[KISTI 기술문서] ISBN 978-89-294-1694-2

리눅스 프로세스 디버깅 및 성능 분석도구 기초

2024. 6. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24-L02-C06-S01
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

■ 리눅스(Linux)는 지난 수십년간 놀라운 발전을 이루어 왔으며, 현대의 컴퓨팅 환경에 큰 영향을 미치고 있다. 1991년 Linus Torvalds가 리눅스 커널의 초기 버전을 발표한 이후, 오픈소스 프로젝트로 성장을 이루었고, Debian, RedHat, Slackware와 같은 배포판이 등장하여 서버 환경에서 주로 사용되었다. 리눅스 기반의 다양한 오픈 소스 프로젝트와 더불어 GNOME, KDE와 같은 데스크톱 환경이 발전하면서 사용자 친화성과 활용성을 넓혀가고 있다. 2008년 Google이 출시한 Android OS는 리눅스 커널을 기반으로한 모바일 운영체제이며, 스마트TV, 자동차, IoT 등 다양한 임베디드 시스템에서 널리 사용되고 있다.

■ 본 문서에서는 리눅스 또는 일반적인 운영체제 시스템에 대한 이해를 돕기위해 몇 가지 개발 및 분석 도구에 대하여 예제 코드를 중심으로 설명한다.

■ 리눅스 시스템 프로그래밍을 공부하는 이유는 다음과 같다:
- 운영 체제가 어떻게 동작하고 이해할 수 있다: 프로그램이 커널과 어떻게 상호작용하는지, 프로세스나 메모리는 어떻게 관리되는지, 파일 시스템의 동작원리 등을 이해할 수 있음
- 개발자의 역량 향상: 시스템 프로그램 개발 일반적인 프로그램 개발보다 보다 저수준에서 지식을 요구한다. 이를 위해 개발자는 코드의 작성, 성능 최적화, 리소스 관리 등 고급 기술을 습득 할 수 있다.
- 보안 수준 향상: 보안과 관련된 취약점을 발견하거나 해결하는 과정에서 저수준 시스템 이해는 필수적이다.

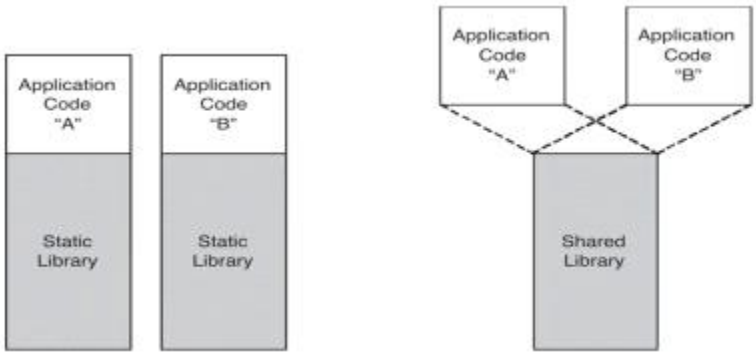
■ 리눅스 시스템 프로그램을 공부하기 위해서는 다음과 같은 지식이 필요하거나 도움이 된다:
- 운영체제의 기본적인 원리: 프로세스 관리, 메모리 관리, 파일 시스템, 입출력 디바이스에 관한 기초적인 지식
- 프로그래밍 언어: C언어, 쉘 스크립트의 기본적인 사용법
- 시스템 호출 및 라이브러리: 시스템 콜의 동작 원리와 간단한 사용법, glibc와 같은 표준 라이브러리에 대한 지식
- 디버깅 도구: gdb, strace, gprof등의 도구를 이용한 디버깅 및 성능 분석 도구

■ 본 문서에서 작성된 코드는 우분투 리눅스 20.04 64비트 운영체제에서 테스트하였음

2. 동적 링크 라이브러리

2.1 정적 링크와 동적 링크

- 정적 링킹(static linking)와 동적 링킹(dynamic linking)의 차이는 실행 가능한 목적 파일을 만들때 프로그램이 사용하는 모듈을 복사하는 방식의 차이를 말함
- 그림1 참고. 출처[1)]



[그림 1] 정적 링크 방식과 동적 링크 방식의 차이

기준	정적 링크	동적 링크
라이브러리 참조	- 컴파일 타임에 필요한 라이브러리를 실행 파일에 포함. 컴파일 타임에 링커에 의해 링크	- 프로그램이 실행할 때 필요한 라이브러리를 런타임시 OS에 의해로딩
라이브러리 위치	- 컴파일된 실행파일은 필요한 라이브러리 코드를 포함하고 있음	- 실행 파일은 필요한 라이브러리에 대한 정보만을 가지고 있음, 라이브러리는 별도의 파일에 저장됨
실행파일 크기	- 실행파일의 크기가 비교적 큼	- 실행파일의 크기가 비교적 작음
메모리 로딩	- 실행 파일의 크기가 크기 때문에 처음 실행시 로딩 타임이 길다	- 런타임에 라이브러리를 로딩하는데 시간이 걸림 - 동일한 라이브러리는 메모리에서 공유되므로 메모리 사용량을 줄어듦
독립성	- 이식성이 높고, 실행파일은 독립적임	- 라이브러리가 업데이트 되면 실행파일을 다시 컴파일하지 않아도 된다. - 라이브러리 의존성이 깨어질 경우 프로그램 실행에 영향을 받음

[표 1] 정적 링크 방식과 동적 링크 방식의 비교

2.2 정적 링크와 동적 링크 예제

- 테스트를 위한 다음과 같은 코드를 작성함.

```
// foo.h
extern void foo(void);
```

```
// foo.c
#include <stdio.h>
void foo(void) {
   puts("Hello, Foo");
}
```

1) https://medium.com/@dkwok94/the-linking-process-exposed-static-vs-dynamic-libraries-977e92139b5f

- 정적 라이브러리 생성. (ar은 아카이브 파일(.a)을 관리하는 명령임)

```
$ gcc -o foo.o -c foo.c
$ ar -crs libfoo.a  foo.o
$ ar -t libfoo.a
foo.o
$ file libfoo.a
libfoo.a: current ar archive
```

- 위 라이브러리를 사용하는 프로그램을 다음과 같이 작성 후 컴파일

```
// main.c
#include <stdio.h>
#include "foo.h"
int main(void) {
    puts("calling foo..");
    foo();
    return 0;
}
$ gcc -g  -o main.o  -c main.c
$ gcc     -o main_nodebug.o  -c main.c
```

※ 컴파일시 -g 옵션을 사용하면 디버깅 정보를 포함한다. objdump -g 명령을 이용하여 디버깅 정보를 확인함.

```
$ objdump -g main.o
main.o:     file format elf64-x86-64

Contents of the .debug_info section (loaded from main.o):

  Compilation Unit @ offset 0x0:
   Length:        0x2fd (32-bit)
   Version:       4
   Abbrev Offset: 0x0
   Pointer Size:  8
[...]
Contents of the .debug_str section (loaded from main.o):
[...]
```

- 정적링크 방식을 사용하려면 -static 옵션을 사용한다. 이때 참조할 라이브러리(-l)와 검색 경로(-L) 옵션이 필요함

```
$ gcc -static -o main_static  main.o  -lfoo -L .  -Wl,-Map=main_static.map
$ file  main_static
main_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, Build
ID[sha1]=70d867c046f1587867f671d11c3e61d2e760bf28, for GNU/Linux 3.2.0, with debug_info, not st
ripped
$ ./main_static
calling foo..
Hello, Foo
```

- 동적 링킹을 위해서는 동적 라이브러리 파일(.so shared object)을 생성해야 한다. 이를 위해 -shared 옵션을 이용한다.

```
$ gcc -c -Wall -Werror -fpic foo.c
$ gcc -shared -o libfoo.so foo.o
$ file libfoo.so
libfoo.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, BuildID
[sha1]=7542ef388b4bdfeb573dc582cae202c633293976, not stripped
```

※ -fpic 옵션은 position-independent code를 생성한다. -fpic 와 -fPIC 옵션중 -fPIC 옵션은 코드 사이즈가 증가, -fpic 는 작고 빠른 코드를 생성함[2)3)]

※ readelf 명령을 이용하여 오브젝트 파일과 동적 오브젝트파일을 비교하면 다음과 같이 타입이 다른 것을 알수 있음

```
$ readelf -h foo.o | egrep "Type|Entry|program"
  Type:                              REL (Relocatable file)
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
$ readelf -h libfoo.so | egrep "Type|Entry|program"
  Type:                              DYN (Shared object file)
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
```

- 라이브러리를 링크하여 실행파일 생성 (-L 옵션을 이용하여 경로를 지정해 주어야 함)

```
$ gcc -o main_shared  main.o -lfoo
/usr/bin/ld: cannot find -lfoo
collect2: error: ld returned 1 exit status

$ gcc -o main_shared  main.o -lfoo -L . -g  -Wl,-Map=main_shared.map
$ file main_shared
main_shared: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interp
reter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=38e1b9ee3fd7b591a2f96455fffd4bb8457564dd, for
GNU/Linux 3.2.0, with debug_info, not stripped
```

- 실행결과 (LD_LIBRARY_PATH 를 지정하여 런타임시 찾을 수 있도록 한다)

```
$ ./main_shared
./main_shared: error while loading shared libraries: libfoo.so: cannot open shared object file:
No such file or directory
$ export LD_LIBRARY_PATH=. ;  ./main_shared
calling foo..
Hello, Foo
```

※ /etc/ld.so.conf 에 경로를 넣어주어도 됨. ld.so.conf.d 폴더에 파일을 생성. 현재폴더(.)를 설정해 준다.

2) https://stackoverflow.com/questions/3544035/what-is-the-difference-between-fpic-and-fpic-gcc-parameters
3) https://devanix.tistory.com/198

```
$ sudo vi /etc/ld.so.conf.d/foo.conf
.
:wq
$ sudo ldconfig
$ ./main_shared    # LD_LIBRARY_PATH 없이 실행가능
```

- 실행파일의 크기를 비교해 보면 동적 링크한 파일이 더 작음을 알 수 있다.

```
$ du -h main_static main_shared
852K    main_static
20K     main_shared
```

- ldd 명령을 이용하여 공유 라이브러리를 출력해 보면 다음과 같다. ldd명령은 해당 명령을 실행하기 위해서 필요한(의존적인) shared object들을 출력한다. 상세출력 옵션(-v)을 사용할 수 있다. 실행파일은 런타임 시에 libfoo.so라는 동적 라이브러리가 필요함을 알 수 있다.

```
$ ldd main_static
    not a dynamic executable
$ ldd -v main_shared
    linux-vdso.so.1 (0x00007ffc0b8f5000)
    libfoo.so => ./libfoo.so (0x00007f25b0fc8000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f25b0dd6000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f25b0fdd000)

    Version information:
    ./main_shared:
        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
    ./libfoo.so:
        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
    /lib/x86_64-linux-gnu/libc.so.6:
        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
```

※ linux-vdso.so 는 리눅스 커널내부에 존재하며 파일 형태로 존재하지 않는다. vitual shared object [4]

※ 위의 ldd 명령에서 해당 오브젝트가 로딩되는 주소가 함께 표시된다. ASLR(address space layout randomization) 기능이 활성화 되어 있다면, 이 주소는 계속해서 변경된다. ASLR을 해제하는 방법은 root권한으로 명령 [printf 0 > /proc/sys/kernel/randomize_va_space]을 실행하면 된다.

※ ELF statifier[5] 라는 툴을 사용하면 동적 바이너리를 정적 바이너리로 변환할 수 있다.

- rpath 옵션은 런타임(runtime) 라이브러리 검색 경로를 추가한다. 런타임에 공유라이브러리 객체를 검색하기 위해 사용된다. (-Wl 은 링커 옵션으로 -Wl,aaa,bbb,ccc 와 같이 콤마로 구분하며 ld aaa bbb ccc 와 같이 링커 실행시 아규먼트로 전달된다)

```
$ unset LD_LIBRARY_PATH
$ gcc -Wl,-rpath=/home/ubuntu/lib  -o main_shared2  -save-temps  main.c -lfoo -L .
$ readelf -d main_shared2 | grep -i runpath
 0x000000000000001d (RUNPATH)            Library runpath: [/home/ubuntu/lib]
```

4) https://man7.org/linux/man-pages/man7/vdso.7.html
5) http://statifier.sourceforge.net/

※ -save-temps 옵션: 컴파일 중간 과정으로 생기는 전처리 파일(.i), 어셈블리 파일(.s) 등을 삭제하지 않는다

2.3 링크 방식 차이점 분석

- nm 유틸리티는 오브젝트 파일 내부의 심볼에 관한 정보를 출력한다. shared 보다는 static 경우 포함된 심볼이 훨씬 많음을 알수 있다.

```
$ nm main_static  ¦ wc -l
1717
$ nm main_shared  ¦ wc -l
34
$ nm main_shared ¦ egrep " foo¦ main¦ puts"
                 U foo
                 U __libc_start_main@@GLIBC_2.2.5
0000000000001169 T main
                 U puts@@GLIBC_2.2.5
$ nm main_static ¦ egrep " foo¦ main¦ puts"
0000000000401d25 T foo
0000000000401d05 T main
00000000004c07a0 d main_arena
0000000000411680 W puts
```

- 첫번째 컬럼은 심볼의 값(주소)을 의미하고, 두번째 컬럼은 심볼타입으로 의미는 다음과 같다. 세번째 컬럼은 심볼의 이름이다.

심볼 타입	설명
A	absolute, and will not be changed by further linking
B, b	BSS data section
C	Common, uninitialized data
D, d	initialized data section
G, g	initialized data section for small objects
i	indirect function
I	indirect reference to another symbol
N	debugging symbol
n	read-only data section
p	stack unwind section
R, r	read-only data section
S, s	uninitialized or zero-initialized data section for small objects
T, t	the text (code) section
U	undefined
u	unique global symbol
V, v	weak object
W, w	weak symbol that has not been specifically tagged as a weak object symbol
-	stabs symbol in an a.out object file
?	unknown type symbol

[표 2] 심볼 타입의 의미

- 심볼 테이블은 objdump -t 명령을 이용하여 출력할 수도 있다. shared 바이너리에서 puts과 foo는 동적심볼이므로 *UND*(undefined)로 나타난다. 동적 심볼의 경우는 -T¦--dynamic-syms 옵

션을 이용하면 출력된다.

```
$ objdump -t main_shared ¦ egrep "main¦foo¦puts"
[...]
0000000000000000       F *UND*   0000000000000000              puts@@GLIBC_2.2.5
[...]
0000000000000000       F *UND*   0000000000000000              foo
0000000000001169 g     F .text   0000000000000020              main

$ objdump -T main_shared
main_shared:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000  w   D  *UND*   0000000000000000              _ITM_deregisterTMCloneTable
0000000000000000      DF *UND*   0000000000000000  GLIBC_2.2.5 puts
0000000000000000      DF *UND*   0000000000000000  GLIBC_2.2.5 __libc_start_main
0000000000000000  w   D  *UND*   0000000000000000              __gmon_start__
0000000000000000      DF *UND*   0000000000000000              foo
0000000000000000  w   D  *UND*   0000000000000000              _ITM_registerTMCloneTable
0000000000000000  w   DF *UND*   0000000000000000  GLIBC_2.2.5 __cxa_finalize
```

※ __gmon_start__ 라는 심벌은 프로그램의 성능 분석을 위해 사용되는 함수로 gprof [6]와 같은 프로파일링 도구에서 활용된다. 프로그램이 실행될때 함수가 호출되어 프로그램 실행 중 함수 호출 정보를 수집하기 위한 초기화를 수행한다.

