Part 01. 리눅스 개발 환경

Chapter 04. GDB 디버깅(2)

진행 순서

Chapter 04_01 backtrace
Chapter 04_02 backtrace_symbols
Chapter 04_03 backtrace_symbols_fd
Chapter 04_04 backtrace 예제
Chapter 04_05 addr2line
Chapter 04_06 GDB 실전 TIP

Chapter 04_01 backtrace

어플리케이션 프로그램 동작 중 문제 발생 시 운영체제 설정 등을 통하여 코어 덤프 파일을 남길 수 있고,
이를 디버깅하여 프로그램의 문제점을 파악할 수 있습니다.
하지만 코어 덤프 파일을 남길 수 없는 환경에서 segfault 발생, SIGKILL 시그널 수신 등 문제점이 발생한 상황이거나
시스템 진단 등의 목적으로 의도적으로 backtrace를 남길 수 있다면 유용하게 활용할 수 있습니다.

```
#include <execinfo.h>

int backtrace (void **buffer, int size);
```

backtrace 함수는 포인터 목록으로 현재 스레드에 대한 backtrace를 가져와 정보를 버퍼에 넣습니다.
size는 버퍼 크기에 맞는 void * 요소의 수입니다.

리턴값은 확보된 버퍼의 실제 항목 수이며 최대 크기 입니다.

버퍼내의 포인터들은 실제로 스택을 검사하여 얻은 반환 주소이며, 스택 프래임 당 하나의 반환 주소 입니다.

Chapter 04_02 backtrace_symbols

```
#include <execinfo.h>

char **backtrace_symbols (void *const *buffer, int size);
```

backtrace_symbols 함수는 backtrace 함수에서 얻은 정보를 문자열 배열로 변환합니다.
인수 buffer는 backtrace 함수를 통해 얻은 주소 배열에 대한 포인터 여야하며 size는 해당 배열의 항목 수 입니다.

반환값은 배열 버퍼와 마찬가지로 크기 항목이 있는 문자열 배열에 대한 포인터입니다.
각 문자열에서 함수 이름, 함수에 대한 오프셋, 실제 리턴 주소(16진)가 포함됩니다.

함수 이름을 프로그램에서 사용할 수 있도록 링커에게 추가 플래그를 전달해야 합니다. (-rdynamic)

backtrace_symbols의 반환값은 malloc 함수를 통해 얻은 포인터이며 해당 포인터를 해제하는 것은 호출자의 책임입니다.
개별 문자열이 아닌 반환 값만 해제해야 합니다.

문자열에 충분한 메모리를 확보할 수 없는 경우 리턴값은 NULL 입니다.

Chapter 04_03 backtrace_symbols_fd

```
#include <execinfo.h>

void backtrace_symbols_fd (void *const *buffer, int size, int fd);
```

backtrace_symbols_fd 함수는 backtrace_symbols 함수와 동일한 변환을 수행합니다.
호출자에게 문자열을 반환하는 대신 문자열을 파일 디스크립터 fd에 한 줄에 하나씩 씁니다.
malloc 기능을 사용하지 않으므로 해당 기능이 실패할 수 있는 상황에서 사용할 수 있습니다.

Chapter 04_04 backtrace 예제

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>

/* Obtain a backtrace and print it to stdout. */
void print_trace (void)
{
    void *array[10];
    size_t size;
    char **strings;
    size_t i;

    size = backtrace (array, 10);
    strings = backtrace_symbols (array, size);

    printf ("Obtained %d stack frames.\n", size);

    for (i = 0; i < size; i++)
        printf ("%s\n", strings[i]);

    free (strings);
}
```

```
/* A dummy function to make the backtrace more interesting. */
void dummy_function (void)
{
    print_trace ();
}

int main (void)
{
    dummy_function ();
    return 0;
}
```

```
[root@localhost ch4]# gcc -g backtrace_example.c -o backtrace_example –rdynamic
[root@localhost ch4]# ./backtrace_example
Obtained 5 stack frames.
./backtrace_example(print_trace+0x19) [0x4008bf]
./backtrace_example(dummy_function+0x9) [0x400942]
./backtrace_example(main+0x9) [0x40094e]
/lib64/libc.so.6(__libc_start_main+0xf3) [0x7f7f14ba1873]
./backtrace_example(_start+0x2e) [0x4007ee]
```

Chapter 04_05 addr2line

addr2line 명령은 주소를 파일 이름과 줄 번호로 변환하는데 사용됩니다.
addr2line 명령을 사용하여 기계어 명령(machine instruction) 주소를 명령이 시작된 파일의 행에 매핑할 수 있습니다.

addr2line [options] [addr addr ...]

예를 들어 앞 부분에서 설명한 backtrace_example 실행의 결과가 아래와 같을 때

