Glibc and System Calls Documentation

Release 1.0

Rishi Agrawal <rishi.b.agrawal@gmail.com>

Dec 28, 2017

Contents

CHAPTER 1

Introduction

In this book we will see how our code interacts with the glibc library which inturn interacts with the system calls in order to get some work done from the computer.

We will go deep into the glibc code and see how it is all organized. How system calls are called from the user space programs. How arguments are passed and how are return values accessed.

We will see the code, we will see the same thing using debugger. The same thing we will see with the strace utility as well.

1.1 Acknowledgements

Most of the contents in this book is inspired from the contents in the internet, various blogs and internet. This is my first attempt at writing a document which is big enough to be called as a book.

Your suggestions and comments are very much required. You can interact with me on rishi.b.agrawal@gmail.com. Additionaly, incase you see any issue or if you would like to contribute, you can use the github repo https://github.com/rishiba/doc_syscalls for it.

CHAPTER 2

Basics of a Linux System

2.1 Introduction

In this chapter we will see some of the very basic concepts of the operating systems and programs which run on it.

- What is a computer program, how to convert the `.c` file to an `executable` and what are the steps involved.
- What are libraries? What are shared libraries and static libraries?
- What are system calls?
- What is a kernel?
- How the block diagram of the system looks like?

2.2 Programs and Compilation

Your program is a set of instructions to the computer which your computer needs to follow in order to get some work done for you.

For running a program on a Linux System these are the steps involved.

- Write the program.
- Pre-process the program. Run `gcc -E hello_world.c > pre.c`.
- Assemble the pre-processed code. Run `gcc -S pre.c`. You will get a file `pre.s`
- Compile the assembled code. Run `gcc -c pre.s`. You will get a file `pre.s`.
- Run the linker on the compiled code. `gcc pre.o`. You will get a file with name as `a.out`.

These steps are pretty simple and straight forward but there is a lot of things which go under the hood and is hidden under the `gcc` command.

2.2.1 What is gcc

- gcc is a computer program which takes another program as an input and converts it into ELF file format. ELF file format is the file format of the executable files which can be run on Linux machines.

2.2.2 Stages of compilation

- gcc has to undergo a lot of stages while compiling your code. The sequence is PREPROCESSING -> COMPILATION -> ASSEMBLING -> LINKING

Preprocessing

- This stage converts the macros in the c file to c code which can be compiled. See the file pre.e. Here the macro #include has been expanded and the whole file stdio.h has been copied in the c file.

Compilation

- Here the assembled code will be converted into the opcode of the assembly instruction.

Assembling

- This stage will convert the C programming language into the instruction set of the CPU. See the file pre.s. Here you will only see assembly instructions.

Linking

1. Here the code will be linked with the libraries present on the system. Note that printf function is not defined in your code, neither it is defined in the file stdio.h. It is just declared in the header file and it is stored in the compiled and executable format in a shared library on the system.

2.2.3 Hands-On

- Write the code

```c
#include <stdio.h>

int main() {
    printf("\n\nHello World\n");
    return 0;
}
```

- Pre-process the file

  gcc -E hello_world.c > pre.c

- Read the pre.c file to understand what has been done in the pre-processing stage.
- Assemble the pre.c file

  gcc -S pre.c - you will get a file pre.s - Read the file to see the assembled code

- Compile the `pre.s` file

  `gcc -c pre.s` - you will get a file `pre.o` - Read the file with `objdump -D pre.o` - You will get to see the full contents of the file

- Link the file
- Now this is a bit tricky as calling `ld` with the right option will be required. We will see how `gcc` does it.
- Run `gcc hello_world.c -v` to see what `gcc` does. This is very specific to the flavor of Linux because of the folder paths it has. The same command may not run on your machine. My flavor is

```
$ uname -a
Linux rishi-office 4.4.0-83-generic #106-Ubuntu SMP Mon Jun 26 17:54:43 UTC 2017 x86_
↪64 x86_64 x86_64 GNU/Linux

rishi@rishi-office:~/publications/doc_syscalls/code_system_calls/00$ cat /etc/lsb-
↪release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04.2 LTS"
```

- Here is the output of the command `gcc hello_world.c -v`. We are focusing only on the last few lines.

/usr/lib/gcc/x86_64-linux-gnu/5/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-fresolution=/tmp/cc8bF6fB.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s –sysroot=/ –build-id –eh-frame-hdr -m elf_x86_64 –hash-style=gnu –as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/5/../../.. /tmp/cchjP9PO.o -lgcc –as-needed -lgcc_s –no-as-needed -lc -lgcc –as-needed -lgcc_s –no-as-needed /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crtn.o

- You will get something like above, this is the exact step done during the linking step. `gcc` internally calls it for linking. Read more about it http://gcc.gnu.org/onlinedocs/gccint/Collect2.html
- We will replace the object file name in the above string and then run the command. New command is

ld -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-fresolution=/tmp/cc1PIEfF.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s –sysroot=/ –build-id –eh-frame-hdr -m elf_x86_64 –hash-style=gnu –as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -lgcc –as-needed -lgcc_s –no-as-needed -lc -lgcc –as-needed -lgcc_s –no-as-needed /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crtn.o pre.o -o pre.elf

- The difference is marked with `>>>>> <<<<<`

/usr/lib/gcc/x86_64-linux-gnu/5/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-fresolution=/tmp/cc8bF6fB.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s –sysroot=/ –build-id –eh-frame-hdr -m elf_x86_64 –hash-style=gnu –as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-

gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib >>>>>>!!!-L/usr/lib/gcc/x86_64-linux-gnu/5/../../.. /tmp/cchjP9PO.o <<<<<!!! -lgcc –as-needed -lgcc_s –no-as-needed -lc -lgcc –as-needed -lgcc_s –no-as-needed /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crtn.o

- Run the command after replacing the object file in the above command.
- You will get your `pre.elf` file
- Run it `./pre.elf`

```
$ ./pre.elf


Hello World
```

- Using the following `Makefile` you can do the above steps one by one and see the results for yourself.

```
C_FILE=hello_world.c
PRE_FILE=pre.c
COMP_FILE=pre.s
ASSEMBLE_FILE=pre.o
ELF_FILE=pre.elf
GCC=gcc
LINK=ld -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-fresolution=/tmp/cc1PIEfF.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib   -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crtn.o

preprocess:
        $(GCC) -E $(C_FILE) -o $(PRE_FILE)


compile: preprocess
        $(GCC) -S $(PRE_FILE) -o $(COMP_FILE)

assemble: compile
        $(GCC) -c $(COMP_FILE) -o $(ASSEMBLE_FILE)

link: assemble
        $(LINK) $(ASSEMBLE_FILE) -o $(ELF_FILE)

clean:
        rm -rf $(PRE_FILE) $(COMP_FILE) $(ASSEMBLE_FILE)
```

2.3 Libraries

A library is a zipped file of compiled code. The code is compiled and kept in a format that any other program can use the code by just linking to it. For this the program should just have the function declared in the code so that the compilation stage knows that the function's code will be linked to at a `later stage`.

In the linking phase the linker links the code by `attaching` the function call's code present in the library to the function place where function is called in the compiled code.

There are two words which I have `formatted differntly` in the above paragraph `attaching` and `later stage`.

An executable is said to be **statically linked** if the `later stage` is the last stage of the compilation and `attaching` is done in the last stage of installation.

An executable is said to be **dynamically linked** if the `later stage` is at the time of program execution and attaching is also done at the time of program execution. This is the role of `loader`.

2.3.1 Static Library

In the above section we have understood that we can compile some code and keep it as a library on the system, then use the code to `link` (read as `attaching`) to some new programs. When we `link` the code at the compile time we call it a statically compiled executable. This increases the size of the executable program as the whole library gets copied to the executable. This has the benefit that the executable becomes self sufficient and can execute on any other Linux machine.

