읽기 전용이고
자동 생성이 아님.

- 실습 결과

프로세스가 커널로 호출하는 시스템 콜을 나타낸다

```
hicys76@hicys76:~/ex_unix_study/test$ gcc -g -o strac_test strac.c
hicys76@hicys76:~/ex_unix_study/test$ strace -o main.strace ./strac_test
hicys76@hicys76:~/ex_unix_study/test$ cat main.strace
execve("./strac_test", ["./strac_test"], [/* 53 vars */]) = 0
brk(0)                                  = 0x8d79000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb77bd000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=112939, ...}) = 0
mmap2(NULL, 112939, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb77a1000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220o\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1434180, ...}) = 0
mmap2(NULL, 1444360, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7640000
mprotect(0xb779a000, 4096, PROT_NONE)   = 0
mmap2(0xb779b000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15a) = 0xb779b000
mmap2(0xb779e000, 10760, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb779e000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb763f000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb763f8d0, limit:1048575, seg_32bit:1, contents:0, read_exec
ages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb779b000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb77dc000, 4096, PROT_READ)   = 0
munmap(0xb77a1000, 112939)              = 0
open("./temp/foo", O_RDONLY)            = -1 ENOENT (No such file or directory)
exit_group(5)                           = ?
hicys76@hicys76:~/ex_unix_study/test$
```

읽기 전용이고
자동 생성이 아님.

- **Example strace Output**

- **Profiling Using strace 1**

  strace –c ./test -> -c 옵션은 실행 **appl**을 대상으로 상위 레벨의 프로파일링을 생성해 준다**.**

```
hicys76@hicys76:~/ex_unix_study/test$ strace -T ./strac_test
execve("./strac_test", ["./strac_test"], [/* 53 vars */]) = 0 <0.000569>
brk(0)                                  = 0x9a2b000 <0.000071>
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory) <0.000054>
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb77ed000 <0.000058>
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory) <0.000076>
open("/etc/ld.so.cache", O_RDONLY)      = 3 <0.000064>
fstat64(3, {st_mode=S_IFREG|0644, st_size=112939, ...}) = 0 <0.000037>
mmap2(NULL, 112939, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb77d1000 <0.000041>
close(3)                                = 0 <0.000036>
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory) <0.000060>
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY) = 3 <0.000065>
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220o\1\0004\0\0\0"..., 512) = 512 <0.000040>
fstat64(3, {st_mode=S_IFREG|0755, st_size=1434180, ...}) = 0 <0.000036>
mmap2(NULL, 1444360, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7670000 <0.000041>
mprotect(0xb77ca000, 4096, PROT_NONE)   = 0 <0.000044>
mmap2(0xb77cb000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15a) = 0xb77cb000 <0.000052>
mmap2(0xb77ce000, 10760, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb77ce000 <0.000044>
close(3)                                = 0 <0.000116>
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb766f000 <0.000041>
set_thread_area({entry_number:-1 -> 6, base_addr:0xb766f8d0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, l
 seg_not_present:0, useable:1}) = 0 <0.000037>
mprotect(0xb77cb000, 8192, PROT_READ)   = 0 <0.000032>
mprotect(0x8049000, 4096, PROT_READ)    = 0 <0.000027>
mprotect(0xb780c000, 4096, PROT_READ)   = 0 <0.000028>
munmap(0xb77d1000, 112939)              = 0 <0.000039>
open("./temp/foo", O_RDONLY)            = -1 ENOENT (No such file or directory) <0.000048>
exit_group(5)                           = ?
hicys76@hicys76:~/ex_unix_study/test$
```

- **Profiling Using strace 1**
  - strace –c ./strace // 시스템 호출 항목 사이에서 흘러간 시간과 호출에 걸린 시간을 측정

```
hicys76@hicys76:~/ex_unix_study/test$ strace -c ./strac_test
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  -nan    0.000000           0         1           read
  -nan    0.000000           0         3         1 open
  -nan    0.000000           0         2           close
  -nan    0.000000           0         1           execve
  -nan    0.000000           0         3         3 access
  -nan    0.000000           0         1           brk
  -nan    0.000000           0         1           munmap
  -nan    0.000000           0         4           mprotect
  -nan    0.000000           0         6           mmap2
  -nan    0.000000           0         2           fstat64
  -nan    0.000000           0         1           set_thread_area
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    25         4 total
hicys76@hicys76:~/ex_unix_study/test$
```

- **Ltrace**
  - Ltrace는 공유 라이브러리의 함수호출을 추적하는 리눅스용 툴이다.
  - ltrace -o 로그파일[output] 실행프로그램

- **ltrace 예제**
  - 아래 예제와 같이 ltrace를 사용하면 공유 라이브러리의 함수호출을 추적할 수 있으므로 프로그램 디버깅 및 동작을 파악하는 데 유용하다.

