Part 01. 리눅스 개발 환경

Chapter 03. GDB 디버깅(1)

진행 순서

Chapter 03_01 GDB 디버깅 개요
Chapter 03_02 GDB 설치
Chapter 03_03 GDB 기본 사용법
Chapter 03_04 코어 덤프
Chapter 03_05 GDB 명령
Chapter 03_06 GDB 실습1
Chapter 03_07 GDB 실습 2

Chapter 03_01 GDB 디버깅 개요

보통은 GDB라고 부르는 GNU 디버거(GNU Debugger)는 GNU 소프트웨어 시스템을 위한 기본 디버거이다. GDB는 다양한 유닉스 기반의 시스템에서 동작하는 이식성있는 디버거로, 에이다, C, C++, 포트란 등의 여러 프로그래밍 언어를 지원한다.

특징

- 프로그램을 줄 단위로 실행하거나 특정 지점에서 멈추도록 할 수 있다.
- 프로그램 수행 중간에 각각의 변수에 어떤 값이 할당되어 있는지 확인할 수 있다.
- 원하는 값을 변수에 할당 후 어떤 일이 벌어지는지 확인할 수 있다.

참고사항

프로그램 바이너리가 gdb로 디버깅 가능 하려면 컴파일 시 –g 옵션을 통하여 디버깅 정보가 추가되어야 함

ex) gcc –g hello.c –o hello

Chapter 03_01 GDB 디버깅 개요

gdb로 할 수 있는 일(상세)

정지점(Break Point) 설정 : 지정한 지정(정지점)에서 프로그램 실행을 일시적으로 중단할 수 있다.
정지점에서 프로그램을 중단한 뒤 특정 값을 표시해 이 값이 옳은지 판단할 수 있다.
정지점은 조건적으로 설정할 수도 있다. 예를들어 # (gdb) condition 1 low==high

하드웨어 감시점 : 일부 프로세서에서는 하드웨어를 사용하여 특정 메모리 값이 변하는지 감시할 수 있다.
예를들어 하위메모리 전역 변수 TheMem이 변하는 순간을 알고 싶다면

```
# (gdb) watch TheMem
Hardware watchpoint 1: TheMem
(gdb) c
Continuing.
Hardware watchpoint 1 : TheMem
Old value = 0
New value = 768
C_FirstMenu (mid=3) at menu1.c:577
```