2.3.2 Shared Library

If the compiled library is linked but not attached to the executable at the time of execution then it is called a dynamically linked executable. This in achieved by just storing the location of the function's address in the library. The executable expects the library to be present on the system where it will be executed. This is one downside of dynamic linking, where as the advantage is that the new executable will have a smaller size.

This is very useful for the libraries which are used by a lot of executable like `glibc`.

See this

```
bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,␣
↪interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,␣
↪BuildID[sha1]=eca98eeadafddff44caf37ae3d4b227132861218, stripped
```

2.4 System Calls

System calls are API's which the Kernel provides to the user space applications. The system calls pass some arguments to the kernel space and the kernel acts accordingly on the arguments

For example: `open()` system call - opens a file so that further read and write operations can be done on the file. The return value of the `open` system call is a `file descriptor` or an `error status`. Successful return value allows the user space applications to use the `file descriptor` for further reads and writes.

System calls get executed in the kernel space. Kernel space runs in an elevated privileged mode. There is a shift of the privileged modes whenever a system call is called and hence its a bad idea to call system calls without considering the time taken to switch to the elevated privileged mode.

For example - lets say that you want to copy a file. One way of copying the file is to read each character of the file and for every character read you write the character to another file. This will call two system calls for every character you read and write. As this is expensive in terms of time its a bad design.

Let us see a small demonstration of this.

```c
/*
 * In this code we will open the /etc/passwd file and copy the file 1000 times
 * to the output file. We will copy it 1000 times so that we have a good amount
 * data to run our test on.
 */

#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

#define BLOCK_SIZE 1

int main ()
{
    char *src_file = "src_file";
    char *dest_file = "copied_file.txt";

    int dest_fd, src_fd, read_byte, write_byte;
    char read_buf[BLOCK_SIZE];

    dest_fd = open (dest_file, O_WRONLY|O_CREAT, S_IRWXU|S_IRWXG|S_IROTH);

    if (dest_fd < 0) {
        perror ("\nError opening the destination file");
        exit(1);
    } else {
        fprintf (stderr, "\nSuccessfully opened the destination file..");
    }

    src_fd = open (src_file, O_RDONLY);

    if (src_fd < 0) {
        perror ("\nError opening the source file");
        exit(1);
    } else {
        fprintf (stderr, "Successfully opened the source file.");
    }


    /*
     * We will start the copy process byte by byte
     */

    while (1) {
        read_byte = read (src_fd, read_buf, BLOCK_SIZE);
        if (read_byte == 0) {
            fprintf(stdout, "Reached the EOF for src file");
            break;
        }
        write_byte = write (dest_fd, read_buf, BLOCK_SIZE);
        if (write_byte < 0) {
```

```
            perror ("Error writing file");
            exit(1);
        }
    }

    close(src_fd);
    close(dest_fd);

    return 0;
}
```

What should instead be done here is that you read a block (set of characters) and then write that block into another file. This will reduce the number of the system calls and thus increase the overall performance of the file copy program.

```
/*
 * In this code we will open the /etc/passwd file and copy the file 1000 times
 * to the output file. We will copy it 1000 times so that we have a good amount
 * data to run our test on.
 */

#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

#define BLOCK_SIZE 4096

int main ()
{
    char *src_file = "src_file";
    char *dest_file = "copied_file.txt";

    int dest_fd, src_fd, read_byte, write_byte;
    char read_buf[BLOCK_SIZE];

    dest_fd = open (dest_file, O_WRONLY|O_CREAT, S_IRWXU|S_IRWXG|S_IROTH);

    if (dest_fd < 0) {
        perror ("\nError opening the destination file");
        exit(1);
    } else {
        fprintf (stderr, "\nSuccessfully opened the destination file..");
    }

    src_fd = open (src_file, O_RDONLY);

    if (src_fd < 0) {
        perror ("\nError opening the source file");
        exit(1);
    } else {
        fprintf (stderr, "Successfully opened the source file.");
    }


    /*
     * We will start the copy process byte by byte
     */
```

```

    while (1) {
        read_byte = read (src_fd, read_buf, BLOCK_SIZE);
        if (read_byte == 0) {
            fprintf(stdout, "Reached the EOF for src file");
            break;
        }
        write_byte = write (dest_fd, read_buf, BLOCK_SIZE);
        if (write_byte < 0) {
            perror ("Error writing file");
            exit(1);
        }
    }

    close(src_fd);
    close(dest_fd);

    return 0;
}
```

```
all:
        gcc -o elf.slow_write slow_write.c -Wall
        gcc -o elf.fast_write fast_write.c -Wall

run: setup all
        time -p ./elf.slow_write
        time -p ./elf.fast_write

clean:
        rm src_file elf.slow_write elf.fast_write copied_file.txt

setup:
        for i in `seq 1 10000`; do cat /etc/passwd >> src_file; done
```

2.5 Kernel

Kernel is an important component of any Operating System. This is the only layer which interacts directly with the hardware. So in order to get any work done from your hardware you need to `ask` the kernel to do this.

This `asking` is done by `system calls`. In assembly level language this is the `syscall` instruction. When you call any system call a function in the kernel is invoked and it gets the work done. The arguments we passed are passed to the kernel and a particular function call is invoked.

For the functions any hardware interaction is needed the kernel interacts with the hardware through the device driver of the hardware.

2.6 Conclusion

In this chapter we have seen some of the important concepts and steps required to take a program from a `.c` file to an executable format on a `Linux` machine. This chapter also introduced us to the concepts of system calls and libraries.

2.7 References

- https://stackoverflow.com/questions/14163208/how-to-link-c-object-files-with-ld
- For further reading refer 1st Chapter `Getting Started` of `Beginning Linux Programming` by `Neil Matthew and Richard Stones`.

CHAPTER 3

Working with `glibc`

3.1 Introduction

This chapter deals with `glibc` library. We have earlier seen how to make our own static library, and a dynamic library.

In this chapter we will see how to work with `glibc` library.

We will Download a fresh `glibc` and will compile it on our systems. We will make some changes to the code and then link our code with this library.

3.2 Why this chapter

This chapter will help you understand the basic concepts related to using `glibc` and making changes to it. Generally you will never need to modify the code to the `glibc`, but in-case you need to make some modifications or if you need to debug a function - this section will be quite useful.

3.3 What is `glibc`

`glibc` is a library which has a lot of functions written for you so that you do not have to write the code again and again. Also it standardizes the way you should be writing your code. It wraps a lot of system specific details and all you need to know is to how to call the particular function, and what to be expected from the function and what are the return values the function will give you.

`glibc` is the `GNU Version of Standard C Library`. All the functions supported in `Standard C Library` can be found in the `glibc`.

For example: Let us say that we have to find the length of a string. Now this is quite a small code to write and we can write the whole thing ourselves, but it is a function which will be used a lot of time across a lot of products. So the library gives you an implementation of this. As the function is present in the library you can safely assume that the function will work fine because of millions of people have used it and tested it.

For the sake of understanding it better we will now go into the code of the library function and see if its similar to our code.

Also we will make some changes to the code so that it stops working incorrectly and then use it in our programs. This exercise is just a demonstration of the following.

- We can read the code of `glibc`.
- We can compile the code of `glibc` ourselves and use the newly compiled library.
- We can change the code of `glibc`.
- We can use the changed code of `glibc`.

3.4 Download and extract `glibc`

The source code of `glibc` is available at `https://ftp.gnu.org/gnu/libc/`. You can sort the list using `Last Modified` to get the latest tar package.