```
[root@localhost ch4]# ./backtrace_example
Obtained 5 stack frames.
./backtrace_example(print_trace+0x19) [0x4008bf]
./backtrace_example(dummy_function+0x9) [0x400942]
./backtrace_example(main+0x9) [0x40094e]
/lib64/libc.so.6(__libc_start_main+0xf3) [0x7f7f14ba1873]
./backtrace_example(_start+0x2e) [0x4007ee]
```

실제로 문제가 생긴 부분이 dummy_function() 함수에 의해서라고 가정하면

```
[root@localhost ch4]# addr2line -e backtrace_example 0x400942
/root/FastCampus/ch4/backtrace_example.c:30
```

print_trace() 함수라고 가정하면

```
[root@localhost ch4]# addr2line -e backtrace_example 0x4008bf
/root/FastCampus/ch4/backtrace_example.c:14
```

와 같이 쉽게 코드의 행을 매칭하여 디버깅할 수 있다.

Chapter 04_06 GDB 실전 TIP

문제

회사에서 리눅스 플랫폼에서 돌아가는 애플리케이션을 개발 중이다.
그런데 코드에 문제가 있는지 실행만하면 프로세스가 죽어버린다.
화면에 SegFault 메시지가 출력되도록 할 수 없을까?
화면에 SegFalut 메시지는 출력되는데,
도대체 이 많은 분량의 코드 중 어디서부터 문제인지 찾을 수 있을까?

해결책1

흐름 절차에 따라 예상되는 지점에 printf를 찍고,
화면에 찍히는 디버깅 출력을 분석한다.

해결책2

SegFault 메시지를 활용하여 GDB 디버깅을 한다.

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void segv_handler(int sig) {
        int j, nptrs;
#define SIZE 100
        void *buffer[100];
        char **strings;

        nptrs = backtrace(buffer, SIZE);
        printf("backtrace() returned %d addresses\n", nptrs);

        fprintf(stderr, "ERROR: signal(%d)\n", sig);
        backtrace_symbols_fd(buffer, nptrs, STDERR_FILENO);
        exit(1);
}

void foo() {
        printf("foo\n");
}

void bar() {
        printf("bar\n");
}

void first_to_space(char *str) {
        printf("func start\n");
        str[0] = ' ';
        printf("func end\n");
}
```

```
int main(int argc, char **argv)
{
        char *str = NULL;

        signal(SIGSEGV, segv_handler);

        foo();
        bar();
        first_to_space(str);

        return 0;
}
```

```
[root@localhost ch3]# gcc -g -rdynamic test.c -o test
[root@localhost ch3]# ./test
foo
bar
func start
backtrace() returned 6 addresses
ERROR: signal(11)
./test(segv_handler+0x25)[0x4009db]
/lib64/libc.so.6(+0x37a20)[0x7f422aaeba20]
./test(first_to_space+0x1a)[0x400a70]
./test(main+0x46)[0x400ac6]
/lib64/libc.so.6(__libc_start_main+0xf3)[0x7f422aad7873]
./test(_start+0x2e)[0x4008fe]
```

man 3 backtrace 참조
주의사항:
backtrace 관련 내요 추가 후 컴파일 시 –rdynamic 링커 필요

```
[root@localhost ch3]# gdb test
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-6.el8
…
Reading symbols from test...done.
(gdb) list *(first_to_space+0x1a)
0x400a70 is in first_to_space (test.c:31).
26              printf("bar\n");
27      }
28
29      void first_to_space(char *str) {
30              printf("func start\n");
31              str[0] = ' ';
32              printf("func end\n");
33      }
34
35      int main(int argc, char **argv)
(gdb)
```

gdb list 명령은 소스코드의 목록에서 10줄 출력해주는 명령
segfault backtrace에서 segv_handle가 호출된 마지만 trace는
first_to_space+0x1a 임을 확인할 수 있다.
해당 위치의 코드를 아래와 같이 따라가면
segfault 문제가 발생한 소스 코드 위치를 한 번에 찾을 수 있다.
(gdb) list *(first_to_space+0x1a)