- gdb를 이용하여 정적 바이너리와 동적 바이너리를 각각 분석한다.

```
$ gdb  main_static
(gdb) disas  main
Dump of assembler code for function main:
   0x0000000000401d05 <+0>:   endbr64
   0x0000000000401d09 <+4>:   push   %rbp
   0x0000000000401d0a <+5>:   mov    %rsp,%rbp
   0x0000000000401d0d <+8>:   lea    0x932f0(%rip),%rdi        # 0x495004
   0x0000000000401d14 <+15>:  callq  0x411680 <puts>
   0x0000000000401d19 <+20>:  callq  0x401d25 <foo>
   0x0000000000401d1e <+25>:  mov    $0x0,%eax
   0x0000000000401d23 <+30>:  pop    %rbp
   0x0000000000401d24 <+31>:  retq
End of assembler dump.
(gdb) disas  foo
Dump of assembler code for function foo:
   0x0000000000401d25 <+0>:   endbr64
   0x0000000000401d29 <+4>:   push   %rbp
   0x0000000000401d2a <+5>:   mov    %rsp,%rbp
   0x0000000000401d2d <+8>:   lea    0x932de(%rip),%rdi        # 0x495012
   0x0000000000401d34 <+15>:  callq  0x411680 <puts>
   0x0000000000401d39 <+20>:  nop
   0x0000000000401d3a <+21>:  pop    %rbp
   0x0000000000401d3b <+22>:  retq
End of assembler dump.
$ gdb  main_shared
```

6) https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
[...]
Reading symbols from main_shared...
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001209 <+0>:   endbr64
   0x000000000000120d <+4>:   push   %rbp
   0x000000000000120e <+5>:   mov    %rsp,%rbp
   0x0000000000001211 <+8>:   lea    0xdec(%rip),%rdi        # 0x2004
   0x0000000000001218 <+15>:  callq  0x1080 <puts@plt>
   0x000000000000121d <+20>:  callq  0x10a0 <foo@plt>
   0x0000000000001222 <+25>:  mov    $0x0,%eax
   0x0000000000001227 <+30>:  pop    %rbp
   0x0000000000001228 <+31>:  retq
End of assembler dump.
(gdb) disas foo
Dump of assembler code for function foo@plt: ;  PLT 엔트리 호출
   0x00000000000010a0 <+0>:   endbr64
   0x00000000000010a4 <+4>:   bnd jmpq *0x2f1d(%rip)        # 0x3fc8 <foo@got.plt>
   0x00000000000010ab <+11>:  nopl   0x0(%rax,%rax,1)
End of assembler dump.
```

- 동적 실행파일의 경우 foo의 내용을 보면 실제적으로는 0x3c8 번지(0x10ab + 0x2f1d)인 <foo@got.plt>를 참조하는 것을 알 수 있다. 해당 메모리 번지에는 0x1050 값이 저장되어 있으므로 이 주소로 점프(분기)됨을 알 수 있다.

```
(gdb) x/2x 0x3fc8
0x3fc8 <foo@got.plt>: 0x1050  0x0000
(gdb) x/4i 0x1050
   0x1050: endbr64
   0x1054: pushq  $0x2
   0x1059: bnd jmpq 0x1020
   0x105f: nop
```

- foo 내부에서 puts()를 호출하므로, 이 부분도 <puts@got.plt>를 참고하여 해당 주소로 점프한다.

```
(gdb) x/h 0x3fc8
0x3fc8 <puts@got.plt>:    0x1030
(gdb) x/32i 0x1030
   0x1030: endbr64
   0x1034: pushq  $0x0
   0x1039: bnd jmpq 0x1020
   0x103f: nop
[...]
```

- PLT(Procedure Linkage Table) 는 동적 링킹에서 사용되는 구조로 프로그램이 다른 라이브러리나 모듈의 함수를 호출할때 사용됨. 다음과 같은 순서로 동작한다.
  함수 호출 → PLT 엔트리 호출 → PLT 엔트리 실행 → GOT 검색 → 실제 함수 호출
- GOT(Global Offset Table)는 전역 심볼의 주소를 저장하고 있으며 지연된 바인딩(lazy binding)을 가능하게 한다.

- 디버거를 이용하여 실행하면서 플로우를 다음과 같이 따라가 본다.

```
(gdb) b main        ; main 시작에서 break를 설정하고 실행을 시작함
Breakpoint 1 at 0x1209: file main.c, line 4.
(gdb) run
Starting program: /home/ubuntu/lib/main_shared

Breakpoint 1, main () at main.c:4
4   int main(void) {
(gdb) x/10i $pc      ; 현재 위치에서 코드를 출력
=> 0x555555555209 <main>: endbr64
   0x55555555520d <main+4>:   push   %rbp
   0x55555555520e <main+5>:   mov    %rsp,%rbp
   0x555555555211 <main+8>:   lea    0xdec(%rip),%rdi        # 0x555555556004
   0x555555555218 <main+15>:  callq  0x555555555080 <puts@plt>
   0x55555555521d <main+20>:  callq  0x5555555550a0 <foo@plt>
   0x555555555222 <main+25>:  mov    $0x0,%eax
   0x555555555227 <main+30>:  pop    %rbp
   0x555555555228 <main+31>:  retq
   0x555555555229:    nopl   0x0(%rax)
(gdb) b *0x55555555521d    ; foo 호출하는 부분에서 break를 걸고 실행
Breakpoint 2 at 0x55555555521d: file main.c, line 6.
(gdb) continue
Continuing.
calling foo..
Breakpoint 2, main () at main.c:6
6       foo();
(gdb) si     ; stepi 하여 foo 안으로 들어감
0x00005555555550a0 in foo@plt ()
(gdb) x/10i $pc   ; 현재 위치에서 코드를 출력 (여기는 PLT 엔트리 이므로 GOT를 이동)
=> 0x5555555550a0 <foo@plt>:  endbr64
   0x5555555550a4 <foo@plt+4>:    bnd jmpq *0x2f1d(%rip)        # 0x555555557fc8 <foo@got.plt>
   0x5555555550ab <foo@plt+11>:   nopl   0x0(%rax,%rax,1)
[...]
(gdb) si
0x00005555555550a4 in foo@plt ()
(gdb) si   ; 실제 foo 함수 안으로 들어옴
0x00007ffff7fc3119 in foo () from ./libfoo.so
(gdb) x/10i $pc  ; 현재 위치에서 코드를 출력, 문자열이 저장된 주소번지를 이용하여 puts호출
=> 0x7ffff7fc3119 <foo>:  endbr64
   0x7ffff7fc311d <foo+4>:    push   %rbp
   0x7ffff7fc311e <foo+5>:    mov    %rsp,%rbp
   0x7ffff7fc3121 <foo+8>:    lea    0xed8(%rip),%rdi        # 0x7ffff7fc4000
   0x7ffff7fc3128 <foo+15>:   callq  0x7ffff7fc3050 <puts@plt>
   0x7ffff7fc312d <foo+20>:   nop
   0x7ffff7fc312e <foo+21>:   pop    %rbp
   0x7ffff7fc312f <foo+22>:   retq
   0x7ffff7fc3130 <_fini>:    endbr64
   0x7ffff7fc3134 <_fini+4>:  sub    $0x8,%rsp
(gdb) x/2s 0x7ffff7fc4000    ; 문자열이 저장된 번지의 내용을 출력
0x7ffff7fc4000:   "Hello, Foo"
0x7ffff7fc400b:   ""
```

※ objdump 명령에 -d(disassemble) 옵션을 사용하면 코드의 역어셈블 결과를 출력할 수 있음

```
$ objdump -d  --visualize-jumps  main_shared
[...]
0000000000001209 <main>:
    1209:  f3 0f 1e fa           endbr64
    120d:  55                    push   %rbp
    120e:  48 89 e5              mov    %rsp,%rbp
    1211:  48 8d 3d ec 0d 00 00  lea    0xdec(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    1218:  e8 63 fe ff ff        callq  1080 <puts@plt>
    121d:  e8 7e fe ff ff        callq  10a0 <foo@plt>
    1222:  b8 00 00 00 00        mov    $0x0,%eax
    1227:  5d                    pop    %rbp
    1228:  c3                    retq
    1229:  0f 1f 80 00 00 00 00  nopl   0x0(%rax)
[...]
```

- objdump -h 는 ELF파일의 섹션 헤더 정보를 출력한다.
  ※ size -Ax 명령도 비슷한 기능을 함

```
$ objdump -h main_static
main_static:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .note.gnu.property 00000020  0000000000400270  0000000000400270  00000270  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.build-id 00000024  0000000000400290  0000000000400290  00000290  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.ABI-tag 00000020  00000000004002b4  00000000004002b4  000002b4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .rela.plt     00000240  00000000004002d8  00000000004002d8  000002d8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
[...]
$ objdump -h main_shared
main_shared:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

※ VMA(virtual memory address)는 실행시 해당 섹션이 가상메모리 공간에 위치하는 주소이다. 다음으로 LMA(load memory address)는 메모리에 로드되는 주소이다. 대부분의 경우에 두 주소는 같다. 다를 수 있는 예는 데이터 섹션이 ROM에 로드되고 실행될때 RAM으로 복사되는 경우이다. 자세한 것은 ld 명령의 메뉴얼을 참고.

※ 아래 표는 objdump -h 명령의 결과 나온 섹션명과 크기를 비교한 것이다. (일부는 생략)

정적(static) 링크 바이너리		동적(shared) 링크 바이너리	
SECTION	SIZE	SECTION	SIZE
.bss	00001718	.bss	00000008
.comment	0000002b	.comment	0000002b
.data	00001a50	.data	00000010
.data.rel.ro	00002df4		
.debug_abbrev	000000ce	.debug_abbrev	000000ce
.debug_aranges	00000030	.debug_aranges	00000030
.debug_info	00000301	.debug_info	00000301
.debug_line	00000113	.debug_line	00000113
.debug_str	00000291	.debug_str	00000291
		.dynamic	00000200
		.dynstr	000000b4
		.dynsym	00000108
.eh_frame	0000a624	.eh_frame	00000148
		.eh_frame_hdr	0000005c
.gcc_except_table	000000c4		
.got	000000f0	.got	00000060
.got.plt	000000d8		
.note.ABI-tag	00000020	.note.ABI-tag	00000020
.note.gnu.build-id	00000024	.note.gnu.build-id	00000024
.note.gnu.property	00000020	.note.gnu.property	00000020
.note.stapsdt	000013e8		
.plt	00000180	.plt	00000050
		.plt.got	00000010
		.plt.sec	00000040
		.rela.dyn	000000c0
.rela.plt	00000240	.rela.plt	00000060
.rodata	0001c00c	.rodata	00000012
.stapsdt.base	00000001		
.tbss	00000040		
.tdata	00000020		
.text	00091ac0	.text	00000202

※ 아래 명령을 이용하면 특정 섹션만 역어셈블함 결과를 확인 가능함.

```
$ objdump -d --section=.plt   main_shared
0000000000001020 <.plt>:
    1020:  ff 35 82 2f 00 00     pushq  0x2f82(%rip)        # 3fa8 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:  f2 ff 25 83 2f 00 00  bnd jmpq *0x2f83(%rip)        # 3fb0 <_GLOBAL_OFFSET_TABLE_+0x
10>
    102d:  0f 1f 00              nopl   (%rax)
    1030:  f3 0f 1e fa           endbr64
    1034:  68 00 00 00 00        pushq  $0x0
    1039:  f2 e9 e1 ff ff ff     bnd jmpq 1020 <.plt>
    103f:  90                    nop
[...].
```

2.4 동적 로딩

- 동적 적재(dynamic loading, 동적 로딩)[7] 라이브러리는 OS에 의한 프로그램 시작이 아닌 런타임에 메모리에 적재되는 코드를 말함. 모든 것을 미리 로딩하지 않고, 필요할 때만 로딩을 하는 기법. 실행중에 필요한 시점에 필요한 것만 로드함으로써 효율성을 높일 수 있다. 메모리 효율성, 빠른 초기 실행, 유연성 측면에서 장점이 존재함.
- IBM System/360 시스템의 OS/360 운영체제에 동적 로딩 기술이 사용됨[8]
- C언어에서 동적 로딩을 위한 기능은 `dlfcn.h` 에 정의됨.

함수명	기능
`dlopen`	- 형식: `void *dlopen(const char *file, int mode);` - file에 지정된 실행 가능 개체 파일을 호출 프로그램에서 사용할 수 있도록 만듦. mode는 동작을 제어하는 플래그로 `RTLD_LAZY`(심볼이 사용될 때 로드) 또는 `RTLD_NOW`(즉시 모든 심볼 로드)를 사용 - 실패시 `NULL`을 반환함
`dlsym`	- 형식: `void *dlsym(void *, const char *);` - 로드된 공유 라이브러리에서 심볼(함수나 변수)를 가져옴
`dlclose`	- 형식: `int dlclose(void *);` - 로드된 공유 라이브러리를 언로드 함 - 성공시 `0`을 반환, 실패시 `0`이 아닌 값 반환
`dlerror`	- 형식: `char *dlerror(void);` - 관련 오류가 발생했을때 오류에 대한 설명을 반환

- 다음 예제를 통해 알아보자. `dlopen`으로 `libm.so.6` 라는 라이브러리 파일을 열어 cos라는 함수의 심볼을 확인한후 호출한다. `cos(π)(=-1)`의 결과를 출력한다.

```
// dlmain.c
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
#include <unistd.h>
int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);  exit(1);
    }
    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr); exit(1);
    }
    printf ("%f\n", (*cosine)(3.14));   // cos(PI) ==> -1
    sleep(60);
    dlclose(handle);
```

7) https://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html
8) https://en.wikipedia.org/wiki/OS/360_and_successors

```
}
$ gcc -o dlmain  dlmain.c -ldl ; ./dlmain
-0.999999
```

- main 프로그램을 static 으로 링크 할 수 있지만 다음과 같이 경고가 출력됨

```
$ gcc  -static  -o dlmain dlmain.c -ldl
/usr/bin/ld: /tmp/ccRpvPbo.o: in function `main':
dlmain.c:(.text+0x21): warning: Using 'dlopen' in statically linked applications requires at ru
ntime the shared libraries from the glibc version used for linking
```

- cos라는 함수를 라이브러로 만들어 main함수를 재 컴파일하지 않고, 실행해 본다. 다음 코드를 작성하고, 라이브러리로 만들어 준다. LD_LIBRARY_PATH를 사용하여 현재 경로의 모듈을 로딩하게 함으로써, 직접 작성한 cos함수가 호출된다.

```
// cosine.c
double cos(double a) {
  return 99.99;
}
$ gcc -c -Wall -Werror -fpic cosine.c
$ gcc -shared -o libm.so.6  cosine.o
$ LD_LIBRARY_PATH=.   ./dlmain
99.990000
```

- 실행중인 프로세서의 메모리 맵은 /proc/<pid>/maps 에서 확인 가능하다. (프로세스의 메모리맵 정보는 gdb에서 info proc mappings 명령으로도 가능함)

```
$ ps ax ¦ grep dlmain
  13645 pts/1    S+     0:00 ./dlmain
$ cat /proc/13645/maps
[...]
7f3756827000-7f3756828000 r--p 00000000 103:01 555195     /home/ubuntu/lib2/libm.so.6
[...]
7f375685a000-7f375685b000 rw-p 0002d000 103:01 31786      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f375685b000-7f375685c000 rw-p 00000000 00:00 0
7fff69863000-7fff69884000 rw-p 00000000 00:00 0                          [stack]
[...]
```

- strace 명령을 이용하여 시스템 호출을 추적한다. 라이브러리 파일(.so)을 읽어서 메모리에 로딩하는 과정을 확인할 수 있다.

```
$ strace ./dlmain
execve("./dlmain", ["./dlmain"], 0x7ffcb79bb190 /* 27 vars */) = 0
[...]
openat(AT_FDCWD, ".//libm.so.6", O_RDONLY¦O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@\20\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG¦0775, st_size=15736, ...}) = 0
[...]
mmap(NULL, 16424, PROT_READ, MAP_PRIVATE¦MAP_DENYWRITE, 3, 0) = 0x7f9baff2b000
mmap(0x7f9baff2c000, 4096, PROT_READ¦PROT_EXEC, MAP_PRIVATE¦MAP_FIXED¦MAP_DENYWRITE, 3, 0x1000)
= 0x7f9baff2c000
mmap(0x7f9baff2d000, 4096, PROT_READ, MAP_PRIVATE¦MAP_FIXED¦MAP_DENYWRITE, 3, 0x2000) = 0x7f9ba
```

```
ff2d000
mmap(0x7f9baff2e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x200
0) = 0x7f9baff2e000
close(3)                                = 0
mprotect(0x7f9baff2e000, 4096, PROT_READ) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
write(1, "99.990000\n", 1099.990000
)             = 10
[...]
exit_group(0)                           = ?
+++ exited with 0 +++
```

※ dlopen 함수가 호출되면 OS는 다음과 같은 단계를 수행한다.
- 전달된 파일 경로를 기준으로 라이브러리의 위치를 찾는다. LD_LIBRARY_PATH 환경 변수나 표준 라이브러리 디렉터리에서 파일을 찾는다.
- 라이브러리 파일을 읽어 ELF 형식 파일의 헤더를 파싱한다.
- 메모리 메핑을 위하여 mmap 시스템 호출을 통하여 라이브러리를 프로세스 주소 공간에 매핑함.
- 라이브러리에서 사용되는 심볼의 실제 메모리 위치를 찾기 위하여 심볼 테이블을 참고한다.
- 재배치 정보(relocation table)을 이용하여 코드와 데이터 내의 포인터를 올바른 메모리 주소로 수정한다.
- 라이브러리가 로드될 때 한번 실행되는 초기 코드인 초기화 루틴(.init)을 실행한다.