From the page I got the link as `https://ftp.gnu.org/gnu/libc/glibc-2.24.tar.xz`.

- Let us download this source, see the following snippet for the exact commands.

```
$ wget https://ftp.gnu.org/gnu/libc/glibc-2.24.tar.xz
--2017-01-29 07:50:02--  https://ftp.gnu.org/gnu/libc/glibc-2.24.tar.xz
Resolving ftp.gnu.org (ftp.gnu.org)... 208.118.235.20, 2001:4830:134:3::b
Connecting to ftp.gnu.org (ftp.gnu.org)|208.118.235.20|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13554048 (13M) [application/x-tar]
Saving to: 'glibc-2.24.tar.xz'

glibc-2.24.tar.xz                    100%[==>]  12.93M   709KB/s    in 21s

2017-01-29 07:50:26 (622 KB/s) - 'glibc-2.24.tar.xz' saved [13554048/13554048]
```

3.4.1 Extract the code

- The downloaded code is a compressed tar file. We need to extract it.

```
rishi@rishi-VirtualBox:~$ tar -xf glibc-2.24.tar.xz
```

- This creates a directory names `glibc-2.24` in the folder.

3.5 Walkthrough `glibc`

- Here is a listing of all the directories inside the extracted `glibc` directory. You can see the directories where the code related to `math strings stdlib` are present.

```
rishi@rishi-VirtualBox:~$ cd glibc-2.24/
rishi@rishi-VirtualBox:~/glibc-2.24$ ls
abi-tags         ChangeLog.3                        ChangeLog.old-ports-mips
aclocal.m4       ChangeLog.4                        ChangeLog.old-ports-powerpc
argp             ChangeLog.5                        ChangeLog.old-ports-tile
assert           ChangeLog.6                        config.h.in
benchtests       ChangeLog.7                        config.make.in
```

```
bits               ChangeLog.8                        configure
BUGS               ChangeLog.9                        configure.ac
catgets            ChangeLog.old-ports                conform
ChangeLog          ChangeLog.old-ports-aarch64        CONFORMANCE
ChangeLog.1        ChangeLog.old-ports-aix            COPYING
ChangeLog.10       ChangeLog.old-ports-alpha          COPYING.LIB
ChangeLog.11       ChangeLog.old-ports-am33           cppflags-iterator.mk
ChangeLog.12       ChangeLog.old-ports-arm            crypt
ChangeLog.13       ChangeLog.old-ports-cris           csu
ChangeLog.14       ChangeLog.old-ports-hppa           ctype
ChangeLog.15       ChangeLog.old-ports-ia64           debug
ChangeLog.16       ChangeLog.old-ports-linux-generic  dirent
ChangeLog.17       ChangeLog.old-ports-m68k           dlfcn
ChangeLog.2        ChangeLog.old-ports-microblaze     elf
extra-lib.mk       LICENSES     nscd                  stdio-common
extra-modules.mk   locale       nss                   stdlib
gen-locales.mk     localedata   o-iterator.mk         streams
gmon               login        po                    string
gnulib             mach         posix                 sunrpc
grp                Makeconfig   PROJECTS              sysdeps
gshadow            Makefile     pwd                   sysvipc
hesiod             Makefile.in  README                termios
hurd               Makerules    resolv          test-skeleton
iconv              malloc       resource        time
iconvdata          manual       rt              timezone
include            math         Rules           version.h
inet               mathvec      scripts         wcsmbs
INSTALL            misc         setjmp          wctype
intl               NAMESPACE    shadow          WUR-REPORT
io                 NEWS         shlib-versions
libc-abis          nis          signal
libidn             nptl         socket
libio              nptl_db      soft-fp
```

- Some string related code is here

```
rishi@rishi-VirtualBox:~/glibc-2.24$ ls string/str*
string/stratcliff.c    string/strcmp.c     string/strerror_l.c
string/strcasecmp.c    string/strcoll.c    string/strfry.c
string/strcasecmp_l.c  string/strcoll_l.c  string/string.h
string/strcasestr.c    string/strcpy.c     string/string-inlines.c
string/strcat.c        string/strcspn.c    string/strings.h
string/strchr.c        string/strdup.c     string/strlen.c
string/strchrnul.c     string/strerror.c   string/strncase.c
string/strncase_l.c  string/strrchr.c    string/str-two-way.h
string/strncat.c     string/strsep.c     string/strverscmp.c
string/strncmp.c     string/strsignal.c  string/strxfrm.c
string/strncpy.c     string/strspn.c     string/strxfrm_l.c
string/strndup.c     string/strstr.c
string/strnlen.c     string/strtok.c
string/strpbrk.c     string/strtok_r.c
```

- Some math related code is here

```
$ ls math/w_*

math/w_acos.c    math/w_hypotl.c          math/w_log1pl.c
math/w_acosf.c   math/w_ilogb.c           math/w_log2.c
```

```
math/w_acosh.c   math/w_ilogbf.c          math/w_log2f.c
math/w_acoshf.c  math/w_ilogbl.c          math/w_log2l.c
math/w_acoshl.c  math/w_j0.c              math/w_log.c
math/w_acosl.c   math/w_j0f.c             math/w_logf.c
math/w_asin.c    math/w_j0l.c             math/w_logl.c
math/w_asinf.c   math/w_j1.c              math/w_pow.c
math/w_asinl.c   math/w_j1f.c             math/w_powf.c
math/w_atan2.c   math/w_j1l.c             math/w_powl.c
math/w_atan2f.c  math/w_jn.c              math/w_remainder.c
math/w_atan2l.c  math/w_jnf.c             math/w_remainderf.c
math/w_atanh.c   math/w_jnl.c             math/w_remainderl.c
math/w_atanhf.c  math/w_lgamma.c          math/w_scalb.c
math/w_atanhl.c  math/w_lgamma_compat.c   math/w_scalbf.c
math/w_cosh.c    math/w_lgamma_compatf.c  math/w_scalbl.c
math/w_coshf.c   math/w_lgamma_compatl.c  math/w_scalbln.c
math/w_coshl.c   math/w_lgammaf.c         math/w_scalblnf.c
math/w_exp10.c   math/w_lgammaf_main.c    math/w_scalblnl.c
math/w_exp10f.c  math/w_lgammaf_r.c       math/w_sinh.c
math/w_exp10l.c  math/w_lgammal.c         math/w_sinhf.c
math/w_exp2.c    math/w_lgammal_main.c    math/w_sinhl.c
math/w_exp2f.c   math/w_lgammal_r.c       math/w_sqrt.c
math/w_exp2l.c   math/w_lgamma_main.c     math/w_sqrtf.c
math/w_expl.c    math/w_lgamma_r.c        math/w_sqrtl.c
math/w_fmod.c    math/w_log10.c           math/w_tgamma.c
math/w_fmodf.c   math/w_log10f.c          math/w_tgammaf.c
math/w_fmodl.c   math/w_log10l.c          math/w_tgammal.c
math/w_hypot.c   math/w_log1p.c
math/w_hypotf.c  math/w_log1pf.c
```