Chapter 03_01 GDB 디버깅 개요

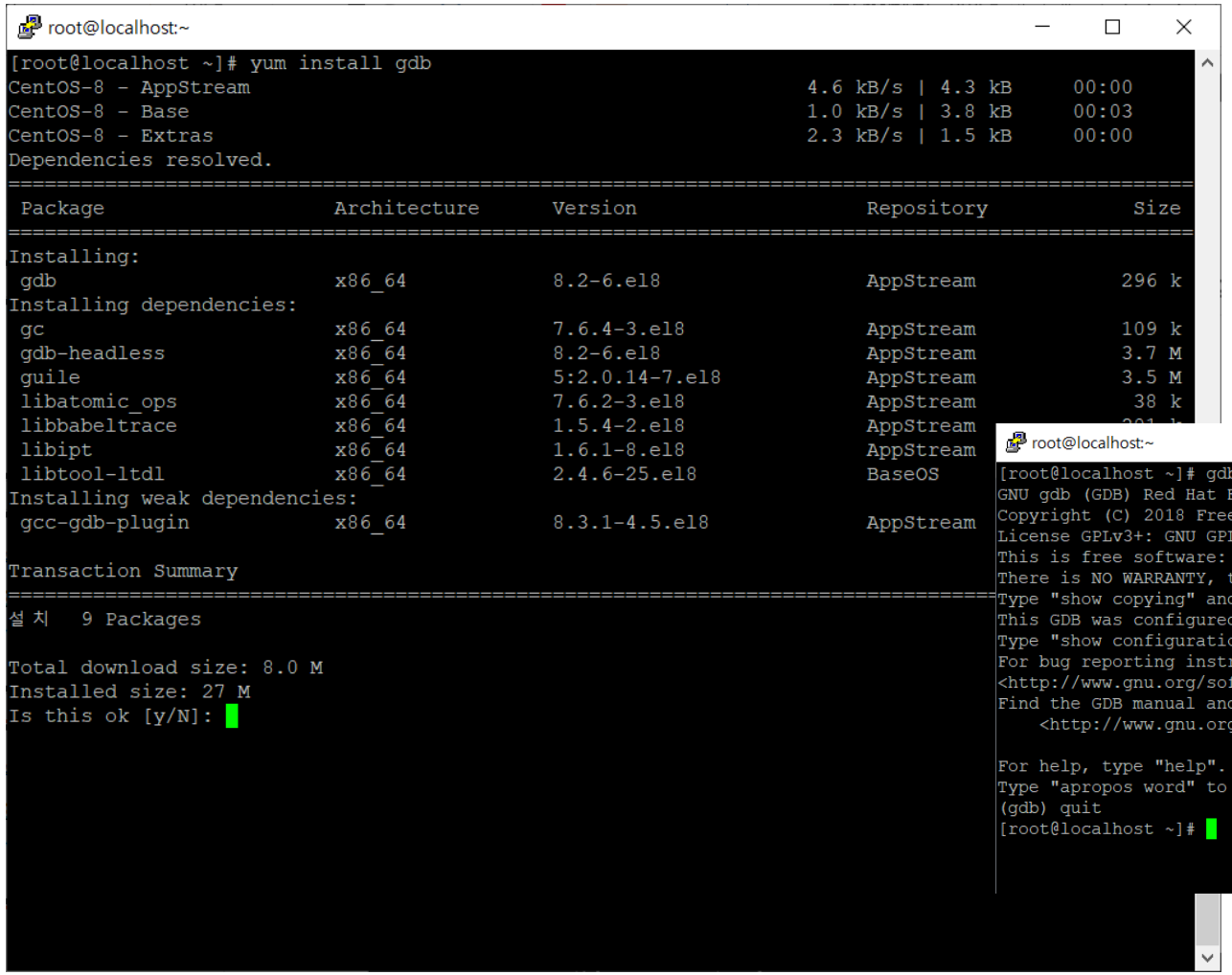
gdb로 할 수 있는 일(상세)

프로그램 값과 속성 표시 : 프로그램을 실행하면서 gdb 명령으로 현재 변수 값을 표시할 수 있다.

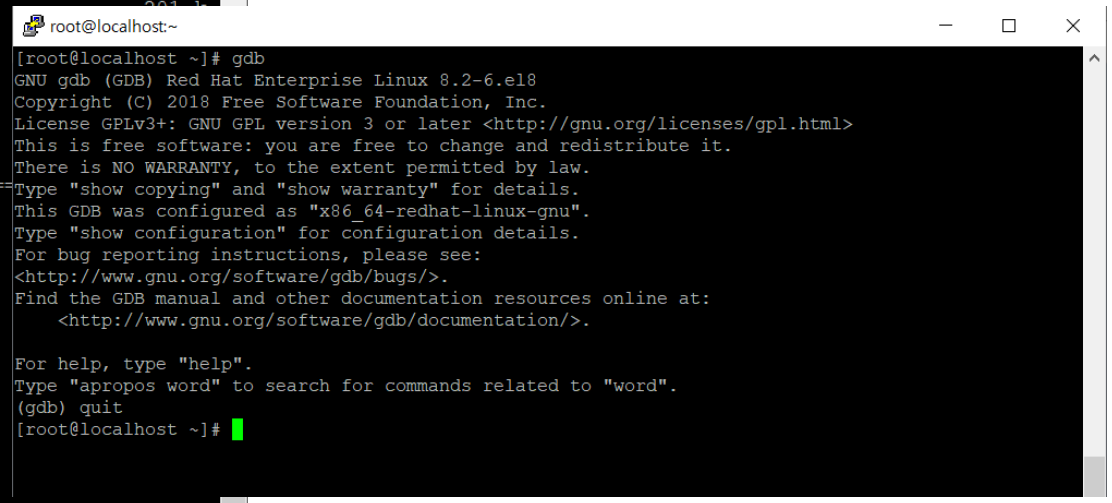
프로그램 단계별 수행 : 실행 프로그램의 각 행을 한 번에 하나씩 실행할 수 있다.