※ dlsym 함수가 호출되면 OS는 다음과 같은 단계를 수행한다.
- 전달된 심볼 이름을 기준으로 라이브러리의 심볼 테이블에서 해당 심볼을 찾는다.
- 심볼이 발견되면 그 주소를 반환한다.

※ dlclose 함수가 호출되면 OS는 다음과 같은 단계를 수행한다.
- 라이브러리의 참조 카운트를 감소시킨다. 참조카운트가 0이되면 라이브러리를 언로드 할 수 있게 됨
- 라이브러리의 종료루틴(.fini)을 실행
- munmap 을 사용하여 라이브러리를 프로세스의 주소 공간에서 언로드 하고 메모리를 해제한다.

※ DLL(dynamic-link library)은 MS Windows 및 OS/2 운영체제에서 사용되는 동적 링크 라이브러리이다. 실행코드, 데이터, 리소스(resource)가 .dll 파일에 포함될 수 있다. 의존성 지옥(dependency hell)이란. 소프트웨어 패키지의 업그레이드시 생기는 패키지간의 버전 의존성에 의하여 발생되는 문제를 말한다. 응용프로그램이 의존하고 있는 라이브러리들이 긴 사슬 처럼 서로 의존하는 경우(long chains of dependency), 서로 다른 응용프로그램이 같은 라이브러리의 서로 다른 버전에 의존하여 충돌하는 경우(conflicting dependency), 의존하는 라이브러리들간에 순환적인 의존성(circular dependency)을 갖는 경우 이와 같은 문제가 발생할 수 있다. DLL 지옥(DLL hell)이란 말도, MS 윈도우 기반의 응용프로그램에서 DLL간의 의존성이 복잡함으로 인하여 만들어진 용어로, Rick Anderson이 2000년 1월에 발표한 <The End of DLL Hell>이라는 문서를 통해 소개된 바 있다.

3. 프로세스 디버깅

3.1 ptrace 개요

- 프로세스 디버깅은 특정 프로그램의 문제를 파악하고 어떻게 동작하는지 분석하는 과정이다. 디버깅을 위해서는 디버거(debugger)와 같은 도구를 이용하는 방법, 프로세스의 로그를 분석하는 방법, 프로파일러(profiler)와 같이 프로그램의 실행 시간에 정보를 수집하고 분석하는 방법등이 사용된다. 이 중에서 디버거는 프로세스의 실행중 중단점(break point)를 설정하거나, 프로세스의 상태(변수, 메모리)를 검사하고 변경할 수 있는 기능 등을 제공한다.
- ptrace[9] 시스템 호출은 한 프로세서(tracer)가 다른 프로세스(tracee)의 실행을 모니터링하고 제어할 수 있는 수단을 제공한다. 경우에 따라서는 제어 대상이 되는 프로세서의 메모리나 레지스터를 수정할 수 있다. 프로그램을 디버깅 하기 위한 용도로 흔히 사용된다.
- 추적은 반드시 쓰레드(thread) 단위로 이루어 진다. 다중쓰레드(multithreaded) 프로세스는 각각의 개별 쓰레드 단위로 서로 다른 추적자(tracer)에게 연결된다. 이를 위해서는 추적의 대상이 되는 프로세스에서 먼저 `ptrace()` 시스템 호출을 해 주어야 한다.

- ptrace 호출 형식

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,  void *addr, void *data);
```

- `request` 는 수행할 행동을 결정한다.

값	설명
PTRACE_TRACEME	이 프로세스가 부모에 의해 추적될 것임을 나타낸다. 피추적자가 호출
PTRACE_PEEKTEXT	피추적자 메모리의 주소 addr에서 워드를 읽고 그 워드를 ptrace() 호출 결과로 반환
PTRACE_POKETEXT	워드 data를 피추적자 메모리 주소 addr로 복사한다.
PTRACE_GETREGS PTRACE_GETFPREGS	피추적자의 범용 레지스터(또는 부동소수점 레지스터)들을 추적자 내의 주소 data로 복사한다.
PTRACE_CONT	정지된 피추적 프로세스를 재시작한다. data가 0이 아니면 피추적자에게 보낼 시그널 번호로 해석한다.
PTRACE_SYSCALL PTRACE_SINGLESTEP	정지된 피추적 프로세스를 재시작하되 다음번 시스템 호출 진입이나 퇴장에서 또는 한 명령을 실행한 후에 피추적자가 멈추도록 해 놓는다.
PTRACE_KILL	피추적자에게 SIGKILL을 보내서 종료시킨다.

[표 3] ptrace 시스템 호출에서 request의 종류

9) https://man7.org/linux/man-pages/man2/ptrace.2.html

3.2 디버거 기본

- 디버거가 동작을 설명하기 위해 아래의 코드를 작성한다. How Debuggers Work 사이트[10]를 참고 하였음을 밝힌다.
- main() 함수에서 fork() 시스템 호출을 통하여 자식(child) 프로세스를 분기하고, 부모 프로세스에서는 분기된 자식 프로세스를 추적함. 자식 프로세스는 디버깅의 대상이 되는 target 또는 tracee가 됨.
- 자식 프로세스는 ptrace() 시스템 호출을 하여 OS 커널에 부모 프로세스가 자신을 추적할 수 있도록 요청함. 또한 execl을 호출하여 인수로 제공된 프로그램으로 자기 자신을 대체시킴
- 부모 프로세스는 wait는 분기된 자식 프로세스가 중지될때 까지 기다린다. ptrace()호출시 PTRACE_SINGLESTEP 요청으로 자식 프로세스의 ID를 요청한다. OS는 자식 프로세스를 시작하되 다음 명령을 실행한 후에 곧바로 중지한다. icounter는 하위 프로세스가 실행한 명령의 수를 계산한다.

```
// tracer1.c (from How Debuggers Work)
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/ptrace.h>

// process ID와 함께 메시지를 출력
void procmsg(const char* format, ...) {
    va_list ap;
    fprintf(stdout, "[%d] ", getpid());
    va_start(ap, format);
    vfprintf(stdout, format, ap);
    va_end(ap);
}
// child(=target = tracee) 프로세스
void run_target(const char* programname) {
    procmsg("target started. will run '%s'\n", programname);
    // 이 프로세스가 parent 에 의해 추적될 것을 OS에게 알림
    if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) {
        perror("ptrace");  return;
    }
    // 프로세스 이미지를 주어진 프로그램으로 대체하여 실행
    execl(programname, programname, NULL);
}
// parent(= debugger = tracer) 프로세스
void run_debugger(pid_t child_pid) {
    int wait_status;
    unsigned icounter = 0;
    procmsg("debugger started\n");

    // child가 첫번째 명령에서 중지될때까지 대기
    wait(&wait_status);

    while (WIFSTOPPED(wait_status)) {
```

10) https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1/

```
        icounter++;
        // child가 다음 명령을 실행하도록 함
        if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
            perror("ptrace");
            return;
        }
        // 다음 명령이 끝날때까지 대기
        wait(&wait_status);
    }
    procmsg("the child executed %u instructions\n", icounter);
}
int main(int argc, char** argv) {
    pid_t  child_pid;
    if (argc < 2) {
        fprintf(stderr, "Expected a program name as argument\n"); return -1;
    }
    child_pid = fork();
    if (child_pid == 0) // child process
        run_target(argv[1]);
    else if (child_pid > 0) // parent process
        run_debugger(child_pid);
    else {
        perror("fork"); return -1;
    }
    return 0;
}
```

- 다음과 같은 간단한 프로그램을 컴파일 하고 추적 프로그램에서 실행한다.

```
// traced_helloworld.c
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

- 32비트로 컴파일 하기 위해 -m32 옵션을 사용한다.
 ※ 32비트 컴파일을 위해 multilib 관련 라이브러리가 요구된다.

    sudo apt-get install gcc-multilib g++-multilib

```
$ gcc -m32 -o tracer1.o -c  tracer1.c
$ gcc -m32 -o tracer1   tracer1.o
$ gcc -m32 -o tracee_shared  traced_helloworld.c
```

```
$ file  tracee_shared
tracee_shared: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.2, BuildID[sha1]=7260e47e7f730b49884265d94803cf47f68cbc83, for GNU
/Linux 3.2.0, not stripped
```

- tracer 프로그램의 인수로 추적할 프로그램의 실행파일을 설정하여 실행한다.
 비교적 간단한 프로그램임에도 불구하고, 19만개 이상의 명령이 실행되었음을 알 수 있다.
 타켓은 동적으로 링크된 프로그램으로 실행시 공유 라이브러리 로더를 통해 실행된다.

```
$ ./tracer1  tracee_shared
[3259] debugger started
[3260] target started. will run 'tracee_shared'
Hello, world!
[3259] the child executed 194273 instructions
```

- 타켓을 정적 링크하기 위해 -static 옵션을 사용한다.
  이번에는 비교적 적은 수인 2만8천개의 명령이 실행되었음을 알 수 있다.

```
$ gcc -m32 -static -o tracee_static  traced_helloworld.c
$ file tracee_static
tracee_static: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linke
d, BuildID[sha1]=02a8aa756c47ef6796904689bf7b2d7337894988, for GNU/Linux 3.2.0, not stripped
$ ./tracer1  tracee_static
[3292] debugger started
[3293] target started. will run 'tracee_static'
Hello, world!
[3292] the child executed 28214 instructions
```

- C코드가 아닌 어셈블리를 이용하여 타켓코드를 작성할 수도 있다.
  이 경우 단지 7개의 명령이 실행됨을 확인한다.

```
; tracee_hello.asm
section    .text
    ; The _start symbol must be declared for the linker (ld)
    global _start
_start:
    ; Prepare arguments for the sys_write system call:
    ;   - eax: system call number (sys_write)
    ;   - ebx: file descriptor (stdout)
    ;   - ecx: pointer to string
    ;   - edx: string length
    mov    edx, len
    mov    ecx, msg
    mov    ebx, 1
    mov    eax, 4
    int    0x80    ; Execute the sys_write system call
    mov    eax, 1
    int    0x80    ; Execute sys_exit
section   .data
msg db    'Hello, world!', 0xa
len equ    $ - msg
```

```
$  nasm -f elf -o tracee_hello.o tracee_hello.asm
$  ld -m elf_i386 -s tracee_hello.o -o tracee_asm
$ file tracee_asm
tracee_asm: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, stripp
ed
$ ./tracee_asm
Hello, world!
```

```
$ ./tracer1   tracee_asm
[3337] debugger started
[3338] target started. will run 'tracee_asm'
Hello, world!
[3337] the child executed 7 instructions
```

3.3 프로세스 상태 추적

- trace1와 유사하지만 약간 다른 목적으로 디버거를 사용하기 위해 trace2.c 코드를 다음과 같이 작성한다.
- 달라진 부분은 while 루프 내부에서 PTRACE_GETREGS 명령으로 ptrace() 시스템 호출을 하는 부분으로 프로세스의 레지스터 정보를 읽는 역할을 한다. user_regs_struct 구조체는 sys/user.h 에 정의 된다.

```
// tracer2.c (tracer1.c 에서 수정)
[...]
#include <sys/user.h>
[...]
void run_debugger(pid_t child_pid)
{
    int wait_status;
    unsigned icounter = 0;
    procmsg("debugger started\n");

    wait(&wait_status);   // child가 첫번째 명령에서 중지될때까지 대기

    while (WIFSTOPPED(wait_status)) {
      icounter++;

      struct user_regs_struct regs;
      ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
#if __x86_64__
        unsigned instr = ptrace(PTRACE_PEEKTEXT, child_pid, regs.rip, 0);
        procmsg("icounter = %u.  rip = 0x%08x.  instr = 0x%08x\n",
          icounter, regs.rip, instr);
# else
        unsigned instr = ptrace(PTRACE_PEEKTEXT, child_pid, regs.eip, 0);
        procmsg("icounter = %u.  eip = 0x%08x.  instr = 0x%08x\n",
          icounter, regs.eip, instr);
#endif
      // child가 하나의 명령을 실행한 후 멈추도록 한다.
      if (ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0) {
          perror("ptrace");
          return;
      }
      // 다음 명령에서 멈출때까지 기다림
      wait(&wait_status);
    }
    procmsg("the child executed %u instructions\n", icounter);
}
[...]
```

- 컴파일 후 앞의 방법과 같이 실행하면 프로세서의 EIP 값과 instr 값이 출력된다.

```
$ gcc -m32 -o tracer2  tracer2.c
$ ./tracer2  tracee_static
[8562] debugger started
[...]
[8562] icounter = 26925.  eip = 0x0806e825.  instr = 0x14ec8353
[8562] icoHello, world!
```

```
unter = 26926.  EIP = 0x0806e826.  instr = 0x8b14ec83
[8562] icounter = 26927.  eip = 0x0806e829.  instr = 0x20245c8b
[8562] icounter = 26928.  eip = 0x0806e82d.  instr = 0x24244c8b
[...]
[8562] the child executed 27337 instructions
$ ./tracer2  tracee_asm
[3442] debugger started
[3443] target started. will run 'tracee_asm'
[3442] icounter = 1.  eip = 0x08049000.  instr = 0x00000eba
[3442] icounter = 2.  eip = 0x08049005.  instr = 0x04a000b9
[3442] icounter = 3.  eip = 0x0804900a.  instr = 0x000001bb
[3442] icounter = 4.  eip = 0x0804900f.  instr = 0x000004b8
[3442] icounter = 5.  eip = 0x08049014.  instr = 0x01b880cd
Hello, world!
[3442] icounter = 6.  eip = 0x08049016.  instr = 0x000001b8
[3442] icounter = 7.  eip = 0x0804901b.  instr = 0x000080cd
[3442] the child executed 7 instructions
```