- The header files for the library is here.

```
$ ls include/
aio.h          gconv.h             net                stackinfo.h
aliases.h      getopt.h            netdb.h            stap-probe.h
alloca.h       getopt_int.h        netgroup.h         stdc-predef.h
argp.h         glob.h              netinet            stdio_ext.h
argz.h         gmp.h               nl_types.h         stdio.h
arpa           gnu                 nss.h              stdlib.h
assert.h       gnu-versions.h      nsswitch.h         string.h
atomic.h       grp.h               obstack.h          strings.h
bits           grp-merge.h         poll.h             stropts.h
byteswap.h     gshadow.h           printf.h           stubs-prologue.h
caller.h       iconv.h             programs           sys
complex.h      ifaddrs.h           protocols          syscall.h
cpio.h         ifunc-impl-list.h   pthread.h          sysexits.h
ctype.h        inline-hashtab.h    pty.h              syslog.h
des.h          langinfo.h          pwd.h              tar.h
dirent.h       libc-internal.h     regex.h            termios.h
dlfcn.h        libc-symbols.h      resolv.h           tgmath.h
elf.h          libgen.h            rounding-mode.h    time.h
endian.h       libintl.h           rpc                ttyent.h
envz.h         libio.h             rpcsvc             uchar.h
err.h          limits.h            sched.h            ucontext.h
errno.h        link.h              scratch_buffer.h   ulimit.h
error.h        list.h              search.h           unistd.h
execinfo.h     locale.h            set-hooks.h        utime.h
fcntl.h        malloc.h            setjmp.h           utmp.h
features.h     math.h              sgtty.h            values.h
```

```
fenv.h          mcheck.h          shadow.h          wchar.h
fmtmsg.h        memory.h          shlib-compat.h    wctype.h
fnmatch.h       mntent.h          signal.h          wordexp.h
fpu_control.h   monetary.h        spawn.h           xlocale.h
ftw.h           mqueue.h          stab.h
```

3.6 Reading some functions of glibc

3.6.1 Reading strlen

- Let us see the code of `strcmp.c`. The file is present in the extracted `glibc` directory.

```c
/* Copyright (C) 1991-2016 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <http://www.gnu.org/licenses/>.  */

#include <string.h>

#undef strcmp

#ifndef STRCMP
# define STRCMP strcmp
#endif

/* Compare S1 and S2, returning less than, equal to or
   greater than zero if S1 is lexicographically less than,
   equal to or greater than S2.  */
int
STRCMP (const char *p1, const char *p2)
{
  const unsigned char *s1 = (const unsigned char *) p1;
  const unsigned char *s2 = (const unsigned char *) p2;
  unsigned char c1, c2;

  do
    {
      c1 = (unsigned char) *s1++;
      c2 = (unsigned char) *s2++;
      if (c1 == '\0')
        return c1 - c2;
    }
  while (c1 == c2);
```

```

  return c1 - c2;
}
libc_hidden_builtin_def (strcmp)
```

- The code is pretty simple to understand. It iterates through the string till the time it finds both the characters equal.

  What I want to emphasize is that the `glibc` is just a collect of c functions, written in c files, packaged and compiled, and we can also make similar functions and libraries and publish.

3.6.2 Walkthrough `div`

- Let us now see the code of `stdlib/div.c`. I have again picked a very simple function which will enable you to understand that the functions and functionality provided by the `glibc` is just a simple function which we write almost daily in our code.

3.7 Compiling and installing `glibc`

Generally compiling and installing code on Linux system involves the following stages

1. Configuring - running `configure` with right options.
2. Compiling - running `make` with right options.
3. Install - running `make install`.

We will also go through the same steps and complete compilation and installation of the new library.

3.7.1 Configuring `glibc`

We will get into the `glibc-2.24` source directory and run the `configure` script. I have intentionally shown the mistakes which happened so that you also understand the small things which needs to be taken care while configuring and compiling.

```
rishi@rishi-VirtualBox:~/glibc-2.24$ ./configure
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for gcc... gcc
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for readelf... readelf
checking for g++... g++
checking whether we are using the GNU C++ compiler... yes
checking whether g++ accepts -g... yes
checking whether g++ can link programs... yes
configure: error: you must configure in a separate build directory
```

- We got an error that we should use a separate directory for running `configure`

```
rishi@rishi-VirtualBox:~/glibc-2.24$ mkdir ../build_glibc

rishi@rishi-VirtualBox:~/glibc-2.24$ cd ../build_glibc/
```

- Let us now run the configure command.

```
rishi@rishi-VirtualBox:~/build_glibc$ ../glibc-2.24/configure
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for gcc... gcc
checking for suffix of object files... o
checking version of sed... 4.2.2, ok
checking for gawk... no

>>>>>>>>>>>>>>>>>SNIP<<<<<<<<<<<<<<<<<<<<<<

checking if gcc is sufficient to build libc... yes
checking for nm... nm
configure: error:
*** These critical programs are missing or too old: gawk
*** Check the INSTALL file for required versions.
```

- The configure step gave errors - let us install `gawk` now.

```
rishi@rishi-VirtualBox:~/build_glibc$ sudo apt-get install gawk
[sudo] password for rishi:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
libsigsegv2
Suggested packages:
gawk-doc
The following NEW packages will be installed:
gawk libsigsegv2

>>>>>>>>>>>>>SNIP<<<<<<<<<<<<<<<

Setting up gawk (1:4.1.3+dfsg-0.1) ...
```

- Check if the command is present.

```
rishi@rishi-office:~/mydev/publications/system_calls$ which gawk
/usr/bin/gawk
```

- Let us run configure again

```
rishi@rishi-VirtualBox:~/build_glibc$ ../glibc-2.24/configure
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for gcc... gcc
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes

>>>>>>>>>>SNIP<<<<<<<<<<<<<<<<<<<<<<

running configure fragment for sysdeps/unix/sysv/linux/x86_64
running configure fragment for sysdeps/unix/sysv/linux
checking installed Linux kernel header files... 3.2.0 or later
checking for kernel header at least 2.6.32... ok
*** On GNU/Linux systems the GNU C Library should not be installed into
*** /usr/local since this might make your system totally unusable.
*** We strongly advise to use a different prefix.  For details read the FAQ.
```

```
*** If you really mean to do this, run configure again using the extra
*** parameter `--disable-sanity-checks`.
```

- Configure does not want to overwrite the default library and hence we need to give another directory to install the library.
- Let us make a directory and run the configure script.

```
rishi@rishi-VirtualBox:~/build_glibc$ mkdir ../install_glibc
rishi@rishi-VirtualBox:~/build_glibc$ ../glibc-2.24/configure --prefix=/home/rishi/
↪install_glibc/
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for gcc... gcc
checking for suffix of object files... o
configure: creating ./config.status

>>>>>>>SNIP<<<<<<<<<<<<

config.status: creating config.make
config.status: creating Makefile
config.status: creating config.h
config.status: executing default commands
```

- Configure completed

```
rishi@rishi-VirtualBox:~/build_glibc$ ls
bits  config.h  config.log  config.make  config.status  Makefile
```