Wget으로 https~의 컨텐츠를 얻을 때 사용되는 공유 라이브러리를 호출을 추적하고 있다.
(메시지를 파일로 출력하려면 –o 옵션을 이용한다.)

```
hicys76@hicys76:~/ex_unix_study/test$ ltrace -o log.txt wget https://www.codeblog.org/
--2011-12-28 05:15:28--  https://www.codeblog.org/
Resolving www.codeblog.org... 211.14.6.122
접속 www.codeblog.org|211.14.6.122|:443... 접속됨.
HTTP request sent, awaiting response... 200 OK
Length: 4575 (4.5K) [text/html]
Saving to: `index.html'

100%[=======================================================================================>] 4,575       --.-K/s   in 0.003s

2011-12-28 05:15:29 (1.34 MB/s) - `index.html' saved [4575/4575]

hicys76@hicys76:~/ex_unix_study/test$ grep SSL log.txt | head
SSL_library_init(0, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8)                          = 1
SSL_load_error_strings(0, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8)                    = 0
OPENSSL_add_all_algorithms_noconf(0, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8)         = 1
SSL_library_init(0, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8)                          = 1
SSLv23_client_method(0, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8)                      = 0xb77e85c0
SSL_CTX_new(0xb77e85c0, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8)                      = 0x9b22190
SSL_CTX_set_default_verify_paths(0x9b22190, 0xbf9c94d8, 4, 0xbf9c97a8, 0xbf9c97a8) = 1
SSL_CTX_load_verify_locations(0x9b22190, 0, 0, 0xbf9c97a8, 0xbf9c97a8)              = 0
SSL_CTX_set_verify(0x9b22190, 0, 0, 0xbf9c97a8, 0xbf9c97a8)                         = 0x9b22190
SSL_CTX_ctrl(0x9b22190, 33, 1, 0, 0xbf9c97a8)                                       = 1
hicys76@hicys76:~/ex_unix_study/test$
```

-p 옵션을 이용하면 기존 프로세스에 attach 할 수도 있다.

- **Profiling Using ltrace**
  - -c 옵션을 주면 라이브러리 함수에 대해서 호출된 횟수, 에러 횟수, 실행된 시간 등의 통계를 볼 수 있다.
  - 동적으로 링크된 라이브러리를 사용하도록 컴파일 된 프로그램만 사용가능

```
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
100.00    0.112742         112      1000 puts
------ ----------- ----------- --------- --------------------
100.00    0.112742                  1000 total
```

- **valgrind – Memory lake 검출하기**
  - Valgrind는 Linux/x86, Linux/amd64, Linux/ppc32에서 사용 가능한, 프로그램 동작을 동적으로 분석해 주는 툴이다.
  - 메모리 누수(memory lake), 잘못된 메모리 해제, 이중 해제(double free)
  - 다양한 종류의 비정상적인 메모리 접근
  - 멀티스레드 프로그램에서 메모리 접근 경합 (race condition)

- 명령행 옵션
- valgrind --leak-check=full --leak-resolution=high --show-reachable=yes ./valgrind_test1
  - --show-reachable=yes
    - 버그2와 같이 문제를 일으키지 않을 수도 있는메모리 누수도 보고한다.
  - --trace-children=yes
    - 분석중에 프로그램이 fork로 발생한 프로세스도 분석 대상이다.
  - --track-fds=yes
    - 닫지 않은 파일 지시자도 보고한다.
  - --error-limit=no
    - 에러 횟수 한계치를 초과해도 계속 분석하도록 한다.
  - --num-callers=<number>
    - 에러 위치까지 백트레이스를 몇 단계까지 지정한다.
  - --xml=yes
    - XML 형식으로 출력해준다.

- **valgrind 예제**

```
1
2  /* valgrind_test !! */
3
4  #include<cstdlib>
5
6  int *leaky_foo(void){
7      int *a = new int;
8      int *b = new int;  // bug1 메모리 누수
9      return a;
10 }
11
12
13 static int *c;
14 int main(int argc, char **argv){
15     c = leaky_foo(); // bug 2:메모리누수(프로그램이 종료할 때까지 c를 delet하지 않음)
16     char *d = (char*)std::malloc(sizeof(char)*10U);
17     delete[] d;      // bug 3: malloc으로 할당한 메모리를 delet해야함 (free필요)
18 }
```