- 타켓 프로그램(앞에서 어셈블리로 작성한 버전)을 objdump 명령으로 역어셈블(-d) 하여 보면 다음과 같다. EIP는 instruction pointer로 현재 실행중인 코드의 위치를 의미함.

```
$ objdump -d tracee_asm
tracee_asm:     file format elf32-i386

Disassembly of section .text:
08049000 <.text>:
 8049000:  ba 0e 00 00 00        mov    $0xe,%edx
 8049005:  b9 00 a0 04 08        mov    $0x804a000,%ecx
 804900a:  bb 01 00 00 00        mov    $0x1,%ebx
 804900f:  b8 04 00 00 00        mov    $0x4,%eax
 8049014:  cd 80                 int    $0x80
 8049016:  b8 01 00 00 00        mov    $0x1,%eax
 804901b:  cd 80                 int    $0x80
```

- 64비트의 경우도 거의 동일함.
  ※ 64비트에서는 EIP대신 RIP를 사용해야 함.(__x86_64__ 매크로 이용)
  타켓 프로그램도 64비트 다시 컴파일 해 주어야 함.

```
$ gcc -m64 -o tracer1  tracer1.c
$ gcc -m64 -o tracer2  tracer2.c
$ gcc -m64 -o tracee_shared  traced_helloworld.c
$ file tracee_shared
tracee_shared: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, inte
rpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=5e27d97899214b11afa51433ed4ab8062dfd28cd, fo
r GNU/Linux 3.2.0, not stripped
$ nasm -f elf64 -o tracee_hello.o tracee_hello.asm
$ ld -m elf_x86_64  -s tracee_hello.o -o tracee_asm64
$ file tracee_asm64
tracee_asm64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, stripped
$ ./tracer2 tracee_asm64
[3545] debugger started
[3546] target started. will run 'tracee_asm64'
[3545] icounter = 1.  RIP = 0x00401000.  instr = 0x00000eba
```

```
[3545] icounter = 2.  RIP = 0x00401005.  instr = 0x402000b9
[3545] icounter = 3.  RIP = 0x0040100a.  instr = 0x000001bb
[3545] icounter = 4.  RIP = 0x0040100f.  instr = 0x000004b8
[3545] icounter = 5.  RIP = 0x00401014.  instr = 0x01b880cd
Hello, world!
[3545] icounter = 6.  RIP = 0x00401016.  instr = 0x000001b8
[3545] icounter = 7.  RIP = 0x0040101b.  instr = 0x000080cd
[3545] the child executed 7 instructions
```

3.4 중단점

■ 동작 원리

- 중단점(breakpoint)은 디버깅의 중요한 기능중의 하나임. 중단점을 구현하려면 소프트웨어 인터럽트(또는 트랩)을 사용해야 함. 인터럽트가 발생하면 CPU는 현재 실행을 중지하고 상태를 저장한 후 인터럽트 핸들러 루틴으로 점프하고, 핸들러가 작업을 마치면 중지된 위치부터 실행을 재개한다.
- X86 에서는 INT3 이라는 특수명령을 통하여 디버거를 위한 예외 처리를 제공함. 인텔의 아키텍쳐 SDM(software developer's manual) 문서에는 다음과 같이 opcode가 정의됨
- INT3명령의 opcode는 1바이트(CC)이며 디버그 예외처리 핸들러를 호출하기 위한 용도이다.

INT *n*/INTO/INT 3—Call to Interrupt Procedure

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT 3	ZO	Valid	Valid	Interrupt 3—trap to debugger.
CD *ib*	INT *imm8*	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	ZO	Invalid	Valid	Interrupt 4—if overflow flag is 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm8	NA	NA	NA

- 프로그램이 INT3 명령을 실행하면 OS는 프로그램을 중지하고, SIGTRAP 신호를 전송한다. 추적(디버거) 프로세스가 SIGTRAP 신호 알림을 받으면 이를 처리한다.

- INT3 명령을 테스트하기 위한 코드. 인라인 어셈블리 코드에서 INT3을 직접 호출. gdb로 테스트 해보면 해당 코드에서 SIGTRAP 시그널이 감지되면서 중지된다.

```
// breakpoint.c
int main() {
    int i;
    for(i=0; i<3;i++) {
        __asm__("int3");
    }
}
```

```
$ gcc -m32 -g -o breakpoint32  breakpoint.c
$ ./breakpoint32
Trace/breakpoint trap (core dumped)
$ gdb ./breakpoint32
(gdb) run
```

```
Starting program: /home/ubuntu/sources/tmp/breakpoint32

Program received signal SIGTRAP, Trace/breakpoint trap.
main () at breakpoint.c:4
4       for(i=0; i<3;i++) {
; INT3을 만나면서 중지됨. continue를 3번 번복하여 프로그램 종료
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
main () at breakpoint.c:4
4       for(i=0; i<3;i++) {
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
main () at breakpoint.c:4
4       for(i=0; i<3;i++) {
(gdb) c
Continuing.
[Inferior 1 (process 16089) exited normally]
```

※ 실행이 중지된 상태에서 ps 명령으로 프로세스 상태를 확인. t는 디버거의 추적기능으로 중단됨을 의미함. /proc/<pid>/status 파일에서 어떤 프로세서가 추적하고 있는지 알 수 있다.

```
$ ps ax ¦ grep breakpoint
  60544 pts/0    S+     0:00 gdb ./breakpoint32
  60553 pts/0    t      0:00 /home/ubuntu/sources/breakpoint32
$ egrep "State¦Tracer" /proc/60553/status
State: t (tracing stop)
TracerPid: 60544
```

- 예제 코드. 문자열을 출력하기 위해 시스템 호출(sys_write)을 2번 수행한다. 컴파일 후 실행하여 결과를 확인한다.

```
; tracee_break32.asm
section    .data
    msg1    db      'Hello,', 0xa
    len1    equ     $ - msg1
    msg2    db      'world!', 0xa
    len2    equ     $ - msg2
section    .text
    global _start
_start:
    mov     ecx, msg1
    mov     edx, len1
    mov     ebx, 1
    mov     eax, 4
    int     0x80  ; sys_write

    mov     ecx, msg2
    mov     edx, len2
    mov     ebx, 1
    mov     eax, 4
    int     0x80  ; sys_write
```

```
    mov   eax,1
    int     0x80
$ nasm -f elf  -o tracee_break32.o  tracee_break32.asm
$ ld -m elf_i386  -s tracee_break32.o   -o tracee_break_asm32
$ file ./tracee_break_asm32
./tracee_break_asm32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically link
ed, stripped
$ ./tracee_break_asm32
Hello,
world!
```

- 빌드된 바이너리를 objdump 명령을 이용하여 역어셈블해 보면 다음과 같다.

```
$ objdump -d tracee_break_asm32
tracee_break_asm32:     file format elf32-i386

Disassembly of section .text:

08049000 <.text>:
 8049000:  b9 00 a0 04 08         mov    $0x804a000,%ecx
 8049005:  ba 07 00 00 00         mov    $0x7,%edx
 804900a:  bb 01 00 00 00         mov    $0x1,%ebx
 804900f:  b8 04 00 00 00         mov    $0x4,%eax
 8049014:  cd 80                  int    $0x80
 8049016:  b9 07 a0 04 08         mov    $0x804a007,%ecx
 804901b:  ba 07 00 00 00         mov    $0x7,%edx
 8049020:  bb 01 00 00 00         mov    $0x1,%ebx
 8049025:  b8 04 00 00 00         mov    $0x4,%eax
 804902a:  cd 80                  int    $0x80
 804902c:  b8 01 00 00 00         mov    $0x1,%eax
 8049031:  cd 80                  int    $0x80
```

- 소프트웨어 트랩을 이용하여 중단점을 설정하려면 타겟 명령의 첫번째 바이트를 int3 명령으로 바꾸어 준다. 위예제에서는 첫번째 syscall이 종료된 바로 직후의 명령임.

```
// tracer3.c (tracer2.c 에서 run_debugger 부분만 수정)
#include <string.h>
[...]
void run_debugger(pid_t child_pid) {
  int wait_status;
  struct user_regs_struct regs;
  procmsg("debugger started\n");
  wait(&wait_status);  // child가 첫번째 명령에서 stop될때까지 대기

  // 중단점을 설정할 주소에서 데이터를 읽음
  unsigned addr = TARGETADDRESS;
  unsigned data = ptrace(PTRACE_PEEKTEXT, child_pid, addr, NULL);
  procmsg("peek : [0x%08x] ==> 0x%08x\n", addr,  data);

  // 트랩 명령(int 3)을 해당 주소에 쓰기
  unsigned data_with_trap = (data & 0xffffff00) | 0xcc; // 64비트 에서 long unsigned 임
  procmsg("poke : [0x%08x] <== 0x%08x\n", addr,  data_with_trap);
  ptrace(PTRACE_POKETEXT, child_pid, addr, data_with_trap);
```

```
  data_with_trap = ptrace(PTRACE_PEEKTEXT, child_pid, addr, NULL);
  procmsg("peek : [0x%08x] ==> 0x%08x\n", addr,  data_with_trap);
  // child를 재시작한다.
  ptrace(PTRACE_CONT, child_pid, 0, 0); // continue

  wait(&wait_status); // 위 주소 번에서 트랩이 발생될때까지 대기

  if (WIFSTOPPED(wait_status)) {
    procmsg("Child got a signal: %s\n", strsignal(WSTOPSIG(wait_status)));
  } else {
    perror("wait"); return;
  }

  // See where the child is now
  ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
  procmsg("Child stopped at eip = 0x%08x\n", regs.eip);
}
[...]
$ gcc -m32 -o tracer3 -DTARGETADDRESS=0x8049016  tracer3.c
$ ./tracer3  tracee_break_asm32
[15723] debugger started
[15724] target started. will run 'tracee_break_asm32'
[15723] peek : [0x08049016] ==> 0x04a007b9
[15723] poke : [0x08049016] <== 0x04a007cc
[15723] peek : [0x08049016] ==> 0x04a007cc
Hello,
[15723] Child got a signal: Trace/breakpoint trap
[15723] Child stopped at eip = 0x08049017
```

- 첫번째 sys_write가 호출된 이후에 트랩(Trace/breakpoint)이 발생되었음을 알 수 있다. 이때의 EIP레지스터의 값은 0x08049017 으로 중단점을 설정한 주소(0x8049016)에서 정확하게 1 바이트가 증가됨을 알 수 있다.
- 트랩이 발생된 이후 변경했던 원래의 데이터를 복원하고, 프로세스를 다시 진행한다. 정상적으로 프로세스가 실행되고, 두 번째 메지지가 출력되는 것을 확인함.

```
// tracer4.c (tracer3.c 에서 아래 내용을 추가)
[...]
void run_debugger(pid_t child_pid) {
[...]
  wait(&wait_status); // 위 주소 번에서 트랩이 발생될때까지 대기
  if (WIFSTOPPED(wait_status)) {
    procmsg("Child got a signal: %s\n", strsignal(WSTOPSIG(wait_status)));
  } else {
    perror("wait"); return;
  }

  // see where the child is now
  ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
  procmsg("Child stopped at eip = 0x%08x\n", regs.eip);

  // remove the breakpoint by restoring the previous data
  procmsg("poke : [0x%08x] <== 0x%08x\n", addr,  data);
  ptrace(PTRACE_POKETEXT, child_pid, addr, data);
```

```
  regs.eip = addr;
  procmsg("continue at eip <= 0x%08x\n", regs.eip);
  ptrace(PTRACE_SETREGS, child_pid, 0, &regs);

  ptrace(PTRACE_CONT, child_pid, 0, 0);

  procmsg("parent wait.. \n");
  pid_t waitPid;
  waitPid = wait(&wait_status);

  procmsg("parent terminate.\n");
}
$ gcc -m32 -o tracer4  -DTARGETADDRESS=0x8049016   tracer4.c
$ ./tracer4 tracee_break_asm32
[15763] debugger started
[15764] target started. will run 'tracee_break_asm32'
[15763] peek : [0x08049016] ==> 0x04a007b9
[15763] poke : [0x08049016] <== 0x04a007cc
[15763] peek : [0x08049016] ==> 0x04a007cc
Hello,
[15763] Child got a signal: Trace/breakpoint trap
[15763] Child stopped at eip = 0x08049017
[15763] poke : [0x08049016] <== 0x04a007b9
[15763] continue at eip <= 0x08049016
[15763] parent wait..
world!
[15763] parent terminate.
```

- 타켓 프로그램을 C언어로 작성하여 테스트

```
// tracee_break.c
#include <stdio.h>
int main() {
  int i = 0;
  printf("Hello i=%d\n", i);
  i = 100;
  printf("Hello i=%d\n", i);
  return 0;
}
$ gcc -m32 -static -O0 -o tracee_break_c32  tracee_break.c
$ ./tracee_break_c32
Hello i=0
Hello i=100
$ objdump -j .text --disassemble=main  ./tracee_break_c32
./tracee_break_c32:     file format elf32-i386

Disassembly of section .text:
08049ce5 <main>:
 8049ce5:  f3 0f 1e fb           endbr32
 8049ce9:  8d 4c 24 04           lea    0x4(%esp),%ecx
 8049ced:  83 e4 f0              and    $0xfffffff0,%esp
 8049cf0:  ff 71 fc              pushl  -0x4(%ecx)
 8049cf3:  55                    push   %ebp
```

```
8049cf4:  89 e5                   mov    %esp,%ebp
8049cf6:  53                      push   %ebx
8049cf7:  51                      push   %ecx
8049cf8:  83 ec 10                sub    $0x10,%esp
8049cfb:  e8 c0 fe ff ff          call   8049bc0 <__x86.get_pc_thunk.bx>
8049d00:  81 c3 00 b3 09 00       add    $0x9b300,%ebx
8049d06:  c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)  ; i = 0
8049d0d:  83 ec 08                sub    $0x8,%esp
8049d10:  ff 75 f4                pushl  -0xc(%ebp)
8049d13:  8d 83 08 f0 fc ff       lea    -0x30ff8(%ebx),%eax
8049d19:  50                      push   %eax
8049d1a:  e8 f1 73 00 00          call   8051110 <_IO_printf> ; printf
8049d1f:  83 c4 10                add    $0x10,%esp
8049d22:  c7 45 f4 64 00 00 00    movl   $0x64,-0xc(%ebp)  ; i = 100
8049d29:  83 ec 08                sub    $0x8,%esp
8049d2c:  ff 75 f4                pushl  -0xc(%ebp)
8049d2f:  8d 83 08 f0 fc ff       lea    -0x30ff8(%ebx),%eax
8049d35:  50                      push   %eax
8049d36:  e8 d5 73 00 00          call   8051110 <_IO_printf> ; printf
8049d3b:  83 c4 10                add    $0x10,%esp
8049d3e:  b8 00 00 00 00          mov    $0x0,%eax
8049d43:  8d 65 f8                lea    -0x8(%ebp),%esp
8049d46:  59                      pop    %ecx
8049d47:  5b                      pop    %ebx
8049d48:  5d                      pop    %ebp
8049d49:  8d 61 fc                lea    -0x4(%ecx),%esp
8049d4c:  c3                      ret
```

```
$ gcc -g -m32 -o tracer4 -DTARGETADDRESS=0x8049d1f  tracer4.c
$ ./tracer4 tracee_break_c32
[15852] debugger started
[15853] target started. will run './tracee_break_c32'
[15852] peek : [0x08049d1f] ==> 0xc710c483
[15852] poke : [0x08049d1f] <== 0xc710c4cc
[15852] peek : [0x08049d1f] ==> 0xc710c4cc
Hello i=0
[15852] Child got a signal: Trace/breakpoint trap
[15852] Child stopped at eip = 0x08049d20
[15852] poke : [0x08049d1f] <== 0xc710c483
[15852] continue at eip <= 0x08049d1f
[15852] parent wait..
Hello i=100
[15852] parent terminate.
```

※ 64비트에서는 poke를 할때 `long unsigned` (8바이트) 단위로 write 되므로 다음과 같이 해 주어야 함.

```
unsigned long data_with_trap = (data & ~0xFFFFFFFFUL) | 0xCC;
```

※ ARM 아키텍쳐에도 X86의 `INT3`와 같은 명령이 존재함. ARM의 `brk #imm` 명령은 프로그램을 중단하고 예외를 발생시킴. `#imm`은 16비트의 정수로 임의의 값으로 지정 가능함. 유사하게 SunMicrosystems의 SPARC 아키텍쳐에는 `ta` (trap always) 명령이 존재함. RISC-V에는 `EBREAK`, `ECALL` 명령이 존재함.