3.7.2 Compiling `glibc`

- Let us run the `make` command now. Go to the `build_glibc` directory and run the `make` command.

```
rishi@rishi-VirtualBox:~/build_glibc$ make -j 16
make -r PARALLELMFLAGS="" -C ../glibc-2.24 objdir=`pwd` all
make[1]: Entering directory '/home/rishi/glibc-2.24'
LC_ALL=C gawk -f scripts/sysd-rules.awk > /home/rishi/build_glibc/sysd-rulesT \


rishi@rishi-VirtualBox:~/build_glibc$ ls
bits  config.h  config.log  config.make  config.status  Makefile
rishi@rishi-VirtualBox:~/build_glibc$
rishi@rishi-VirtualBox:~/build_glibc$
rishi@rishi-VirtualBox:~/build_glibc$


rishi@rishi-VirtualBox:~/build_glibc$ make -j 16
make -r PARALLELMFLAGS="" -C ../glibc-2.24 objdir=`pwd` all
make[1]: Entering directory '/home/rishi/glibc-2.24'
LC_ALL=C gawk -f scripts/sysd-rules.awk > /home/rishi/build_glibc/sysd-rulesT \

    >>>>>>>>>>>>>>>>>>>>>SNIP<<<<<<<<<<<<<<<<<<<
gcc -nostdlib -nostartfiles -o /home/rishi/build_glibc/elf/pldd    -Wl,-z,combreloc -
↪Wl,-z,relro -Wl,--hash-style=both /home/rishi/build_glibc/csu/crt1.o /home/rishi/
↪build_glibc/csu/crti.o `gcc  --print-file-name=crtbegin.o` /home/rishi/build_glibc/
↪elf/pldd.o /home/rishi/build_glibc/elf/xmalloc.o  -Wl,-dynamic-linker=/home/rishi/
↪install_glibc/lib/ld-linux-x86-64.so.2 -Wl,-rpath-link=/home/rishi/build_glibc:/
↪home/rishi/build_glibc/math:/home/rishi/build_glibc/elf:/home/rishi/build_glibc/
↪dlfcn:/home/rishi/build_glibc/nss:/home/rishi/build_glibc/nis:/home/rishi/build_
↪glibc/rt:/home/rishi/build_glibc/resolv:/home/rishi/build_glibc/crypt:/home/rishi/
↪build_glibc/mathvec:/home/rishi/build_glibc/nptl /home/rishi/build_glibc/libc.so.6 /
↪home/rishi/build_glibc/libc_nonshared.a -Wl,--as-needed /home/rishi/build_glibc/elf/
↪ld.so -Wl,--no-as-needed -lgcc  `gcc  --print-file-name=crtend.o` /home/rishi/build_
↪glibc/csu/crtn.o
```

```
make[2]: Leaving directory '/home/rishi/glibc-2.24/elf'
make[1]: Leaving directory '/home/rishi/glibc-2.24'
```

- Make runs successfully.
- Let us check the `install_glibc` directory. It has nothing in it.

```
$ ls ../install_glibc/
```

- Let us run the `make install` command. This needs to be done in the `build_glibc` directory.

```
$ make install
LC_ALL=C; export LC_ALL; \
make -r PARALLELMFLAGS="" -C ../glibc-2.24 objdir=`pwd` install
make[1]: Entering directory '/home/rishi/glibc-2.24'
make  subdir=csu -C csu ..=../ subdir_lib
make[2]: Entering directory '/home/rishi/glibc-2.24/csu'
make[2]: Leaving directory '/home/rishi/glibc-

>>>>>>>>>>>SNIP<<<<<<<<<<<<<<<

-f /home/rishi/build_glibc/elf/symlink.list
test ! -x /home/rishi/build_glibc/elf/ldconfig || LC_ALL=C \
/home/rishi/build_glibc/elf/ldconfig  \
/home/rishi/install_glibc/lib /home/rishi/install_glibc/lib
/home/rishi/build_glibc/elf/ldconfig: Warning: ignoring configuration file that␣
↪cannot be opened: /home/rishi/install_glibc/etc/ld.so.conf: No such file or␣
↪directory
make[1]: Leaving directory '/home/rishi/glibc-2.24'
```

3.7.3 Installing `glibc`

- Let us now check the `install_glibc` directory. It has the required files of the new compiled library.

```
rishi@rishi-VirtualBox:~/build_glibc$ ls ../install_glibc/
bin  etc  include  lib  libexec  sbin
```

3.8 Using new `glibc`

Let us now use the above library to link and run our code. We will add a new function to the `glibc`, change the behavior of a function in `glibc` and use the new function and call the changed function.

This will give us a good understanding of how to compile and link with the new library.

Here is the code for adding some changes to the `glibc` code. See the file `glibc-2.24/stdlib/div.c` and `glibc-2.24/include/stdlib.h`.

Here is the diff

3.8.1 `glibc-2.24/stdlib/div.c`

- Here we have added a function `mydiv` which just returns -1 on invocation and have changed the way the function div behaves. Now when we will pass 99 and 99 to div it will return 100 and 100. Read the default behavior in the man pages.

- Here is the declaration of the new function.

3.8.2 `glibc-2.24/stdlib/stdlib.h`

- Here is the code which calls the functions.

Listing 3.1: code_system_calls/03/div/test_div.c

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


int main () {

    div_t result = div(99, 99);
    int x = mydiv();

    printf ("\n\nQuotient %d Remainder %d", result.quot, result.rem);
    printf ("\nValue returned by mydiv is %d\n\n", x);
    return 0;
}
```

- Here is the `Makefile` which will be used to compile the program.

```
TARGET = test_div
OBJ = $(TARGET).o
SRC = $(TARGET).c
CC = gcc
CFLAGS = -g
LDFLAGS = -nostdlib -nostartfiles -static
GLIBCDIR = /home/rishi/glibc/install_glibc/lib/
INCDIR = /home/rishi/glibc/install_glibc/include
STARTFILES = $(GLIBCDIR)/crt1.o $(GLIBCDIR)/crti.o `gcc --print-file-name=crtbegin.o`
ENDFILES = `gcc --print-file-name=crtend.o` $(GLIBCDIR)/crtn.o
LIBGROUP = -Wl,--start-group $(GLIBCDIR)/libc.a -lgcc -lgcc_eh -Wl,--end-group

$(TARGET): $(OBJ)
	$(CC) $(LDFLAGS) -o $@ $(STARTFILES) $^ $(LIBGROUP) $(ENDFILES)

$(OBJ): $(SRC)
	$(CC) $(CFLAGS) -c $^ -I `gcc --print-file-name=include` -I $(INCDIR)

clean:
	rm -f *.o *.~ $(TARGET)
	rm test.c.*
	rm a.out


# https://stackoverflow.com/questions/10763394/how-to-build-a-c-program-using-a-
↪custom-version-of-glibc-and-static-linking/10772056#10772056
```

- Run the `make` command.

```
$ make
gcc -g -c test_div.c -I `gcc --print-file-name=include` -I /home/rishi/glibc/install_
↪glibc/include
gcc -nostdlib -nostartfiles -static -o test_div /home/rishi/glibc/install_glibc/lib//
↪crt1.o /home/rishi/glibc/install_glibc/lib//crti.o `gcc --print-file-name=crtbegin.
↪o` test_div.o -Wl,--start-group /home/rishi/glibc/install_glibc/lib//libc.a -lgcc -
↪lgcc_eh -Wl,--end-group `gcc --print-file-name=crtend.o` /home/rishi/glibc/install_
↪glibc/lib//crtn.o
```

- Run the statically linked code

```
$ ./test_div

Values are 99 and 99
Calling mydiv function

Quotient 100 Remainder 100
Value returned by mydiv is -1
```

- See the size of the statically linked code. The huge size is due to static linking. In case of dynamically linked code the size will be very less.

```
$ ls -lah test_div
-rwxrwxr-x 1 rishi rishi 3.3M Jul 24 12:21 test_div
```

- Using `file` command see the `statically linked` flag in the file.

```
rishi@rishi-office:~/publications/doc_syscalls/code_system_calls/03/div$ file test_div
test_div: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked,
↪ for GNU/Linux 2.6.32, BuildID[sha1]=ad293fdf108078a42635ed6f91ad317ad93ec9d2, not␣
↪stripped
```

- Check the file type of the executable.