스택 프레임 : 프로그램에서 함수를 호출할 때마다, 호출 관련 정보가 스택 프레임(stack frame) 또는 프레임(frame)이라는 자료 블록에 저장된다.
함수 호출 하나당 프레임은 하나이다. 프레임에는 함수에 주어진 매개변수, 함수의 지역 변수, 함수 실행 주소 등이 들어있다. 모든 스택 프레임은 콜 스택(call stack)이라는 메모리 영역에 할당한다.
프레임을 조작하는 기본 명령에는 frame, info frame, backtrace가 있다.

Chapter 03_02 GDB 설치



yum install gdb



Chapter 03_03 GDB 기본 사용법

```
[root@localhost ~]# gdb -h
This is the GNU debugger.  Usage:

    gdb [options] [executable-file [core-file or process-id]]
    gdb [options] --args executable-file [inferior-arguments ...]
    gdb [options] [--python|-P] script-file [script-arguments ...]
    gdb [options] --args executable-file [inferior-arguments ...]
```

컴파일 시 –g 옵션을 이용하여 만들어진 프로그램(실행파일)을 인자로 gdb를 실행하면 디버깅이 시작된다.
ex) # gdb TestProgram

만약 TestProgram을 실행하다가 core dump가 발생하면 core 파일과 함께 gdb를 실행한다.
ex) # gdb TestProgram core

gdb는 현재 실행중인 프로세스에 대해서도 디버깅 작업을 수행할 수 있다.
프로세스 ID를 인자로 gdb를 수행하면 된다. 예를 들어 PID가 1234 일 경우
ex) # gdb TestProgram 1234

Chapter 03_04 코어 덤프

코어 덤프(core dump)는 컴퓨터 프로그램이 특정 시점에 작업 중이던 메모리 상태를 기록한 것으로, 보통 프로그램이 비정상적으로 종료했을 때 만들어진다. 실제로는, 그 외에 중요한 프로그램 상태도 같이 기록되곤 하는데, 프로그램 카운터, 스택 포인터 등 CPU 레지스터나, 메모리 관리 정보, 그 외 프로세서 및 운영 체제 플래그 및 정보 등이 포함된다. 코어 덤프는 프로그램 오류 진단과 디버깅에 쓰인다.
(https://en.wikipedia.org/wiki/Core_dump)

```
[root@localhost ~]# ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 3147
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 3147
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

```
[root@localhost ~]# ulimit -c unlimited
[root@localhost ~]# ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 3147
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 3147
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

사용자 응용 프로그램이 문제가 생겼을 경우 코어(core) 파일을 생성할 수 있으려면,
시스템에서 제한하는 코어 파일 크기가 0보다 커야 한다.
코어 파일 크기가 0이면, 시스템은 코어 덤프(core dump)를 허용하지 않는다.
ulimit –a 명령으로 코어 파일 크기를 확인할 수 있고,
ulimit –c 명령으로 코어 파일 크기를 변경할 수 있다.

Chapter 03_04 코어 덤프

코어 파일 생성 위치 확인
[root@localhost systemd]# cat /proc/sys/kernel/core_pattern
|/usr/lib/systemd/systemd-coredump %P %u %g %s %t %c %h %e
CentOS의 경우 상기와 같이 core_pattern의 첫 문자가 파이프(|)일 경우 systemd로 리다이렉션한다.