3.5 디버깅 정보

- 현대 컴파일러는 중첩된 제어 구조, 동적 타입의 변수 등의 기능 등을 포함 하고 있는 상위 수준 코드를 기계어(머신코드)로 변환하는 작업을 수행한다. 기계어의 목적은 목적하는 CPU에서 가능하면 빠르게 코드를 수행하는 것이다. C언어에서 대부분의 각 코드 라인은 하나 이상의 머신 코드로 변환되는데, 변수는 컴파일러에 의해서 스텍(stack), 레지스터(register), 또는 최적화된 다른 위치에 배치된다. 객체지향 프로그램의 객체(object)는 기계어로 변환과정에서 메모리 버퍼에서 하드 코딩된 오프셋으로 추상화 될 뿐이다.
- 앞서의 예제에서 중단점을 설정하려고 하는 주소번지를 어떻게 알 수 있을까? 혹은 어떤 특정 변수에 접근하기 위한 주소번지는 어떻게 찾을 수 있을까? 이 물음에 대한 답은 디버깅 정보에 있다.
- 디버깅 정보는 컴파일러에 의해 생성되며, 실행가능한 프로그램(executable)과 원래의 소스코드의 관계를 기술한다. 이 정보는 미리 정의된 형식으로 머신코드와는 별개로 저장된다.
- 디버깅 정보에 관한 정의된 형식은: stabs, COFF, PE-COFF, OMF, IEEE-695, DWARF[11] 등이 있다.

형식	설명
stabs	- 이름은 symbol table strings을 의미함 - Unix의 a.out 객체 파일의 심볼 테이블이며, 디버깅 정보는 string으로 저장됨
COFF	- Common Object File Format - Unix System V Release 3에서 유래 - “common"이라는 이름에도 불구하고, 아키텍쳐마다 조금씩 차이가 있음 - XCOFF(IBM RS/6000), ECOFF(MIPS, Alpha), PE-COFF(Windows) 등의 변형이 존재
PE-COFF	- 마이크로소프트 Windows95에서 사용됨 - MS CodeView(CV4) 디버깅 데이터 포맷을 포함함.
OMF	- Object Module Format으로 CP/M, DOS, OS/2 시스템에 사용됨 - 가장 초보적인 기능을 제공
IEEE-695	- MS와 HP가 1980년대 후반 공동으로 개발. 임베디드 시스템을 대상 - 대부분의 이키텍쳐에 적용할 수 있는 유연한 구조
DWARF 1	- Debugging With Attributed Record Formats - 1988년 Bell랩의 Brian Russell이 개발 - C컴파이러와 sdb 디버거(SVR4)에 사용 - PLSIG(Programming Languages Special Interest Group)이 주도 - 단점은 compact하지 않고, 크기를 많이 차지한다는 점
DWARF 2	- 디버깅 데이터의 사이즈를 줄이고, C++언어에 대한 지원을 추가함.
DWARF 3	- 2003년 Free Standard Group이 주도하여 개발. - 2005년 12월 최종 표준이 릴리즈 됨
DWARF 4	- 2007년 Linux Foundation이 설립됨. - VLIW 구조를 지원, profile 기반의 컴파일러 최적화 지원, 문서화 등의 변화 - 2010년 6월 최종 표준이 릴리즈 됨
DWARF 5	- 2017년 2월 릴리즈됨

[표 4] 디버깅 정보 형식

11) https://dwarfstd.org/

■ DWARF 개요

- 디버깅 정보는 DIE(debugging information entry)들로 구성됨. DIE는 tag라는 정보를 포함함. DIE는 트리구조로 연결되어 있음. DIE의 속성으로는 constant(함수명), 변수(함수의 시작 주소), 다른 DIE에 대한 참조 등으로 구성됨. ULEB128(little endian base 128)인코딩을 사용함.[12)]

■ DWARF 특징

- 언어 독립적 : 다양한 프로그래밍 언어(C, C++, Fortran등)과 호환
- 플랫폼 독립적 : 다양한 운영체제와 하드웨어 플랫폼에서 사용가능
- 풍부한 디버깅 정보: 소스 코드 라인 정보, 변수와 함수의 이름 및 타입, 데이터 구조, 스택 프레임, 최적화 정보 등 다양한 정보를 제공
- 소스 코드와 기계 코드 간의 매핑: 소스 코드의 각 줄이 기계 코드의 어느 부분과 대응하는지에 대한 정보를 포함하여, 디버거가 소스 코드와 기계 코드 간의 관계를 정확히 파악할 수 있게 함.
- 스택 프레임 정보: 함수 호출 스택과 각 스택 프레임에 대한 정보를 제공하여, 디버거가 함수 호출 체인과 각 함수의 지역 변수 및 매개변수를 추적할 수 있게 합니다.
- 타입 정보: 프로그램의 데이터 타입에 대한 상세 정보를 포함하여, 디버거가 변수와 데이터 구조의 타입과 의미를 정확히 이해할 수 있게 합니다.
- 최적화 정보: 컴파일러 최적화에 의해 재배치되거나 제거된 변수와 코드에 대한 정보를 포함하여, 최적화된 코드에서도 디버깅을 가능하게 합니다.
- 확장 가능성: 새로운 프로그래밍 언어나 디버깅 요구사항에 맞추어 확장 가능하며, 다양한 디버거와 호환됩니다.
- 표준화된 형식: 표준화된 형식을 사용하여 다양한 도구와의 상호 운용성을 보장합니다.

■ LEB128 인코딩

- 임의의 큰 정수를 저장하는데 사용되는 가변 길이 압축 인코딩. DWARF 디버그 파일 등에 사용됨. 부호가 없는 경우 ULEB128(unsigned)임.

```
ULEB128예시:      MSB              LSB
십진수 624485 ==>  10011000011101100101   (길이: 20)
                   010011000011101100101   7-비트의 배가 되도록 0을 붙임
                  0100110 0001110 1100101   7-비트씩 나누기
              00100110 10001110 11100101   MSB 1비트를 덧붙이기. 가장 왼쪽 1바이트는 제외
                  0x26     0x8E     0xE5  각 바이트를 16진수로 표현
             ==> 출력 0xE5 8E 26          LSB 부터 출력(little endian)
```

- dwarfdump는 ELF 파일의 디버깅 정보를 출력하는 프로그램이다.

```
 $ sudo apt install dwarfdump
```

- 디버깅 정보를 출력하기 (도움말 출력)

```
 $ dwarfdump  --help
------------------------------------------------------------------------
Print Debug Sections
------------------------------------------------------------------------
```

12) https://en.wikipedia.org/wiki/LEB128

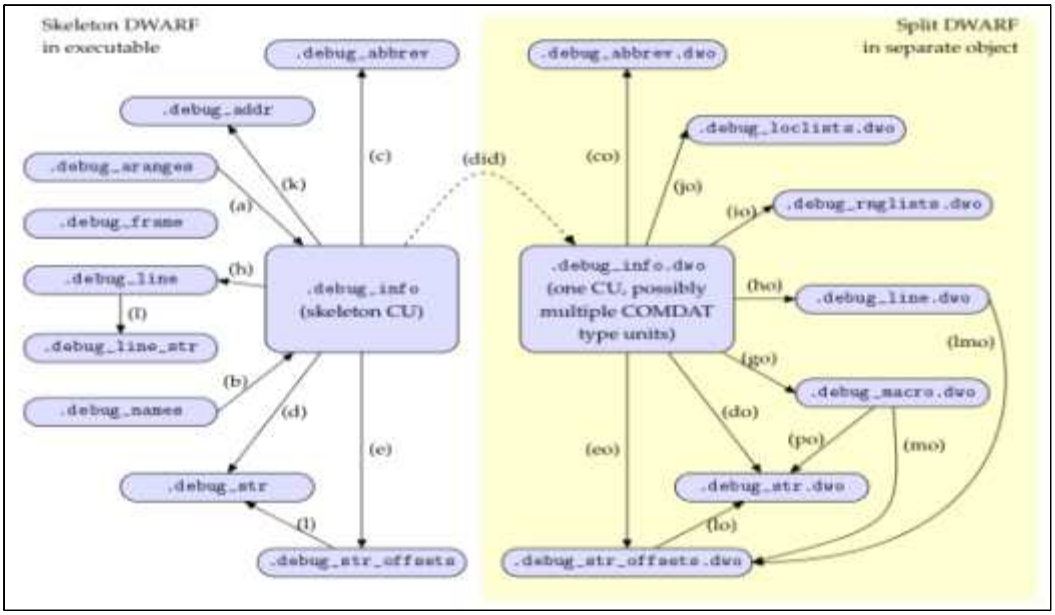
```
-b   --print-abbrev      Print abbrev section
-a   --print-all         Print all debug_* sections
-r   --print-aranges     Print aranges section
-F   --print-eh-frame    Print gnu .eh_frame section
-I   --print-fission     Print fission sections:
                         .gdb_index, .debug_cu_index, .debug_tu_index,
                         .debug_tu_index, .gdb_index, .debug_cu_index,
                         .debug_tu_index, .debug_tu_index, .gdb_index,
                         .debug_cu_index, .debug_tu_index
[...]
```

```
$ dwarfdump  main_static
.debug_info

COMPILE_UNIT<header overall offset = 0x00000000>:
< 0><0x0000000b>  DW_TAG_compile_unit
                    DW_AT_producer              GNU C17 9.4.0 -mtune=generic -march=x86-64 -g -
fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection
                    DW_AT_language              DW_LANG_C99
                    DW_AT_name                  main.c
                    DW_AT_comp_dir              /home/ubuntu/lib
                    DW_AT_low_pc                0x00401d05
                    DW_AT_high_pc               <offset-from-lowpc>32
                    DW_AT_stmt_list             0x00000000
[...]
.debug_str
name at offset 0x00000000, length   11 is '_IO_buf_end'
name at offset 0x0000000c, length   11 is '_old_offset'
[...]
.debug_aranges

COMPILE_UNIT<header overall offset = 0x00000000>:
< 0><0x0000000b>  DW_TAG_compile_unit
                    DW_AT_producer              GNU C17 9.4.0 -mtune=generic -march=x86-64 -g -
fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection
                    DW_AT_language              DW_LANG_C99
                    DW_AT_name                  main.c
                    DW_AT_comp_dir              /home/ubuntu/lib
                    DW_AT_low_pc                0x00401d05
                    DW_AT_high_pc               <offset-from-lowpc>32
                    DW_AT_stmt_list             0x00000000
```

- 디버깅을 위한 ELF 섹션

Section Name	Description
.debug_info	Core DWARF information section
.debug_types	Type descriptions
.debug_str	String table used in .debug_info
.debug_aranges	Lookup table for mapping addresses to compilation units
.debug_loc	Location lists used in the DW_AT_location attributes
.debug_line	Line number information

.debug_abbrev	Provide the definitions for abbreviation codes used in describing the debug info in the .debug_info and .debug_types sections.
.debug_pubnames	Lookup table for global objects and functions
.debug_pubtypes	Lookup table for global types



[그림 2] DWARF 섹션 관계도

- dwarfdump 명령의 주요 옵션

```
dwarfdump [options] file
 -b  --print-abbrev   : abbrev 섹션 출력
 -r  --print-aranges  : aranges 섹션 출력
 -F  --print-eh-frame : .eh_frame 섹션 출력
 -s  --print-strings  : string table 출력
 -l   --print-lines   : line 섹션 출력
 -E   --elf           : object 헤더 섹션 정보 출력
 -S any=<text>        : 속성에서 <text>를 검색
```

- .dwo 파일은 DWARF 디버깅 정보의 일부를 외부의 파일로 분리하여 저장하는 파일 형식임. 큰 프로젝트에서 디버깅 정보를 효율적으로 관리하고, 빌드 시간을 줄이며, 실행 파일의 크기를 줄이는 데 유용함.

※ 디버깅 정보를 별도의 .dwo 파일로 저장하기

```
$ gcc -g -c hello.c -o hello.o -gsplit-dwarf
$ gcc -o hello hello.o
$ objdump -h --wide hello.dwo
hello.dwo:     file format elf64-x86-64
[...]
```

4. 프로파일링

- 소프트웨어공학에서 프로파일링(profiling) 또는 성능 분석은 프로그램의 시간 복잡도 및 공간(메모리) 복잡도, 특정 명령어 실행, 함수 호출 주기 및 빈도 등을 측정하는 분석 기법을 말함.
- 성능 분석 도구는 1970년대 IBM/360과 IBM/370 플랫폼에 존재함. 인터럽트 기반으로 실행코드의 hot spots을 찾아내기 위해 PSW(program status word)에 기록하는 방식으로 동작하였음.[13)]

4.1 GPROF

- GNU Gprof[14)]는 성능 분석 프로파일링 도구임. 프로파일링은 프로그램의 메모리 사용량, 실행 시간을 분석하는 것으로 gprof는 주로 시간복잡성을 측정하여 어떤 부분이 가장 많은 실행시간을 소모하는지 파악하는데 중점을 둠.
- 4.2BSD(Berkeley Unix) 시스템을 위한 버전(by Susan Graham)[15)]과 GNU 프로젝트에서 개발된 버전(by Jay Fenlason)[16)]이 존재함
- 프로파일링을 하려면 프로파일링이 가능하도록 실행 프로그램을 컴파일 해야 하는데 gcc에서 `-pg` 플래그를 사용함. 컴파일된 프로그램을 정상적으로 실행하면 `gmon.out` 파일이 생성됨. 생성된 프로파일 데이터를 `gprof` 를 이용하여 분석 보고서를 생성한다.
- 프로파일링 보고서는 평면 프로파일(flat profile)과 호출 그래프(call graph)정보를 포함. flat profile은 각 함수가 소모한 총 시간과 호출 횟수를 보여준다. call graph는 함수들 간의 호출 관계와 자식 함수들이 소모한 시간을 보여준다.

```
// test_gprof.c
#include <stdio.h>
void new_func1(void);
void func1(void) {
    printf("Inside func1 \n");
    int i = 0;
    for(;i<0xffffffff;i++);
    new_func1();
    return;
}
static void func2(void) {
    printf("Inside func2 \n");
    int i = 0;
    for(;i<0xffffffaa;i++);
    return;
}
int main(void) {
    printf("Inside main()\n");
    int i = 0;
```

13) https://en.wikipedia.org/wiki/IBM_System/360_architecture
14) https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
15) https://www.researchgate.net/publication/243782903_Gprof_A_call_graph_execution_profiler
16) https://www.cs.tufts.edu/comp/150PAT/tools/gprof/gprof.pdf

```
    for(;i<0xffffff;i++);
    func1();
    func2();
    return 0;
}
// test_gprof_new.c
#include <stdio.h>
void new_func1(void) {
    printf("Inside new_func1()\n");
    int i = 0;
    for(;i<0xfffffee;i++);
    return;
}
```

- 컴파일 할때 어셈블리 코드를 확인하기 위해 --save-temps 옵션을 사용

```
$ gcc -Wall -pg -c test_gprof.c --save-temps
$ gcc -Wall -pg -c test_gprof_new.c --save-temps
$ gcc -o test_gprof  -pg test_gprof.o test_gprof_new.o
```

- 어셈블리 코드를 확인하면 각 함수의 앞 부분에 다음과 같이 mcount를 호출하는 동일한 명령이 추가된 것을 확인 할 수 있음

```
$ grep mcount *.s
test_gprof_new.s:1:   call    *mcount@GOTPCREL(%rip)
test_gprof.s:1:   call    *mcount@GOTPCREL(%rip)
test_gprof.s:1:   call    *mcount@GOTPCREL(%rip)
test_gprof.s:1:   call    *mcount@GOTPCREL(%rip)
```

- 프로그램을 실행하면 gmon.out 파일이 생성됨

※ GMON_OUT_PREFIX 환경 변수를 이용하면 디폴트 파일명을 바꿀수 있음

```
  $ GMON_OUT_PREFIX=profile  ./test_gprof
    ==> profile.### 파일이 생성  ###는 프로세스 번호
```

```
$ rm -f gmon.out ; time  ./test_gprof
Inside main()
Inside func1
Inside new_func1()
Inside func2

real    0m17.048s
user    0m17.046s
sys 0m0.000s
$ file gmon.out
gmon.out: GNU prof performance data - version 1
```

※ 프로파일 데이터의 형식은 sys/gmon_out.h 에 의하여 정의된다.