```
rishi@rishi-VirtualBox:~/test_code$ file static-test
static-test: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically␣
↪linked, for GNU/Linux 2.6.32,␣
↪BuildID[sha1]=866f4fe367915159ae62cc80a0ae614059d67153, not stripped
```

3.9 Conclusion

In this chapter we have seen pretty important things with respect to using `glibc`. We have seen where to find `glibc`, how to download, extract, make changes and compile the `glibc` library in your system.

Doing all the steps hands-on will enable you understand the whole workflow more clearly and will thus improve your understanding of systems.

CHAPTER 4

System Calls On x86_64 from User Space

There are three parts to calling a system call like any function call.

- Setting up the arguments to be passed to the kernel space. Here we gather the right arguments to pass to the function. Based on these argument the kernel will do the required work for you.
- Call the system call using the `syscall` assembly instruction. This is exact place where the programs `hand-over` the work to the kernel. The process then waits for the system call to return. In asynchronous system calls the process will get a return value to indicate that the task has been submitted correctly and kernel is doing the job.
- Get back the return value. This is the return status of the work done by the kernel. Using this the kernel notifies the process about the task done. There is also a global error number variable which stores the error (if any) encountered by the kernel.

In the sections below we will see each of them in detail.

4.1 Setting Up Arguements

Note: The following text is copied verbatim from the document **System V Application Binary Interface AMD64 Architecture Processor 57 Supplement Draft Version 0.99.6**, Section **AMD64 Linux Kernel Conventions**. The copyright belongs to the original owners of the document.

```
Calling Conventions

The Linux AMD64 kernel uses internally the same calling conventions as user-
level applications (see section 3.2.3 for details). User-level applications that like
to call system calls should use the functions from the C library. The interface
between the C library and the Linux kernel is the same as for the user-level appli-
cations with the following differences:

1. User-level applications use as integer registers for passing the sequence
```

```
%rdi, %rsi, %rdx, %rcx, %r8 and %r9. The kernel interface uses %rdi,
%rsi, %rdx, %r10, %r8 and %r9.
2. A system-call is done via the syscall instruction. The kernel destroys
registers %rcx and %r11.
3. The number of the syscall has to be passed in register %rax.
4. System-calls are limited to six arguments, no argument is passed directly on
the stack.
5. Returning from the syscall, register %rax contains the result of the
system-call. A value in the range between -4095 and -1 indicates an error,
it is -errno.
6. Only values of class INTEGER or class MEMORY are passed to the kernel.
```

See the `System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6`. Section `AMD64 Linux Kernel Conventions` for the details.

4.1.1 Reiterating The Above Again

Hence when we have called any function in user space we will have the following state of the registers when we are in the called function.

Table 4.1: "Arguments Passing In Linux"

Register	Argument User Space	Argument Kernel Space
%rax	Not Used	System Call Number
%rdi	Arguement 1	Arguement 1
%rsi	Arguement 2	Arguement 2
%rdx	Arguement 3	Arguement 3
%r10	Not Used	Arguement 4
%r8	Arguement 5	Arguement 5
%r9	Arguement 6	Arguement 6
%rcx	Arguement 4	Destroyed
%r11	Not Used	Destroyed

Note: This table summarizes the differences when a function call is made in the user space, and when a system call is made. This will be more clear in coming texts. Right now make a note of it

4.1.2 Passing arguments

- Arguments are passed in the registers. The called function then uses the register to get the arguments.
- The arguments are passed in the following sequence `%rdi, %rsi, %rdx, %r10, %r8 and %r9.`
- Number of arguments are limited to `six`, no arguments will be passed on the stack.
- Only values of class `INTEGER` or class `MEMORY` are passed to the kernel.
- Class `INTEGER` This class consists of integral types that fit into one of the general purpose registers.
- Class `MEMORY` This class consists of types that will be passed and returned in memory via the stack. These will mostly be strings or memory buffer. For example in `write()` system call, the first parameter is `fd` which is of

class `INTEGER` while the second argument is the `buffer` which has the data to be written in the file, the class will be `MEMORY` over here. The third parameter which is the count - again has the class as `INTEGER`.

Note: The above information is sourced from AMD64 Architecture Processor Supplement Draft Version 0.99.6

4.2 Calling the System Call

- A system-call is done via the `syscall` assembly instruction. The kernel destroys registers `%rcx` and `%r11`.
- The number of the system call has to be passed in register `%rax`.

4.3 Retrieving the Return Value

- Returning from the `syscall`, register `%rax` contains the result of the system-call. A value in the range between `-4095` and `-1` indicates an error, it is `-errno`.

CHAPTER 5

Setting Up Arguments

5.1 Introduction

In the previous chapter *Setting Up Arguements* section we have seen the theory part related to passing arguments to the system call interface of the kernel. Now we will do a `hands-on` exercise related to it.

We will see how the above concepts are being implemented in `glibc` code. We will see it in two ways

1. We will walk through `open` system call in `glibc` library. This should show us how the registers are filled with the right value and then assembly instruction `syscall` is been called.
2. We will add a break point in one system call and see the state of the registers.

5.2 Walk through `open` system call in `glibc`

- All the above theory of passing the arguments should match with the code which is written in `glibc`.
- We will now read the code in the `glibc` to find out if the theory matches what is written in the code.
- Now the question is `open` system call - how will it turn to a `syscall` instruction with the right values in the registers.
- Now we need to find out what happens to the `open` system call when compiled. For this we will write a small code and compile it statically. Using `objdump` we will be able to see the actual function calls.
- Use the following file for the purpose.

```c
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>


int main ()
```

```
{
    int fd = open ("/etc/passwd", O_RDONLY);
    close(fd);
    return 0;
}
```

- To compile use the following command `gcc open.c --static -g -o elf.open`
- To get the `objdump` output use the command `objdump elf.open -D > objdump.txt`
- File where `SYS_open` maps to `__NR_open` : `/usr/include/x86_64-linux-gnu/bits/syscall.h`
- File where `__NR_open` maps to actual number `2` : `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`
- From the `objdump` we saw that `__libc_open` was called. This called `__open_nocancel` and it had a `syscall` instruction.
- See the `objdump.txt`, search for `__open_nocancel`.

```
0000000000433e09 <_open_nocancel>:
433e09:   b8 02 00 00 00          mov    $0x2,%eax
433e0e:   0f 05                   syscall
433e10:   48 3d 01 f0 ff ff       cmp    $0xfffffffffffff001,%rax
433e16:   0f 83 f4 46 00 00       jae    438510 <__syscall_error>
433e1c:   c3                      retq
433e1d:   48 83 ec 08             sub    $0x8,%rsp
433e21:   e8 ca 2f 00 00          callq  436df0 <__libc_enable_asynccancel>
433e26:   48 89 04 24             mov    %rax,(%rsp)
433e2a:   b8 02 00 00 00          mov    $0x2,%eax
433e2f:   0f 05                   syscall
433e31:   48 8b 3c 24             mov    (%rsp),%rdi
433e35:   48 89 c2                mov    %rax,%rdx
433e38:   e8 13 30 00 00          callq  436e50 <__libc_disable_asynccancel>
433e3d:   48 89 d0                mov    %rdx,%rax
433e40:   48 83 c4 08             add    $0x8,%rsp
433e44:   48 3d 01 f0 ff ff       cmp    $0xfffffffffffff001,%rax
433e4a:   0f 83 c0 46 00 00       jae    438510 <__syscall_error>
433e50:   c3                      retq
433e51:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
433e58:   00 00 00
433e5b:   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

- Now, when in `glibc-2.3` dir I started finding the code for the function `__open_nocancel` I found this
- File is `sysdeps/unix/sysv/linux/generic/open.c`

```
int __open_nocancel (const char *file, int oflag, ...)
{
    int mode = 0;

    if (__OPEN_NEEDS_MODE (oflag))
    {
        va_list arg;
        va_start (arg, oflag);
        mode = va_arg (arg, int);
        va_end (arg);
    }
```

```
    return INLINE_SYSCALL (openat, 4, AT_FDCWD, file, oflag, mode);
}
```