- 출력결과

```
--4662:1:main
--4662:1:main
--4662:1:aspacem  <<< SHOW_SEGMENTS: Memory layout at client startup (24 segments, 4 segnames)
--4662:1:aspacem  ( 0) /usr/local/lib/valgrind/memcheck-x86-linux
--4662:1:aspacem  ( 1) /home/hicys76/ex_unix_study/test/valgrind_test/valgrind_test1
--4662:1:aspacem  ( 2) /lib/i386-linux-gnu/ld-2.13.so
--4662:1:aspacem    0: RSVN 0000000000-0003ffffff     64m ----- SmFixed
--4662:1:aspacem    1: file 0004000000-000401bfff  114688 r-x-- d=0x801 i=2501842 o=0       (2)
--4662:1:aspacem    2: file 000401c000-000401dfff    8192 rw--- d=0x801 i=2501842 o=110592  (2)
--4662:1:aspacem    3:      000401e000-0008047fff     64m
--4662:1:aspacem    4: file 0008048000-0008048fff    4096 r-x-- d=0x801 i=24564848 o=0       (1)
--4662:1:aspacem    5: file 0008049000-000804afff    8192 rw--- d=0x801 i=24564848 o=0       (1)
--4662:1:aspacem    6: anon 000804b000-000804bfff    4096 rwx--
--4662:1:aspacem    7: RSVN 000804c000-000884afff 8384512 ----- SmLower
--4662:1:aspacem    8:      000884b000-0037ffffff    759m
--4662:1:aspacem    9: FILE 0038000000-003803ffff  262144 r-x-- d=0x801 i=2291778 o=4096    (0)
--4662:1:aspacem   10: file 0038040000-0038040fff    4096 r-x-- d=0x801 i=2291778 o=266240  (0)
--4662:1:aspacem   11: FILE 0038041000-00381f1fff 1773568 r-x-- d=0x801 i=2291778 o=270336  (0)
--4662:1:aspacem   12:      00381f2000-00381f2fff    4096
--4662:1:aspacem   13: FILE 00381f3000-00381f4fff    8192 rw--- d=0x801 i=2291778 o=2043904 (0)
--4662:1:aspacem   14: ANON 00381f5000-0038caefff     10m rw---
--4662:1:aspacem   15:      0038caf000-0061d4cfff    656m
--4662:1:aspacem   16: RSVN 0061d4d000-0061d4dfff    4096 ----- SmFixed
--4662:1:aspacem   17: ANON 0061d4e000-00629cdfff     12m rwx--
--4662:1:aspacem   18:      00629ce000-00be29afff   1464m
--4662:1:aspacem   19: RSVN 00be29b000-00bea99fff 8384512 ----- SmUpper
--4662:1:aspacem   20: anon 00bea9a000-00bea9afff    4096 rwx--
--4662:1:aspacem   21:      00bea9b000-00bfa7afff     15m
--4662:1:aspacem   22: ANON 00bfa7b000-00bfa9bfff  135168 rw---
--4662:1:aspacem   23: RSVN 00bfa9c000-00ffffffff   1029m ----- SmFixed
--4662:1:aspacem  >>>
--4662:1:main
```

- **Error**에 관련된 보고

```
--4662:1:mallocfr newSuperblock at 0x42D9000 (pszB 4194288) owner CLIENT/client
==4662== Mismatched free() / delete / delete []
--4662:1:mallocfr newSuperblock at 0x64F2E000 (pszB   65520) owner VALGRIND/demangle
==4662==    at 0x40257E3: operator delete[](void*) (vg_replace_malloc.c:409)
==4662==    by 0x8048512: main (valgrind_test1.cpp:17)
==4662==  Address 0x42d9098 is 0 bytes inside a block of size 10 alloc'd
==4662==    at 0x4026230: malloc (vg_replace_malloc.c:236)
==4662==    by 0x80484FB: main (valgrind_test1.cpp:16)
==4662==
--4662:1:syswrap- thread_wrapper(tid=1): exit
--4662:1:syswrap- run_a_thread_NORETURN(tid=1): post-thread_wrapper
```

```
==4662==
==4662== ****ttttt
==4662== ***   start 2!!!!!!
==4662== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==4662==    at 0x40269FC: operator new(unsigned int) (vg_replace_malloc.c:255)
==4662==    by 0x80484C5: leaky_foo() (valgrind_test1.cpp:7)
==4662==    by 0x80484EA: main (valgrind_test1.cpp:15)
==4662==
==4662== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==4662==    at 0x40269FC: operator new(unsigned int) (vg_replace_malloc.c:255)
==4662==    by 0x80484D4: leaky_foo() (valgrind_test1.cpp:8)
==4662==    by 0x80484EA: main (valgrind_test1.cpp:15)
==4662==
==4662== LEAK SUMMARY:
==4662==    definitely lost: 4 bytes in 1 blocks
==4662==    indirectly lost: 0 bytes in 0 blocks
==4662==      possibly lost: 0 bytes in 0 blocks
==4662==    still reachable: 4 bytes in 1 blocks
==4662==         suppressed: 0 bytes in 0 blocks
==4662==
==4662== For counts of detected and suppressed errors, rerun with: -v
==4662== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 17 from 6)
--4662:1:core_os  VG_(terminate_NORETURN)(tid=1)
```