- gprof 명령을 이용하여 프로파일 파일(gmon.out)을 분석한다.

```
  형식:  gprof [options] <image-file> <profile-file>
$ gprof test_gprof gmon.out > analysis.txt
```

- flat 프로파일 결과

```
$ cat analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 43.42      7.92     7.92        1     7.92     7.92  new_func1
 43.09     15.77     7.86        1     7.86     7.86  func2
 13.88     18.30     2.53        1     2.53    10.45  func1
  0.11     18.32     0.02                             main
[...]
```

- 각 컬럼의 의미는 다음과 같다.

% time	함수의 실행시간을 누적했을때 총 실행시간의 비율
cumulative sec.	해당 함수의 총 시행시간(초)
self seconds	다른 자식 함수의 실행시간을 뺀 순수한 실행시간(초)
calls	함수가 호출된 누적 회수
self ms/call	1회 호출당 이 함수에 소요된 평균 시간(밀리초)
total ms/call	1회 호출당 이 함수에 소요된 총 시간(밀리초)
name	함수의 이름

- call graph 프로파일 결과

```
[...]
                     Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 0.05% of 18.32 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02   18.30                 main [1]
                2.53    7.92       1/1           func1 [2]
                7.86    0.00       1/1           func2 [4]
-----------------------------------------------
                2.53    7.92       1/1           main [1]
[2]     57.0    2.53    7.92       1         func1 [2]
                7.92    0.00       1/1           new_func1 [3]
-----------------------------------------------
                7.92    0.00       1/1           func1 [2]
[3]     43.2    7.92    0.00       1         new_func1 [3]
-----------------------------------------------
                7.86    0.00       1/1           main [1]
[4]     42.9    7.86    0.00       1         func2 [4]
-----------------------------------------------
[...]
```

- 각 컬럼의 의미는 다음과 같다.

index	구별을 위한 인덱스 번호
% time	함수에서 소요된 실행 시간(자식 함수를 포함)의 누적 비율
self	해당 함수에서 소요된 시간
children	자식 함수에서 소요된 시간
called	함수가 호출된 회수. recursive call 일때는 +로 표시됨
name	함수명. 인덱스 번호가 함께 표시됨

- 다른 예제. 피보나치 수열을 구하는 재귀함수 호출 프로그램.

```
// fibonacci.c
#include <stdio.h>
static int fibonacci(int n) {
  if (n <= 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
int main(void) {
  printf("Inside main()\n");
  int i = 0, value;
  for(;i<40;i++) {
    value = fibonacci(i);
    printf("%d %d\n", i, value);
  }
  return 0;
}
```

```
$ gcc -pg -o fibonacci  fibonacci.c
$ time ./fibonacci
0 1
1 1
2 2
3 3
[...]
39 102334155

real    0m4.994s
user    0m4.994s
sys 0m0.000s
$ gprof fibonacci  gmon.out  > analysis2.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.59      0.25     0.25       40     6.24     6.24  fibonacci
 26.23      0.34     0.09                             frame_dummy
[...]
                     Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 2.98% of 0.34 seconds
index % time    self  children    called     name
                             535828510             fibonacci [1]
                0.25    0.00      40/40          main [2]
[1]     74.2    0.25    0.00      40+535828510 fibonacci [1]
                             535828510             fibonacci [1]
-----------------------------------------------
                                                  <spontaneous>
[2]     74.2    0.00    0.25                 main [2]
                0.25    0.00      40/40          fibonacci [1]
-----------------------------------------------
                                                  <spontaneous>
[3]     25.8    0.09    0.00                 frame_dummy [3]
-----------------------------------------------
```

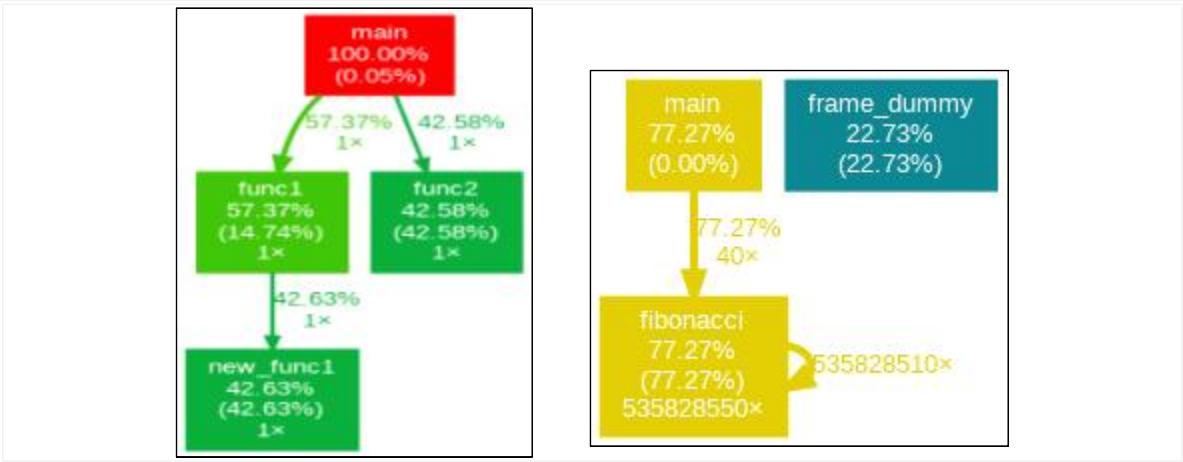
■ 프로파일링 동작 원리[17)]

- 프로파일링은 프로그램의 모든 함수를 컴파일하는 방식을 변경하여 어떤 함수가 몇 번 호출되었는지 추적하는 기법. 이는 '-pg' 옵션을 사용하여 모든 함수가 호출시 mcount 또는 유사한 함수를 호출하게 함으로써 이루어짐. mcount는 호출 그래프를 유지하는 역할을 하며, 스택 프레임을 검사하여 호출 정보를 수집함. 컴파일러와 아키텍처에 따라 다르게 구현방법은 다를 수 있음.
- GCC 2부터는 필요한 정보를 추출하는 __builtin_return_address 함수를 제공하지만, SPARC와 같은 일부 아키텍처에서는 성능상의 이유로 어셈블리 버전의 mcount가 사용됨.
- 프로파일링은 프로그램 실행 시 프로그램 카운터의 위치를 주기적으로 기록하는 히스토그램을 유지하는 작업도 포함. 대부분의 UNIX 시스템은 profil() 시스템 호출을 사용해 프로그램 카운터 위치를 기록하지만, Linux 2.0 이전 버전은 대신 setitimer()를 사용. 후자의 방법은 더 많은 오버헤드를 발생시키며 정확도가 떨어짐.
- 특수 시작 루틴(monstartup)은 히스토그램 메모리를 할당하고 profil()을 설정하거나 신호를 설정. 이 루틴은 특수 프로파일링 시작 파일(gcrt0.o)을 통해 호출
- 프로그램 종료 시 mcleanup 함수가 atexit()을 통해 호출되어 결과를 gmon.out 파일에 기록.
- 바이너리의 심벌을 확인해 보면 mcount, mcleanup, mstartup 등을 확인 가능

```
$ nm test_gprof
[...]
0000000000001258 T main
                 U _mcleanup@@GLIBC_2.2.5
                 U mcount@@GLIBC_2.2.5
                 U __monstartup@@GLIBC_2.2.5
[...]
```

- gprof2dot[18)] 유틸리티를 이용하여 호출 그래프를 시각적으로 확인할 수 있음.

```
설치방법:
  sudo pip install gprof2dot==2019.11.30
  sudo apt install graphviz
$ gprof test_gprof gmon.out | gprof2dot | dot -Tpng -o output.png
```



[그림 3] 두가지 예제 코드드의 gprof결과를 시각적으로 나타낸것

17) https://sourceware.org/binutils/docs/gprof/Implementation.html
18) https://pypi.org/project/gprof2dot/2019.11.30/

4.2 GCOV

- Gcov[19]는 테스트 커버리지 프로그램으로, 코드에서 라인 단위로 실행 빈도를 분석할 수 있는 도구임. gcov 라는 로그 파일을 생성함. 실행파일은 **-fprofile-arcs (gcda 파일 생성)** 및 **-ftest-coverage (.gcno 파일 생성)** 옵션을 이용하여 컴파일 해야 함.

```
// test_gcov.c
#include <stdio.h>
void func1() {
    printf("Hello World \n");
}
void func2(int delay) {
    printf("Delay: %d", delay);
    while (delay--);
}
int main() {
    for (int i = 0; i < 10; ++i) {
        func1();
    }
    for (int i = 0; i < 100; ++i) {
        func2(i);
    }
    return 0;
}
```

```
$ gcc -o test_gcov  test_gcov.c  -fprofile-arcs -ftest-coverage  -g
$ ./test_gcov
[...]
$ file   test_gcov.gcno  test_gcov.gcda
test_gcov.gcno: GCC gcno coverage (-ftest-coverage), version A.4
test_gcov.gcda: GCC gcda coverage (-fprofile-arcs), version A.4
```

※ gcno 파일은 컴파일 타임에 생성되고, gcda 파일은 런타임에 생성됨

- 생성된 파일과 gcov 명령을 이용하여 분석

```
형식:  gcov [-b] [-c] [-v] [-n] [-l] [-f] [-o 디렉토리이름] 소스파일이름
```

```
$ gcov   test_gcov.c
File 'test_gcov.c'
Lines executed:100.00% of 13
Creating 'test_gcov.c.gcov'
```

```
$ cat test_gcov.c.gcov
        -:    0:Source:test_gcov.c
        -:    0:Graph:test_gcov.gcno
        -:    0:Data:test_gcov.gcda
        -:    0:Runs:1
        -:    1:// test_gcov.c
        -:    2:#include <stdio.h>
       10:    3:void func1() {
       10:    4:    printf("Hello World \n");
       10:    5:}
```

19) https://en.wikipedia.org/wiki/Gcov

```
     100:    6:void func2(int delay) {
     100:    7:    printf("Delay: %d", delay);
    5050:    8:    while (delay--);
     100:    9:}
       1:   10:int main() {
      11:   11:    for (int i = 0; i < 10; ++i) {
      10:   12:        func1();
       -:   13:    }
     101:   14:    for (int i = 0; i < 100; ++i) {
     100:   15:        func2(i);
       -:   16:    }
       1:   17:    return 0;
       -:   18:}
```

4.3 ftrace

- ftracer(function tracer)[20][21]는 리눅스 커널용으로 개발된 프로그램 추적 프레임워크임. user space 밖에서 발생되는 코드에 대한 분석이 가능함. 2008년 발표됨.
- 커널에 내장되어 있는 기능으로, debugfs와 함께 작동한다. 대부분의 배포판에서는 /sys/kernel/debug 경로에 마운트 됨. root권한이 필요함.

```
# df -a ¦ grep debugfs
debugfs                 0       0         0    - /sys/kernel/debug
# cd /sys/kernel/debug/tracing
# cat available_tracers
timerlat osnoise hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup function nop
```

※ 마운트 되어 있지 않을 경우 다음과 같이 마운트함.

```
# mount -t debugfs none /sys/kernel/debug/
```

- 기본적으로는 설정된 트레이서가 없고(nop), 활성화 상태가 아닐(0) 것이다.

```
# cat /sys/kernel/debug/tracing/current_tracer
nop
# cat /sys/kernel/debug/tracing/tracing_on
0
```

- 다양한 트레이서가 존재하지만 function_graph 트레이서를 활성화 한다.

```
echo function_graph > /sys/kernel/debug/tracing/current_tracer
echo 1 > /sys/kernel/debug/tracing/tracing_on
```

- 결과는 /sys/kernel/debug/tracing/trace 를 통해 알수 있다.

```
# cat /sys/kernel/debug/tracing/trace
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# ¦     ¦   ¦                     ¦   ¦   ¦   ¦
```

20) https://en.wikipedia.org/wiki/Ftrace
21) https://www.kernel.org/doc/html/latest/trace/index.html

```
 1)   0.137 us    |                          } /* __tlb_remove_page_size */
 0)               |                        __cond_resched() {
 0)   0.139 us    |                          rcu_all_qs();
 0)   0.837 us    |                        }
[...]
```

- 예제 실행

```
# ftrace를 작동함
echo function_graph > /sys/kernel/debug/tracing/current_tracer
echo 1 > /sys/kernel/debug/tracing/tracing_on

# 다른 터미널에서 로그를 수집한다.
cat /sys/kernel/debug/tracing/trace | tee trace

# 타켓 프로그램을 실행 후, 트레이스를 종료함
export LD_LIBRARY_PATH=. ;  ./main_shared
echo 0 > /sys/kernel/debug/tracing/tracing_on
```

- trace-cmd 명령은 ftrace 추적 기능을 활용하는 명령임

```
$ sudo apt  install trace-cmd
# trace-cmd
trace-cmd version 2.8.3
usage:
  trace-cmd [COMMAND] ...
  commands:
     record - 실시간 추적을 기록하고 trace.dat 파일을 저장
     start - 파일에 기록을 하지 않고 추적을 시작
     extract - 커널 버퍼에서 데이터를 추출하고 trace.dat 를 생성
     stop - 추적을 중지
     restart - 이전 중지 부터 추적을 다시 시작
     show - kernel tracing buffer의 내용을 출력함
     reset - 모든 추적을 비활성함. 커널 버퍼에서 모든 데이터를 지움
     clear - ftrace 링 버퍼의 내용을 삭제
     report - trace.dat 파일을 읽고 이진 데이터를 텍스트로 출력
     stream - 추적을 시작하고 출력을 직접 읽음
     profile - 프로파일링을 시작하고 출력을 직접 읽음
     hist - 이벤트 히스토그램을 표시. trace.date 파일을 이용.
     stat - 실행중인 추적(ftrace) 상태를 표시
     split - trace.date 파일을 더 작은 파일로 분할함
     options - 사용할 수 있는 플러그인 목록을 출력
     listen - 클라이언트와 연결할 수 있는 네트워크 포트를 열기
     list - 플로그인, 이벤트 목록을 출력
     restore - 충돌이 발생한 레코딩 파일을 복원
     snapshot - 실행 중인 추적 스택의 스탭샷 찍기
     stack - 스택 트레이서를 활성/비활성화 하고 내용을 출력
     check-events - 모든 이벤트를 파생하여 점검
```

- 프로그램이 실행되는 동안 커널 내에서 실행중인 레코딩 기능을 활성화 함. CPU 및 모든 작업의 기능. myapp과 관련 없는 작업을 모두 기록함.

```
형식 : trace-cmd record -p function myapp
```

```
$ trace-cmd record -h
```