코어 덤프 파일 실습을 위하여 특정 위치(일반적으로 /var/crash)에 코어 덤프가 생성되도록 수정
# vi /etc/sysctl.conf
kernel.core_pattern = /var/crash/core.%e.%p.%h.%t
%e – executable filename.
%p – PID of dumped process.
%t – time of dump (seconds since 0:00h, 1 Jan 1970).
%h – hostname (same as 'nodename' returned by uname(2)).

# vi /etc/sysctl.conf
fs.suid_dumpable = 2
2(suidsafe) – 일반적으로 모든 바이너리는 루트만 읽을 수 있도록 덤프됩니다. 이를 통해 최종 사용자는 그 덤프를 제거할 수 있지만 직접 액세스 할 수 없습니다. 보안상의 이유로 이 모드의 코어 덤프는 서로 또는 다른 파일을 덮어 쓰지 않습니다. 이 모드는 관리자가 일반 환경에서 문제를 디버그하려고 할 때 적합합니다.

설정 재로딩
sysctl -p

Chapter 03_05 GDB 명령

명령	약어	설명
attach	at	실행 중인 프로세스에 디버거를 붙임
backtrace	bt	stack trace를 출력
break	b	break point를 설정
clear		break point를 해제
continue	c	프로그램 실행을 계속
delete		번호로 특정 break point를 해제
detach		현재 실행중인 프로세스에서 디버거를 제거
display		실행을 중단할 때마다 표현값을 출력
finish		함수 끝까지 실행하고 함수 반환값을 표시
help		도움말 출력
jump		특정 주소로 분기하여 실행 계속
list		소스코드 목록에서 다음 10줄 출력
next	n	프로그램 한 단계를 수행 (step과 달리 행 속에 있는 함수 안으로 들어가지 않음)
print	p	표현값을 출력
run	r	현재 프로그램을 처음부터 실행
set		변수 값을 변경
step	s	프로그램을 중단 지점 다음 행을 수행 (next와 달리 함수일 경우 함수 안으로 진입)
where	w	stack trace를 출력

Chapter 03_06 GDB 실습1

gdb_test1.c

```
#include <stdio.h>

void print_func(char *str) {
        str[0] = ' ';
        printf("String: %s\n", str);
}

int main()
{
        char *str = NULL;
        print_func(str);
        return 0;
}
```

NULL 문자열 참조

```
[root@localhost ch3]# ls
gdb_test1.c
[root@localhost ch3]# gcc -g gdb_test1.c -o gdb_test1
[root@localhost ch3]# ls
gdb_test1  gdb_test1.c
[root@localhost ch3]# ./gdb_test1
세그멘테이션 오류 (core dumped)
[root@localhost ch3]# ls /var/crash/
core.gdb_test1.2024.localhost.localdomain.1585100726
[root@localhost ch3]#
```

컴파일 시 gdb –g 옵션 주의
실행 시 세그멘테이션 오류 발생
/var/crash 디렉토리 이하에 코어 덤프 파일 발생

Chapter 03_06 GDB 실습1

```
[root@localhost ch3]# gdb ./gdb_test1
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-6.el8
...
Reading symbols from ./gdb_test1...done.
(gdb) run
Starting program: /root/FastCampus/ch3/gdb_test1
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-72.el8.x86_64

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005a6 in print_func (str=0x0) at gdb_test1.c:4
4               str[0] = ' ';
(gdb)
```

core 파일 없이 프로그램 디버깅
run을 이용하여 실행
문제 발생 시 중단되면서 정보 출력

Chapter 03_06 GDB 실습1

```
[root@localhost ch3]# gdb ./gdb_test1 /var/crash/core.gdb_test1.2024.localhost.localdomain.1585100726
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-6.el8
...
Reading symbols from ./gdb_test1...done.
[New LWP 2024]
Core was generated by `./gdb_test1'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00000000004005a6 in print_func (str=0x0) at gdb_test1.c:4
4               str[0] = ' ';
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-72.el8.x86_64
(gdb)
```

gdb 실행 시
gdb ./gdb_test1 /var/crash/core.gdb_test1.2024.localhost.localdomain.1585100726
와 같이 인자로 코어 덤프 파일을 함께 지정시
상기와 같이 Segmentation fault 발생 정보 바로 출력