- **valgrind(2) – 비정상적인 메모리 접근 검출하기**
  - 초기화되지 않은 변수 사용
  - 범위를 벗어난 메모리 접근
  - 해제가 끝난 메모리 접근
  - 복사할 원본과 대상 중첩 (overlap)

- **valgrind(2)** 실습해보기

```
1  // horrible
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <unistd.h>
7
8  static char onbss[128];
9  int main(){
10     char onstack[128];
11     int uninitialized, dummy;
12     char *onheap = (char*)malloc(128);
13
14     dummy = onbss[128];
15     dummy = onstack[150];
16
17     if(uninitialized ==0){
18         printf("hello world\n");
19     }
20     close(uninitialized);
21
22     dummy = onheap[128];
23
24     free(onheap);
25     dummy = onheap[0];
26
27     strcpy(onstack, "build on to throw away ; you will anyway.");
28     strcpy(onstack,onstack + 1);
29
30     return 0;
31 }
```

valgrind(2) 결과 출력 1

초기화 되지 않은 변수 사용**: ->** 스택에 확보된 **uninitialized**를 초기화 하지 않은 채 사용하고 있다**.**

if(uninitialized) ~

```
--5118:1:mallocfr newSuperblock at 0x64D0D000 (pszB   65520) owner VALGRIND/errors
--5118:1:mallocfr newSuperblock at 0x41AB000 (pszB 4194288) owner CLIENT/client
==5118== Conditional jump or move depends on uninitialised value(s)
==5118==    at 0x8048551: main (valgrind_test2.c:17)
==5118==
==5118== Syscall param close(fd) contains uninitialised byte(s)
==5118==    at 0x4107DEC: __close_nocancel (syscall-template.S:82)
==5118==    by 0x405EE36: (below main) (libc-start.c:226)
==5118==
```

범위를 벗어난 메모리 접근 **–>** 힙에 **128**바이트만 확보했지만 **129**번째 바이트를 참조하고 있다**.**

free(onheap)

dummy = onheap[0];

```
==5118== Warning: invalid file descriptor -1096220640 in syscall close()
==5118== Invalid read of size 1
==5118==    at 0x8048572: main (valgrind_test2.c:22)
==5118==  Address 0x41ab0a8 is 0 bytes after a block of size 128 alloc'd
==5118==    at 0x4026230: malloc (vg_replace_malloc.c:236)
==5118==    by 0x804852A: main (valgrind_test2.c:12)
==5118==
==5118== Invalid read of size 1
==5118==    at 0x804858C: main (valgrind_test2.c:25)
==5118==  Address 0x41ab028 is 0 bytes inside a block of size 128 free'd
==5118==    at 0x4025E4A: free (vg_replace_malloc.c:366)
==5118==    by 0x8048587: main (valgrind_test2.c:24)
==5118==
```

valgrind(2) 결과 출력 2

해제가 끝난 메모리 접근 –> 메모리를 **free**로 해제한 후에 그 메모리를 참조하고 있다.

free(onheap)

dummy = onheap[0];

```
==5118== Invalid read of size 1
==5118==    at 0x804858C: main (valgrind_test2.c:25)
==5118==  Address 0x41ab028 is 0 bytes inside a block of size 128 free'd
==5118==    at 0x4025E4A: free (vg_replace_malloc.c:366)
==5118==    by 0x8048587: main (valgrind_test2.c:24)
==5118==
```

복사할 원본과 대상이 중첩–> **strcpy,memcpy**등의 함수는 복사한 원본 메모리 영역과 대상 메모리 영역이 중첩되면 오동작 한다.

strcpy(onstack~)

strcpy(onstack~)

```
--5118:2:transtab                          FAST, ec = 79
==5118== Source and destination overlap in strcpy(0xbea8ff7c, 0xbea8ff7d)
==5118==    at 0x40274D0: strcpy (mc_replace_strmem.c:311)
==5118==    by 0x80485C9: main (valgrind_test2.c:28)
==5118==
--5118:1:syswrap- thread_wrapper(tid=1): exit
--5118:1:syswrap- run_a_thread_NORETURN(tid=1): post-thread_wrapper
--5118:1:syswrap- run_a_thread_NORETURN(tid=1): last one standing
```