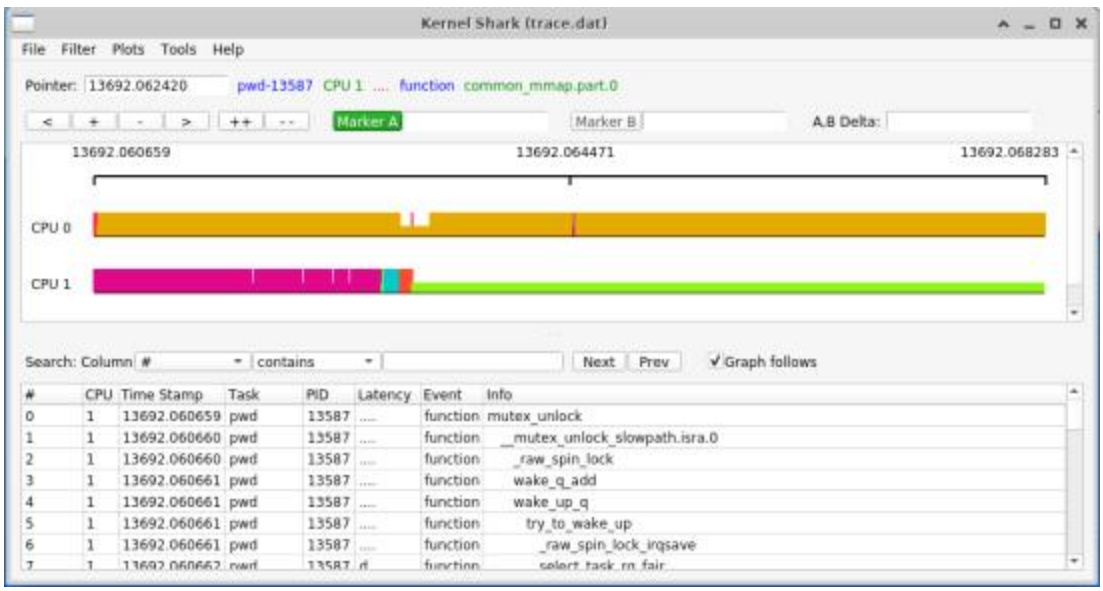
```
$ sudo trace-cmd record -p function  pwd
  plugin 'function'
/root
CPU 0: 19720 events lost
CPU 1: 29290 events lost
CPU0 data recorded at offset=0x70b000
    3018752 bytes in size
CPU1 data recorded at offset=0x9ec000
    2605056 bytes in size
# trace-cmd report
cpus=2
             pwd-13515 [001] 13359.147845: function:             mutex_unlock
             pwd-13515 [001] 13359.147845: function:                __mutex_unlock_slowpath.isra.0
             pwd-13515 [001] 13359.147846: function:                   _raw_spin_lock
[...]
```

- KernelShark[22]는 trace-cmd 의 출력을 읽는 GUI 프론트엔트 유틸리티임. trace-cmd record와 trace-cmd extract 를 통해 입력 데이터를 생성한다.

```
설치 방법 : sudo apt install kernelshark
```

- kernelshark 사용방법

```
$ sudo trace-cmd start -e sched
[...]
$ sudo trace-cmd stop
$ sudo trace-cmd extract -o trace.data
CPU0 data recorded at offset=0x760000
    729088 bytes in size
CPU1 data recorded at offset=0x812000
    1073152 bytes in size
$ kernelshark trace.dat
```



[그림 4] kernelshark 실행 화면

22) https://kernelshark.org/

4.4 strace

- strace[23] (linux syscall tracer)는 시스템 호출과 시그널을 추적하기 위해 사용함. 시스템 호출과 관련한 부분에서 디버깅을 하기 위해, 성능 분석을 위해, 보안 위험과 관련 분석 등을 위해 사용된다. strace는 ptrace 커널 기능을 통해서 구현됨.
- 1991년 SunOS를 위해 개발이 시작됨(by Paul Kranenburg)
- 필터링 옵션(-e)을 이용하여 필요한 기능을 위주로 추적이 가능함.

```
$ strace -h
Usage: strace [-ACdffhikqqrtttTvVwxxyyzZ] [-I N] [-b execve] [-e EXPR]...
              [-a COLUMN] [-o FILE] [-s STRSIZE] [-X FORMAT] [-P PATH]...
              [-p PID]... [--seccomp-bpf]
              { -p PID | [-DDD] [-E VAR=VAL]... [-u USERNAME] PROG [ARGS] }
   or: strace -c[dfwzZ] [-I N] [-b execve] [-e EXPR]... [-O OVERHEAD]
              [-S SORTBY] [-P PATH]... [-p PID]... [--seccomp-bpf]
              { -p PID | [-DDD] [-E VAR=VAL]... [-u USERNAME] PROG [ARGS] }

General:
  -e EXPR        a qualifying expression: OPTION=[!]all or OPTION=[!]VAL1[,VAL2]...
     options:    trace, abbrev, verbose, raw, signal, read, write, fault,
                 inject, status, kvm
[...]
```

```
$ strace -e trace=file ls /nonexistentfile
execve("/usr/bin/ls", ["ls", "/nonexistentfile"], 0x7ffe85d11ee8 /* 26 vars */) = 0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre2-8.so.0", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
statfs("/sys/fs/selinux", 0x7fff9710c360) = -1 ENOENT (No such file or directory)
statfs("/selinux", 0x7fff9710c360)      = -1 ENOENT (No such file or directory)
[...]
: No such file or directory
+++ exited with 2 +++
```

- 예제. 앞에서 작성한 동적로딩(dlmain) 예제. openat 시스템 호출은 프로세스가 파일이나 디렉터리를 열수 있도록 함. 동적 라이브러리가 있는 libm.so 파일을 읽음. mmap 시스템 호출은 파일이나 장치의 내용을 메모리 주소 공간에 직접 매핑함. PROT_READ 속성으로 읽기 전용으로 설정. 등과 같은 내용을 strace 결과로 확인할 수 있음.

```
$ strace ./dlmain
execve("./dlmain", ["./dlmain"], 0x7ffde4d22150 /* 27 vars */) = 0
brk(NULL)                               = 0x562807b13000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffecd755700) = -1 EINVAL (Invalid argument)
[...]
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300\323\0\0\0\0\0\0"..., 832) = 832
```

23) https://strace.io/

```
fstat(3, {st_mode=S_IFREG|0644, st_size=1369384, ...}) = 0
mmap(NULL, 1368336, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f52970bb000
mmap(0x7f52970c8000, 684032, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xd000) = 0x7f5
2970c8000
mmap(0x7f529716f000, 626688, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xb4000) = 0x7f529716f000
mmap(0x7f5297208000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x14c000) = 0x7f
5297208000
close(3)                                = 0
mprotect(0x7f5297208000, 4096, PROT_READ) = 0
munmap(0x7f5297407000, 73340)           = 0
fstat(1, {st_mode=S_IFREG|0664, st_size=9255, ...}) = 0
munmap(0x7f52970bb000, 1368336)         = 0
write(1, "-0.999999\n", 10-0.999999
)             = 10
exit_group(0)                           = ?
```

- strace -c 는 시간, 호출회수, 에러발생 회수 등을 카운트 하여 요약하여 보여줌. 예제는 앞에서 작성한 fibonacci 실행 파일임.

```
$ strace -c ./fibonacci
[...]
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1           read
  0.00    0.000000           0        42           write
  0.00    0.000000           0         3           close
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         7           mmap
  0.00    0.000000           0         3           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           rt_sigaction
  0.00    0.000000           0       484           rt_sigreturn
  0.00    0.000000           0         6           pread64
  0.00    0.000000           0         2           writev
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         2           setitimer
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         1 arch_prctl
  0.00    0.000000           0        19        16 openat
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                   582        18 total
```

4.5 Valgrind

- Valgrind(밸그린드)[24]는 프로세스의 메모리 사용과 메모리 누수(memory leak)과 관련하여 분석할 수 있는 도구이다. 최초 개발자는 Julian Seward이며, 2002년 초기 버전이 발표되었음.
- Valgrind는 본질적으로 동적 프로그램 분석(dynamic binary analysis, DBA)을 가능하게 하는 instrumentation(계측) 도구임. 기계어 수준에서 실행시간에 프로그램의 동작을 분석한다.
- Valgrind의 구조는 코어와 도구들로 구성되어 있으며 제공되는 도구들은 다음과 같다:

도구	설명
Memcheck	메모리 오류 감지기
Cachegrind	캐시 및 분기 예측 프로파일러
Callgrind	호출 그래프를 생성
Helgrind	스레드 오류 감지기
DRD	스레드 오류 감지기
Massif	힙 프로파일러
DHAT	힙 프로파일러

- 빌드 방법. 홈페이지의 다운로드 페이지에서 소스 파일을 받아 빌드함.

```
$ wget https://sourceware.org/pub/valgrind/valgrind-3.23.0.tar.bz2
$ tar jxf valgrind-3.23.0.tar.bz2 ; cd valgrind-3.23.0
$ ./configure --prefix=$HOME/valgrind
$ make ; make install
$ export VALGRIND_LIB=$HOME/valgrind/libexec/valgrind
$ $HOME/valgrind/bin/valgrind --help
usage: valgrind [options] prog-and-args
  tool-selection option, with default in [ ]:
    --tool=<name>             use the Valgrind tool named <name> [memcheck]
                              available tools are:
                              memcheck cachegrind callgrind helgrind drd
                              massif dhat lackey none exp-bbv
[...]
```

- 메모리 누수가 있는 테스트 프로그램을 작성

```
// leak.c
#include <stdlib.h>
void f(void) {
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed
int main(void) {
   f();
   return 0;
}
$ gcc -o leak -g leak.c
```

24) https://valgrind.org/

- 실행방법

```
$ valgrind --tool=memcheck --leak-check=full ./leak
==62228== Memcheck, a memory error detector
==62228== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==62228== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==62228== Command: ./leak
==62228==
==62228== Invalid write of size 4
==62228==    at 0x10916B: f (leak.c:5)
==62228==    by 0x109180: main (leak.c:8)
==62228==  Address 0x4a59068 is 0 bytes after a block of size 40 alloc'd
==62228==    at 0x483C815: malloc (vg_replace_malloc.c:446)
==62228==    by 0x10915E: f (leak.c:4)
==62228==    by 0x109180: main (leak.c:8)
==62228==
==62228==
==62228== HEAP SUMMARY:
==62228==     in use at exit: 40 bytes in 1 blocks
==62228==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==62228==
==62228== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==62228==    at 0x483C815: malloc (vg_replace_malloc.c:446)
==62228==    by 0x10915E: f (leak.c:4)
==62228==    by 0x109180: main (leak.c:8)
==62228==
==62228== LEAK SUMMARY:
==62228==    definitely lost: 40 bytes in 1 blocks
==62228==    indirectly lost: 0 bytes in 0 blocks
==62228==      possibly lost: 0 bytes in 0 blocks
==62228==    still reachable: 0 bytes in 0 blocks
==62228==         suppressed: 0 bytes in 0 blocks
==62228==
==62228== For lists of detected and suppressed errors, rerun with: -s
==62228== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

- cachegrind 실행방법. cachegrind.out 파일이 생성되고, cg_annotate 유틸리티를 이용하여 분석한다.

```
$ valgrind --tool=cachegrind   ./test_gcov
[...]
==62378== I   refs:      331,897
==62378== I1  misses:      1,348
==62378== LLi misses:      1,322
==62378== I1  miss rate:    0.41%
==62378== LLi miss rate:    0.40%
==62378==
==62378== D   refs:      113,464  (77,137 rd   + 36,327 wr)
==62378== D1  misses:      3,294  ( 2,623 rd   +    671 wr)
==62378== LLd misses:      2,748  ( 2,131 rd   +    617 wr)
==62378== D1  miss rate:     2.9% (   3.4%     +    1.8%  )
==62378== LLd miss rate:     2.4% (   2.8%     +    1.7%  )
==62378==
==62378== LL refs:         4,642  ( 3,971 rd   +    671 wr)
==62378== LL misses:       4,070  ( 3,453 rd   +    617 wr)
```

```
==62378== LL miss rate:      0.9% (   0.8%     +    1.7%  )
$ file cachegrind.out.62378
cachegrind.out.62378: ASCII text
$ cg_annotate cachegrind.out.62378
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         49152 B, 64 B, 12-way associative
LL cache:         109051904 B, 64 B, 26-way associative
Command:          ./test_gcov
Data file:        cachegrind.out.62378
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
[...]
```

- Helgrind는 프로그램 동기화 오류를 감지하기 위한 도구임. C, C++, Fortran에서 pthread를 사용한 프로그램에 적용가능.

```
// race.c
#include <pthread.h>
#include <stdio.h>
int counter = 0;  // 공유 변수
void* increment(void* arg) {
    for (int i = 0; i < 10000; ++i) {
        counter++;  // 데이터 레이스 발생 가능
    }
    return NULL;
}
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter: %d\n", counter);
    return 0;
}
```

```
$ gcc -o race race.c -lpthread
$ ./race  ; ./race ; ./race ; ./race
Counter: 20000
Counter: 18554
Counter: 20000
Counter: 18491
```

```
$ valgrind --tool=helgrind ./race
[...]
==72619== ---Thread-Announcement------------------------------------------
==72619==
==72619== Thread #3 was created
[...]
==72619== Thread #2 was created
[...]
==72619== ----------------------------------------------------------------
```

```
==72619==
==72619== Possible data race during read of size 4 at 0x10C014 by thread #3
==72619== Locks held: none
==72619==    at 0x1091BE: increment (in /home/ubuntu/sources/race)
==72619==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-
linux.so)
==72619==    by 0x486B608: start_thread (pthread_create.c:477)
==72619==    by 0x49A5352: clone (clone.S:95)
==72619==
==72619== This conflicts with a previous write of size 4 by thread #2
==72619== Locks held: none
==72619==    at 0x1091C7: increment (in /home/ubuntu/sources/race)
==72619==    by 0x4842B1A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_helgrind-amd64-
linux.so)
==72619==    by 0x486B608: start_thread (pthread_create.c:477)
==72619==    by 0x49A5352: clone (clone.S:95)
==72619==  Address 0x10c014 is 0 bytes inside data symbol "counter"
==72619==
[...]
==72619==
Counter: 20000
==72619==
[...]
==72619== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

- Callgrind는 프로그램 실행시 함수 간의 호출 내역을 그래프로 기록하는 프로파일링 도구임. 프로필데이터는 프로그램 종료시 파일에 기록됨. `callgrind_annotate` 도구를 이용하여 요약된 내용을 출력. `callgrind_control` 명령을 이용하여 프로그램의 실행중에 대화형 제어가 가능하다.