- So INLINE_SYSCALL is being called by this function. This is defined in the file `glibc-2.3/sysdeps/unix/sysv/linux/x86_64/sysdep.h`

```
# define INLINE_SYSCALL(name, nr, args...) \
    ({                                          \
     unsigned long int resultvar = INTERNAL_SYSCALL (name, , nr, args);         \
     if (__glibc_unlikely (INTERNAL_SYSCALL_ERROR_P (resultvar, )))         \
     {                                        \
     __set_errno (INTERNAL_SYSCALL_ERRNO (resultvar, ));            \
     resultvar = (unsigned long int) -1;                 \
     }                                        \
     (long int) resultvar; })
```

- Thus it calls `INTERNAL_SYSCALL` which is defined as

```
# define INTERNAL_SYSCALL(name, err, nr, args...) \
  INTERNAL_SYSCALL_NCS (__NR_##name, err, nr, ##args)
```

- Now let us see the `INTERNAL_SYSCALL_NCS` in the file `./sysdeps/unix/sysv/linux/x86_64/sysdep.h` here see the macro `INTERNAL_SYSCALL_NCS`. **This is the exact macro which is calling the ‘‘syscall‘‘ assembly instruction.** You can see the `asm` instructions in the code.

```
# define INTERNAL_SYSCALL_NCS(name, err, nr, args...) \
    ({                                          \
     unsigned long int resultvar;                     \
     LOAD_ARGS_##nr (args)                        \
     LOAD_REGS_##nr                           \
     asm volatile (                           \
         "syscall\n\t"                            \
         : "=a" (resultvar)                        \
         : "0" (name) ASM_ARGS_##nr : "memory", REGISTERS_CLOBBERED_BY_SYSCALL);    \
     (long int) resultvar; })
```

- Thus here we enter the kernel using the `syscall` assembly instruction.

5.3 Check Arguements Using `gdb`

In the above example we saw how the code calls the `syscall` instruction to enter the kernel and call the required functionality. Write the following code and compile it with `gcc -g filename.c`

`-g` flag adds the debugging information to the executable.

```
#include <fcntl.h>
#include <string.h>

int main ()
{
    char filename[] = "non_existent_file";
    int fd;
    fd = open (filename, O_CREAT|O_WRONLY);

    fd = write (fd, filename, strlen(filename));
    close (fd);
```

```
    unlink (filename);
    return 0;
}
```

- Once done, run the code in the debugger `gdb ./a.out`
- Set the breakpoint in the call on write `break write`
- According to the calling conventions the register `$rdi` should have the file descriptor. `$rdi` should have the string's address and the `$rdx` should have the length of the string.
- Using `print` command will confirm these values.

```
(gdb) b write
Breakpoint 1 at 0x400560
(gdb) r
Starting program: /home/rishi/mydev/books/crash_book/code_system_calls/01/aaa/a.out

Breakpoint 1, write () at ../sysdeps/unix/syscall-template.S:81
81  ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) print $rdi
$1 = 3
(gdb) print (char *) $rsi
$2 = 0x7fffffffdeb0 "non_existent_file"
(gdb) print $rdx
$3 = 17
(gdb)
```

CHAPTER 6

Calling System Calls

There are two ways system calls are being called in the user space. Both of them will eventually call the `syscall` instruction but `glibc` provides a wrapper around that instruction using a function call.

- `glibc` library call - this moves the arguments to the right registers before calling the `syscall` instruction.
- `syscall` assembly instruction - to actually hand over the work to the kernel.

6.1 Glibc `syscall()` interface

- There is a library function in `glibc` named as `syscall`, you can read about it in the man pages by the command `man 2 syscall`.
- We already have the code of `glibc` with us.
- See the function in the file `glibc-2.23/sysdeps/unix/sysv/linux/x86_64/syscall.S`
- On reading the code you will see that the function is moving the argument values to the registers and then calling the assembly instruction `syscall`.
- As `syscall` here is a user space `glibc` library function, first the arguments will be in the registers used for calling user space functions. Once this is done, as the system call is being called, the arguments will be used into the registers where the kernel wishes to find the arguments. See *Reiterating The Above Again*
- Code for `syscall(2)` library function. File is `glibc-2.24/sysdeps/unix/sysv/linux/x86_64/syscall.S`

Note: Remember the note above. As `syscall` is a function which we called in user space, the registers are different. We now need to pick and place the registers in a way that the system call understands it. This is shown in the code below.

```
/* Copyright (C) 2001-2016 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

```

```
   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <http://www.gnu.org/licenses/>.  */

#include <sysdep.h>

/* Please consult the file sysdeps/unix/sysv/linux/x86-64/sysdep.h for
   more information about the value -4095 used below.  */

/* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4, arg5, arg6)
   We need to do some arg shifting, the syscall_number will be in
   rax.  */


        .text
ENTRY (syscall)
        movq %rdi, %rax                 /* Syscall number -> rax.  */
        movq %rsi, %rdi                 /* shift arg1 - arg5.  */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 is on the stack.  */
        syscall                         /* Do the system call.  */
        cmpq $-4095, %rax        /* Check %rax for error.  */
        jae SYSCALL_ERROR_LABEL         /* Jump to error handler if error.  */
        ret                         /* Return to caller.  */

PSEUDO_END (syscall)
```

6.2 `syscall` assembly instruction

We know now that for calling a system call we just need to set the right arguments in the register and then call the `syscall` instruction.

Register `%rax` needs the `system call number`. So where are the `system call numbers` defined? Here we can see the `glibc` code to see the mapping of the number and the system call. Or you can see this in a header file in the system's include directory.

Let us see a excerpt from the file `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`

```
#define __NR_read   0
#define __NR_write  1
#define __NR_open   2
#define __NR_close  3
#define __NR_stat   4
```

Here you can see that the system calls have numbers associated with them.

6.3 Difference between `syscall()` glibc interface and `syscall` assembly instruction

In this section we will write some data to the `STDOUT` (terminal) using three methods.

- First we will issue a `write()` system call.
- Second we will use the `syscall()` function in glibc.
- Third we will write assembly code and call the `syscall` instruction.

This will help us understand `system calls` in more detail.

Now armed with the knowledge of how to call system calls let us write some assembly code where we call a system call.

6.3.1 `write()` system call

We will start by exploring the `write` system call a bit. In the following code we will write `hello world` on the screen. We will not use `printf` for this, rather we will use `1` (the standard descriptor for writing to the terminal) and `write` system call for it.

We need to do this so that we understand our assembly level program a bit better.

Listing 6.1: code_system_calls/07/write.c

```c
#include <fcntl.h>
#include <unistd.h>

int main ()
{
    write (1, "Hello World", 11);
    return 0;
}
```

You should go through the assembly code of the C file. Use command `gcc -S filename.c` This will generate the assembly file with `.s` extension. If you go through the assembly code you will see a call to `write` function. This function is defined in the `glibc`.

6.3.2 `syscall()` function

Now we will do the same using the `syscall` interface which the `glibc` provides.

```c
#include <unistd.h>
#include <sys/syscall.h>


int main ()
{
    syscall (1, 1, "Hello World", 11);
    return 0;
}
```

Here is the assembly code for the above file. This is generated by using the `gcc -S filename.c` command. This generates a file with name as `filename.s`

You can see how the arguments are been copied to the registers for calling the function `syscall()`. This is being done so that in the `syscall()` function the arguments can be moved to the right registers for calling the `syscall` instruction.