Chapter 03_07 GDB 실습2

main.c

```
#include <stdio.h>
#include <stdlib.h> /* for atoi() */

void func1(void);
void func2(void);

int main()
{
        int result;
        char number[] = "abcd";

        printf("main() start\n");

        func1();
        func2();

        result = atoi(number);
        printf("result: %d\n", result);

        return 0;
}
```

func1.c

```
#include <stdio.h>

void func1()
{
        printf("func1 called\n");
}
```

func2.c

```
#include <stdio.h>

void func2()
{
        printf("func2 called\n");
}
```

원래 코드의 의도가 number 변수에 “1234” 문자열이 들어있고, atoi()를 통해 숫자 1234로 변환한 결과를 출력하고자 하는 의도라고 가정
하지만 result 출력 0

```
[root@localhost gdb_test2]# ls
func1.c  func2.c  main.c
[root@localhost gdb_test2]# gcc -g main.c func1.c func2.c -o gdb_test2
[root@localhost gdb_test2]# ./gdb_test2
main() start
func1 called
func2 called
result: 0
```

Chapter 03_07 GDB 실습2

```
[root@localhost gdb_test2]# gdb gdb_test2
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-6.el8
...
Reading symbols from gdb_test2...done.
(gdb)
```

gdb 실행

```
(gdb) break main.c:1
Breakpoint 1 at 0x40061e: file main.c, line 10.
```

main.c 파일의 첫 수행에 break point 지정
main() 함수 속 변수 선언 이후 첫 문장(10번째 줄)에 지정

```
(gdb) run
Starting program: /root/FastCampus/ch3/gdb_test2/gdb_test2
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-72.el8.x86_64

Breakpoint 1, main () at main.c:10
10              char number[] = "abcd";
```

run을 통해 프로그램 실행
Breakpoint1을 통해 main.c의 10번째 줄에서 멈췄다.

Chapter 03_07 GDB 실습2

```
(gdb) next
12              printf("main() start\n");
(gdb) next
main() start
14              func1();
(gdb) next
func1 called
15              func2();
```

next 실행
next의 경우 함수 안으로 진입하지 않고 넘어간다.

```
(gdb) step
func2 () at func2.c:5
5               printf("func2 called\n");
```

step 실행
step의 경우 함수 안으로 진입한다.
func2() 함수 내부의 결과 출력

```
(gdb) next
func2 called
6       }
(gdb) next
main () at main.c:17
17              result = atoi(number);
```

계속 next 실행
result = atoi(number); 구문을 만남

```
(gdb) display result
1: result = 0
(gdb) display number
2: number = "abcd"
```

display result와 display number를 통해
현재 시점의 result와 number 변수 값 출력
여기서 number 변수에 문자열 “1234”가 아닌
“abcd”가 들어 있다는 문제를 인식할 수 있다.

Chapter 03_07 GDB 실습2

```
(gdb) set number = "1234"
```

다음으로 넘어가기 전에 **set** 명령을 이용하여
number 변수에 제대로 된 값을 넣어볼 수 있다.
실제 코드를 수정 후 컴파일 하지 않고
수정된 값으로 변경 후 프로그램을 계속 실행

```
(gdb) continue
Continuing.
result: 1234
[Inferior 1 (process 1996) exited normally]
```

이제 **continue**를 통해 나머지 프로그램을 계속 진행해보면
result: 1234와 같이 올바른 결과가 출력됨을 확인할 수 있다.

```
(gdb) quit
[root@localhost gdb_test2]#
```

quit을 통해 **gdb** 디버깅을 종료하고 **shell**로 나갈 수 있다.

이로써 **gdb**를 통하여 코드의 문제점을 파악하고 어느 부분을 수정해야 하는지 파악이 가능하다.