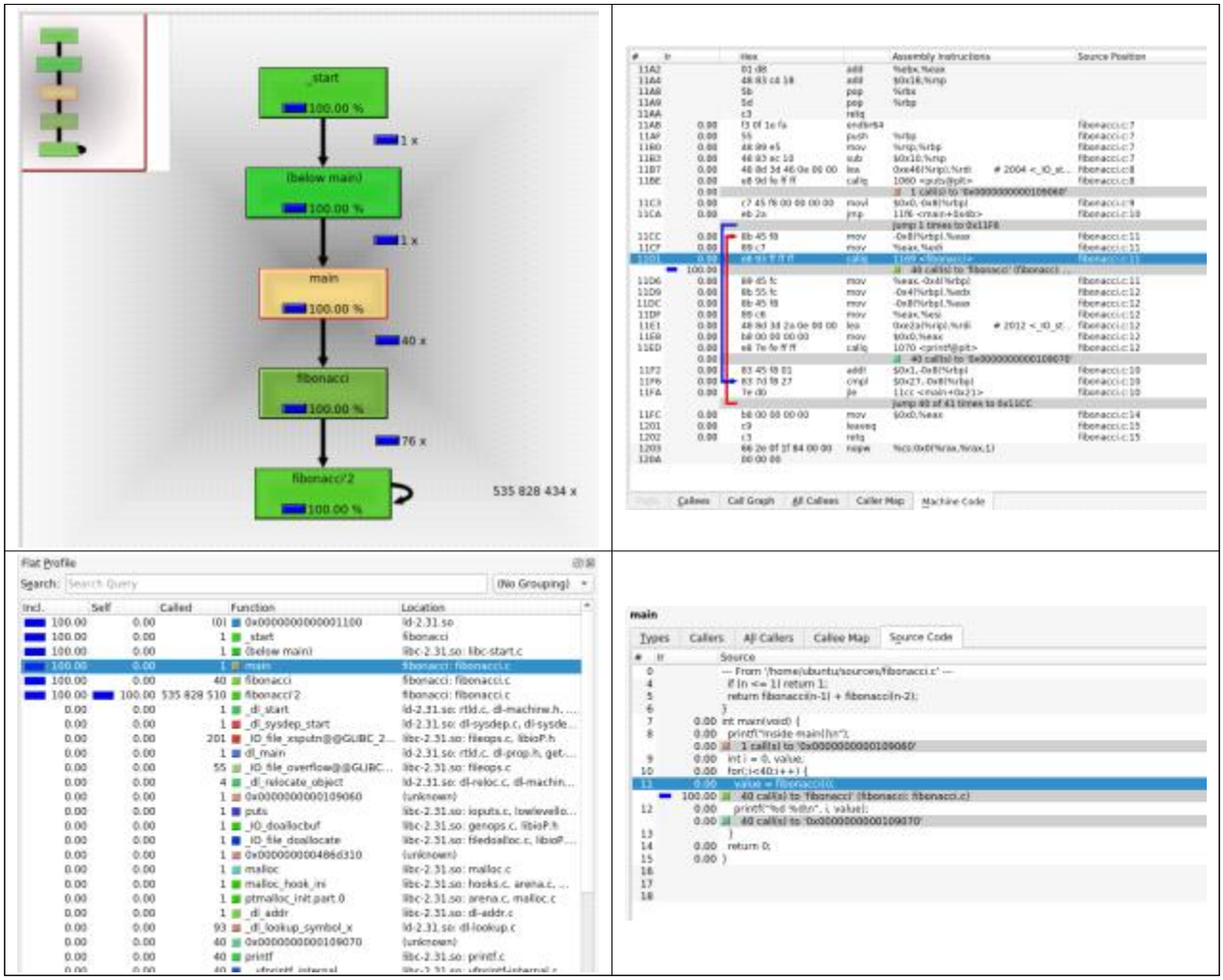
```
$ valgrind --tool=callgrind ./fibonacci
[...]
==72670== For interactive control, run 'callgrind_control -h'.
Inside main()
0 1
1 1
[...]
==72670==
==72670== Events    : Ir
==72670== Collected : 35365181074
==72670==
==72670== I   refs:      35,365,181,074
; callgrind.out.72670 파일이 생성됨
$ callgrind_annotate
Reading data from 'callgrind.out.72670'...
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.72670' (creator: callgrind-3.15.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 7233787287
Trigger: Program termination
```

```
Profiled target:  ./fibonacci (PID 72670, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
35,364,929,217  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir              file:function
--------------------------------------------------------------------------------
13,931,542,322  /build/glibc-e2p3jK/glibc-2.31/gmon/mcount.c:__mcount_internal [/usr/lib/x86_64
-linux-gnu/libc-2.31.so]
11,252,399,571  /build/glibc-e2p3jK/glibc-2.31/gmon/../sysdeps/x86_64/_mcount.S:mcount [/usr/li
b/x86_64-linux-gnu/libc-2.31.so]
10,180,741,386  ???:fibonacci'2 [/home/ubuntu/sources/fibonacci]

$ callgrind_control -b    ; 프로그램 실행중에 실행 stack/back trace 를 조회
PID 72757: ./fibonacci
sending command status internal to pid 72757

  Frame: Backtrace for Thread 1
   [ 0]  __mcount_internal (207124350 x)
   [ 1]  mcount (207124310 x)
   [ 2]  fibonacci (103562109 x)
   [ 3]  fibonacci (103562128 x)
[...]
   [28]  fibonacci (103562128 x)
   [29]  fibonacci (103562128 x)
   [30]  fibonacci (37 x)
   [31]  fibonacci (39 x)   ; return fibonacci(n-1) + fibonacci(n-2);
   [32]  main (1 x)
   [33]  (below main) (1 x)
   [34]  _start (1 x)
   [35]  0x0000000000001100
```

- KCachegrind[25]는 Callgrind의 결과를 가시화 해 주는 GUI 도구이다. 실행 후 `callgrind.out.<pid>` 파일을 로딩함. `valgrind --tool=callgrind --dump-instr=yes --collect-jumps=yes ./program` 옵션을 추가하여 실행

```
설치방법: sudo apt  install kcachegrind
```

25) https://kcachegrind.github.io/html/Home.html



[그림 5] KCachegrind 실행결과

4.6 Perf

- perf[26] 성능 분석 및 모니터링을 위한 도구임. 리눅스 커널에 내장된 성능 카운터를 사용하여 시스템의 다양한 매트릭을 수집하고 분석할 수 있다.
- 설치 방법은 정확한 커널 버전에 맞는 `linux-tools` 패키지를 설치해 주어야 함.

```
$ sudo apt install linux-tools-common
$ uname -r
6.5.0-1018-aws
$ sudo apt install  linux-aws-6.5-tools-6.5.0-1018
$ cat /proc/sys/kernel/perf_event_paranoid
4
$ sudo sysctl kernel.perf_event_paranoid=-1

$ dpkg -L linux-aws-6.5-tools-6.5.0-1018
$ alias perf=/usr/lib/linux-aws-6.5-tools-6.5.0-1018/perf ; perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
```

26) https://en.wikipedia.org/wiki/Perf_(Linux)

```
The most commonly used perf commands are:
  annotate        파일(perf.data, perf record로 생성)을 읽고 분석한다.
  archive         perf.data 파일에서 발견된 빌드ID 를 포함한 객체 파일로 아카이브 생성
  bench           벤치마크 세트를 위한 일반 프레임워크
  buildid-cache   빌드 ID 캐시 관리
  buildid-list    perf.data 파일의 빌드 ID 목록 표시
  c2c             공유 데이터 C2C/HITM 분석기
  config          설정 파일에서 변수 가져오기 및 설정
  daemon          백그라운드에서 기록 세션 실행
  data            데이터 파일 관련 처리
  diff            perf.data 파일을 읽고 차이 프로필 표시
  evlist          perf.data 파일에서 이벤트 이름 목록 표시
  ftrace          커널의 ftrace 기능을 위한 간단한 래퍼
  inject          이벤트 스트림에 추가 정보를 보강하는 필터
  iostat          I/O 성능 지표 표시
  kallsyms        실행 중인 커널에서 심볼 검색
  kmem            커널 메모리 속성을 추적/측정하는 도구
  kvm             KVM 게스트 OS를 추적/측정하는 도구
  list            모든 심볼릭 이벤트 타입 나열
  lock            lock 이벤트 분석
  mem             메모리 접근 프로파일링
  record          명령을 실행하고 그 프로필을 perf.data에 기록
  report          perf record에 의해 생성된 perf.data를 읽고 프로필 표시
  script          perf record에 의해 생성된 perf.data를 읽고 추적 출력 표시
  stat            명령을 실행하고 성능 카운터 통계 수집
  test            기능 테스트 실행
  timechart       작업 중 전체 시스템 동작을 시각화하는 도구
  top             시스템 프로파일링 도구
  version         perf 바이너리의 버전 표시
  probe           새로운 동적 추적 지점(tracepoints) 정의
  trace           strace에서 영감을 받은 도구
```

- 프로그램을 실행하고 프로파일 정보를 수집한 후 분석한다. root 권한으로 실행

```
$ sudo  perf stat -e task-clock,context-switches,cpu-migrations,page-faults,cycles,instructions,branches  ./fibonacci
[...]
 Performance counter stats for './fibonacci':
          5210.63 msec task-clock                       #    1.000 CPUs utilized
               21      context-switches                 #    4.030 /sec
               12      cpu-migrations                   #    2.303 /sec
               63      page-faults                      #   12.091 /sec
      18888254325      cycles                           #    3.625 GHz
      34839737494      instructions                     #    1.84  insn per cycle
       7235480483      branches                         #    1.389 G/sec

      5.211290262 seconds time elapsed

      5.206803000 seconds user
      0.004091000 seconds sys
$ sudo file perf.data
perf.data: data
```

```
$ sudo  perf evlist
cycles:P
dummy:HG
$ perf annotate -s fibonacci
 ; 어셈블리 코드 수준에서 분석 화면이 실행된다.
```

- h키를 누르면 도움말이 나타남. H: hottest 명령으로이동, o: disassembler 출력모드를 변경. k: 행번호 표시 토글. p: local/global 비율 토글. l: 소스코드 파일 위치를 토글.

[그림 6] perf annotate 화면

4.7 SystemTap

- SystemTap[27]은 리눅스 커널 기반 운영 체제들을 동적으로 인스트루먼팅하기 위한 스크립트 언어이자 툴. 2005년 레드햇 엔터프라이즈 리눅스 4업데이트 2에서 처음 도입. 2009년 1.0 버전이 릴리즈됨. 스크립트 파일은 stap 명령으로 실행됨.

```
 < RedHat Enterprise 9 에서 실행 >
# yum install -y  systemtap systemtap-runtime  kernel-devel  kernel-debuginfo
# stap-prep    ; prepare system for systemtap use
# stap --help
Systemtap translator/driver (version 5.0/0.189/0.190, rpm 5.0-4.el9)
Copyright (C) 2005-2023 Red Hat, Inc. and others
This is free software; see the source for copying conditions.
tested kernel versions: 2.6.32 ... 6.6-rc1
enabled features: AVAHI BOOST_STRING_REF DYNINST BPF JAVA PYTHON3 LIBRPM LIBSQLITE3 LIBVIRT LIB
XML2 NLS NSS READLINE MONITOR_LIBS JSON_C LIBDEBUGINFOD
```

27) https://sourceware.org/systemtap/

```
Usage: stap [options] FILE                    Run script in file.
   or: stap [options] -                       Run script on stdin.
   or: stap [options] -e SCRIPT               Run given script.
   or: stap [options] -l PROBE                List matching probes.
   or: stap [options] -L PROBE                List matching probes and local variables.
   or: stap [options] --dump-probe-types      List available probe types.
   or: stap [options] --dump-probe-aliases    List available probe aliases.
   or: stap [options] --dump-functions        List available functions.
[...]
```

※ /usr/share/systemtap/examples/ 경로에 설치되는 예제들을 참고할 것

- HelloWorld 예제.[28] .stp 파일을 작성하고 stap 명령으로 실행. -k 옵션은 임시 디렉터리를 삭제하지 않고 남김. 스크립트는 내부적으로 C코드로 변환되고 컴파일되어 커널 모듈 형태로 변환됨. staprun 명령으로 컴파일된 모듈을 직접 실행할 수 있다.

```
#! /usr/bin/env stap
# helloworld.stp
probe oneshot { println("hello world") }
```

```
# dnf install kernel-devel-5.14.0-362.18.1.el9_3.x86_64
# sudo stap -k helloworld.stp
hello world
Keeping temporary directory "/tmp/stapRZxvZ9"
$ ls /tmp/stapRZxvZ9/
Makefile           stap_1556.ko     stap_1556_src.c      stapconf_export.h
modules.order      stap_1556.mod    stap_1556_src.i      stap_symbols.c
Module.symvers     stap_1556.mod.c  stap_1556_src.o      stap_symbols.o
stap_1556_aux_0.c  stap_1556.mod.o  stap_common.h
stap_1556_aux_0.o  stap_1556.o      stapconf_43d89af8ec5c6c2adefc17a13d971d23_791.h
# staprun  /tmp/stapRZxvZ9/stap_1556.ko
hello world
```

- 프로브(probe)는 스크립트에서 지정된 특정 이벤트를 탐지하기 위한 표시이며, 커널 함수 호출, 특정 시스템 호출, 사용자 공간 함수 호출 등 다양한 지점을 프로브 할수 있음.
  핸들러(Handler)는 프로브가 트리거될때 실행되는 코드임

- 특정함수가 호출된 횟수를 모니터링 하는 예제. myfunc 함수가 총 4번 실행하고 종료될때 count 값이 출력됨.

```
// myprog.c
#include <stdio.h>
#include <unistd.h>
int myfunc(int us) {
  printf("hello world \n");
  usleep(us);
}
int main(void) {
  myfunc(1000);
  myfunc(10000);
  myfunc(100000);
```

28) https://sourceware.org/systemtap/examples/#general/helloworld.stp

```
  myfunc(1000000);
  return 0;
}
$ gcc -g -o myprog myprog.c
```

```
# mytrap.stp
global count
probe process("myprog").function("myfunc").call {
  count++
}
probe process("myprog").end {
  printf("The function was called %d times\n", count)
}
```

```
# stap  mytrap.stp
The function was called 4 times
The function was called 8 times
^C
```

```
# ./myprog  ; ./myprog
hello world
hello world
[...]
hello world
```

[참고 자료]

- Shared libraries with GCC on Linux - https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
- Intel 64 and IA-32 Architectures Software Developer's Manual https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- Playing with ptrace, https://www.linuxjournal.com/article/6100?page=0,1
- DWARF https://dwarfstd.org/
- How debugger works http://www.alexonlinux.com/how-debugger-works
- Introduction to the DWARF Debugging Format, 2012년 4월 https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf
- ptrace(2) man page https://wariua.github.io/man-pages-ko/ptrace%282%29/
- GDB Internals Manual https://www.sourceware.org/gdb/documentation/
- Linux Debuginfo Formats https://www.youtube.com/watch?v=l3h7F9za_pc
- libdwarf https://www.prevanders.net/libdwarfdoc/index.html
- DWARF Extensions for Seprate Debug Information Files https://gcc.gnu.org/wiki/DebugFission
- GNU gprof를 이용한 성능 측정하기 https://blog.naver.com/hermet/175053298
- GPROF Tutorial - How to use Linux GNU GCC Profiling Tool https://www.thegeekstuff.com/2012/08/gprof-tutorial/
- gcov를 통한 커버리지 정보 알아보기 https://jayy-h.tistory.com/17
- Function Tracer Design https://www.kernel.org/doc/html/v4.18/trace/ftrace-design.html
- Linux kernel ftrace 간단한 원리 https://www.bhral.com/post/linux-kernel-ftrace-%EA%B0%84%EB%8B%A8%ED%95%9C-%EC%9B%90%EB%A6%AC
- An Introduction to Linux Tracing and its Concepts https://www.youtube.com/watch?v=1KCXvTMYYAA
- Linux Standard Base Specifications https://refspecs.linuxfoundation.org/lsb.shtml
- Perf in Netflix, Kernel Recipes 2017, Brendan Gregg https://www.youtube.com/watch?v=UVM3WX8Lq2k
- Blazing Performance with Flame Graphs https://www.slideshare.net/slideshow/blazing-performance-with-flame-graphs/28010650
- SystemTap 5.1 https://sourceware.org/systemtap/SystemTap_Beginners_Guide/index.html

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24-L02-C06-S01
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

비매품/무료
95560
9 788929 416942
ISBN 978-89-294-1694-2 (PDF)