6.3.3 `syscall` instruction

Now we will do the same in our assembly code. The idea here is to move the right values to the right registers and then just call the `syscall` instruction. The same is achieved is by calling the `syscall()` function.

```
section .text
    global _start
    _start:                 ; ELF entry point
     ; 1 is the number for syscall write ().

    mov rax, 1
    ; 1 is the STDOUT file descriptor.

    mov rdi, 1

    ; buffer to be printed.

    mov rsi, message

    ; length of buffer

    mov rdx, [messageLen]

    ; call the syscall instruction
    syscall

    ; sys_exit
    mov rax, 60

    ; return value is 0
    mov rdi, 0

    ; call the assembly instruction
    syscall

section .data
    messageLen: dq message.end-message
    message: db 'Hello World', 10
.end:
```

Makefile for assembling the code.

```
all:
        nasm -felf64 write.asm          # Assemble the program.
        ld write.o -o elf.write

clean:
        rm -rf *.o

```

Run the make command and run the file `elf.write`. You will see the output of your program on the screen.

```
$ make
nasm -felf64 write.asm  # Assemble the program.
ld write.o -o elf.write

$ ./elf.write
Hello World
```

6.4 Conclusion

In this chapter we saw the different ways of calling a system call. The three ways are

- to call the function directly like calling `write` directly.
- to call the `glibc` interface for calling system calls namely `syscall()`
- to directly call the `syscall` instruction from any assembly file.

CHAPTER 7

Return Values

7.1 Introduction

A system call is called to get some work done by the kernel. How does the kernel notify the caller about the work done?

The process of notifying about the work done is same as that of any other function call. Through return values and `call-by-reference` arguments. A list of error numbers and its definitions can be found in the file `/usr/include/asm-generic/errno-base.h`.

7.1.1 Return Values

The return value, arguments and possible errors related to a system call are well documented in the `man` pages of the system call.

For converting the `errno` to relevant string error (for example `errno` 2 is "No such file or directory") we have the function `strerror()`.

7.1.2 call-by-reference

Some system call return the values using the `call-by-reference` method. For example `read()` system call. The second argument is the buffer where we want the data to be read. The kernel reads the data from the file and copies the data to the passed buffer.

7.1.3 Error Macros

There are predefined macros in the form of `#define`. These codes help us to write a more readable code. In the following text I have listed the error codes from the file `/usr/include/asm-generic/errno-base.h`

```
$ cat /usr/include/asm-generic/errno-base.h
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO      5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
```

7.1.4 Error Explanation

The man page of `errno` explains the above mentioned error codes in detail. Run the command `man 2 errno`.

```
>>>>>>>>>>>> SNIPPED <<<<<<<<<<<<<<<<<<<<<<<<<<<<<

E2BIG           Argument list too long (POSIX.1)

EACCES          Permission denied (POSIX.1)

EADDRINUSE      Address already in use (POSIX.1)

EADDRNOTAVAIL   Address not available (POSIX.1)

EAFNOSUPPORT    Address family not supported (POSIX.1)

>>>>>>>>>>>> SNIPPED <<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

7.1.5 Return Values

See the man page of open system call by the command `man 2 open`. You will see a section like the following explaining the return value of the `open` system call.

```
RETURN VALUE

           open(),  openat(),  and creat() return the new file
           descriptor, or -1 if an error  occurred  (in  which
           case, errno is set appropriately).
```

7.1.6 Error Example

See the man page of open system call using `man 2 open`. There will be section which will have the list of possible errors which this system call can throw.

```
ERRORS
           open(),  openat(),  and  creat()  can fail with the
           following errors:
```

```
        EACCES The requested access  to  the  file  is  not
                       allowed,  or search permission is denied for
                       one of the directories in the path prefix of
                       pathname,  or the file did not exist yet and
                       write access to the parent directory is  not
                       allowed.  (See also path_resolution(7).)

        EDQUOT Where  O_CREAT  is  specified, the file does
                       not exist, and  the  user's  quota  of  disk
                       blocks  or inodes on the filesystem has been
                       exhausted.

        EEXIST pathname  already  exists  and  O_CREAT  and
                       O_EXCL were used.

            >>>>>>>>>>>> SNIPPED <<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

7.2 How system calls return value?

The return value is returned in the `rax` register. We can see this using a debugger. Let us read 50 bytes of a small file and see what is the status of the return value.

```c
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

void print_10_char(char *buf) {
    int i=0;
    printf("\n\n");

    if (buf) {
        for (i=0; i < 10; i++) {
            if (buf[i] != '\0') {
                printf("%c", buf[i]);
            } else {
                break;
            }
        }
    }
}

int main ()
{

    char buf[4096] = "BUFFER";
    int bytes_read = 0, fd;

    fd = open ("/etc/passwd", O_RDONLY);

    if (fd < 0) {
        perror ("\nError opening the destination file");
        exit(1);
```

```
    } else {
        fprintf (stderr, "\nSuccessfully opened the destination file..");
    }
    bytes_read = read (fd, buf, 20);

    /* Print the first 10 bytes and the number of bytes_read */

    printf ("\nBytes Read %d", bytes_read);
    print_10_char(buf);
    close(fd)
    return 0;
}
```

We will now add a breakpoint at the `read()` system call line and see the register's value changing after the system call. See the snippet below. Here we are compiling the code using `make` and then running the code first.

Then we start the `gdb` and set up `displays` to list the registers `rax` and `rsi`. These registers have the return values. `rax` has the number of bytes read and `rsi` has the pointer to the buffer which we are passing for the bytes to be copied.

We setup a breakpoint at `read` call and then we see the state of the registers before and after the read system calls are called.

Note: For linking we are using our own compiled `glibc`. This helps us when we run the debugger.

- Compile and run the command.

```
$ make
gcc -g -c read.c -I `gcc --print-file-name=include` -I /home/rishi/glibc/install_
↪glibc/include
gcc -nostdlib -nostartfiles -static -o read /home/rishi/glibc/install_glibc/lib//crt1.
↪o /home/rishi/glibc/install_glibc/lib//crti.o `gcc --print-file-name=crtbegin.o`␣
↪read.o -Wl,--start-group /home/rishi/glibc/install_glibc/lib//libc.a -lgcc -lgcc_eh␣
↪-Wl,--end-group `gcc --print-file-name=crtend.o` /home/rishi/glibc/install_glibc/
↪lib//crtn.o
$ ./read

Successfully opened the destination file..
Bytes Read 20

root:x:0:0$
```

- Start `gdb`.

```
$ gdb ./read
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./read...done.
```

- Setup the displays and breaks ini gdb.

```
(gdb) display $rax
1: $rax = <error: No registers.>
(gdb) display (char *) $rsi
2: (char *) $rsi = <error: No registers.>
(gdb) break read
Breakpoint 1 at 0x433680: file ../sysdeps/unix/syscall-template.S, line 84.
```

- Run the program. It will stop just before read is called. See the state of the registers.

```
(gdb) r
Starting program: /home/rishi/publications/doc_syscalls/doc/code_system_calls/08/read/
↪read

Successfully opened the destination file..
Breakpoint 1, read () at ../sysdeps/unix/syscall-template.S:84
84      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
1: $rax = 3
2: (char *) $rsi = 0x7fffffffcd10 "BUFFER"
```

- Call the `read()`. See the state of the registers. The `rax` register has the number of bytes read `20` and the `rsi` register has the pointer to the filled buffer.

```
(gdb) n
main () at read.c:41
41          printf ("\nBytes Read %d", bytes_read);
1: $rax = 20
2: (char *) $rsi = 0x7fffffffcd10 "root:x:0:0:root:/roo"

(gdb)
```

7.3 Printing Error Value

Now let us see how do system call show the error encountered in the system calls. In this code we will try to open a file which does not exist and then we will print the global variable `errno` to get the status of the system call. We will also use the above mentioned function `strerror()` to print a more user friendly message.

```
$ make

$ ./elf.open

Error number is 2
File does not exist. Check if the file is there.
Error is: No such file or directory
```

7.4 Conclusion

In this section we learnt in detail about

- How system calls return values to the caller.
- How system calls notify errors to the caller.
- How to see the return values in the register.
- How to convert a error code to a error string.