Systems Programming under Linux (Revision 1.1)

Hello !

This is the preface of the book.

The book tries to cover the systems programming under linux based operating systems in **C**. The most of the part in this book is my personal experience with programming in the systems level of the linux operating system. I am DevNaga. More about me is at the end of the book.

All the source code examples described in this book are self-contained and are compiled and ran on Ubuntu 14, 16 and 17 as well Fedora 26 linux operating system.

Whats new ?

What's new in Revision 1.1

1. fixes
    1. minor corrections and spell mistake errors
    2. statvfs/fstatvfs
    3. mmap
2. features
    1. two example C programs on `getopt` API and their detailed descriptions
    2. every day an example program in git. here (https://github.com/DevNaga/gists)
    3. C++ examples
    4. pthreads
    5. wait and fork()
    6. environment variable
    7. appendix.e - gcc
    8. appendix.c - CAN protocol and the usage in Linux

more to come every week!

Audience:

1. This book is **NOT for those who seek for kernel programming topics**. This book deals with the layer above it, the system calls and the middleware. Most probably the newer versions of it may contain the kernel level API support and kernel programming.
2. Basic C programming skills are required but not mandatory.. the book teaches them as well (but not in great detail).
3. Linux programming skills will be taught in the book and basic usage of the linux operating systems such as the **Ubuntu** is required.
4. This book teaches a realworld, practical linux systems programming.

5. ofcourse, There are too many coding examples in C, (compiled and tested) for the reader to explore and enjoy while learning.

This book is under continuous development. There will always be new revisions of the book with feature updates. Please see the first portion in What's new section for any new version changes.

Almost all of the code samples are kept at GitHub (https://github.com/DevNaga/gists.git). Please feel free to clone the repository.

Please open a ticket for grammatical errors, a new feature request or topic addition. You can also email (mailto:devendra.aaru@gmail.com) me about them.

Author: Devendra Naga
Linkedin (https://in.linkedin.com/in/devendranaga) GitHub (https://github.com/devnaga) twitter (@devnaga448)

Email: mailto:devendra.aaru@gmail.com (mailto:devendra.aaru@gmail.com)

References

The content of this book is created from scratch basing on what i have learnt as a developer for 8 years.# contents

Preface

This chapter mostly concentrates on the C and C++ programming languages and also gives out little bit python scripts to encourage the reader to learn the Python programming language.

The C programming Language

- C programming language is invented by the great programmers **Dennis Ritchie** and **Brian Kernighan**.
- We use C programming language in many areas such as in embedded devices, high performance backend software for servers and web browsers, gaming engines etc.
- C is highly desired programming language to write any performance oriented code.
- C is still a majorly used language in the current competing world where there are new programming languages every quarter.
- C is very useful in programming device drivers or the software that can be closely written to the hardware.
- C programming language is implemented by the C compiler. We use gcc as the C compiler under linux and BSD'ish operating systems. Gcc implements the C programming language specification. The compiler converts the program written in C into the binary program that is suited for the given architecture (X86, ARM, PPC, MIPS and so on).

- To install gcc on the Ubuntu simply use the following command

  ```
  sudo apt-get install gcc build-essentials
  ```

  Under Fedora (as root):

  ```
  dnf install gcc
  ```

- pick an editor to type the source code such as `atom` editor or `vim` that colors the types, functions etc.. to ease up understanding and coding.
- To play with the code and understand the content in this book, A Linux OS and a C compiler is most necessary. I am using Fedora / Ubuntu as the Linux OS and Gcc as the C compiler. If you do not wish to install Linux OS on your computer, you can still try it out with the VirtualBox as a guest OS.

The Python programming Language

- Python is invented by the great programmer **Guido van Rossum**.
- Python is a typeless language. We do not need to specify the variable types, we simply use them for the purpose and they become the types.
- Python is very useful in the automation, server side programming, backend system implementation, Maps, quick scripting and now-a-days it is being used in the Artificial Intelligence.
- Python is an interpreter language. Meaning that an interpreter is needed to understand the program written in Python. This also means that the Python programs can be executed on any operating system and on any processor architecture provided that there is an interpreter for that operating system and the architecture.
- Python language is extensible with the use of modules. The modules are like the libraries that extend the functionality of the Python.
- To install Python interpreter on the Ubuntu simply use the following command

  ```
  sudo apt-get install python
  ```

  Under Fedora (as root):

  ```
  dnf install python
  ```

- pip is a tool used to install python modules. To install pip on the Ubuntu use the following command

  ```
  sudo apt-get install python-pip
  ```

  Under Fedora

  ```
  dnf install python-pip
  ```

Basics and Starting up

Lets start with a sample command line parser program with the coding. The below example deal with both C and Python as well. I generally use **Python** here to 'declutter' the things and to add more clarity.

Command line parser

Let us start with a small program that performs an action based on the command line arguments passed to it.

In C:

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

int add(int a, int b)
{
    return a + b;
}

int sub(int a, int b)
{
    return a - b;
}

void program_usage(char *prog_name)
{
    printf("%s <add/sub> num1 num2\n", prog_name);
}

int main(int argc, char **argv)
{
    if (argc != 4) {
        program_usage(argv[0]);
        return -1; // always return failure if you unsure of what to do
    }

    if (!strcmp(argv[1], "add")) {
        printf("add result %d\n", add(atoi(argv[2]), atoi(argv[3])));
    } else if (!strcmp(argv[1], "sub")) {
        printf("sub result %d\n", sub(atoi(argv[2]), atoi(argv[3])));
    } else {
        program_usage(argv[0]);
        return -1;
    }

    return 0;
}
```

Example: command line sample program

We use gcc (any version > 5.0 is fine) for compiling our programs presented in this book.

when compiled with `gcc -Wall cmdline.c -o cmdline` option, this will generate the `cmdline`binary file.

This file is an executable file. This file can be run by typing the file name as below in the shell.

```
# ./cmdline
```

The `./` tells the shell to execute the program that is present in this directory.

More about the program running in the upcoming chapters. For now, we now know how to compile and run a program.

The program accepts either of two strings namely, add and sub. If either of them are present, it then executes a function that performs the action (Add two integers or Subtract two integers).

The `program_usage`, `sub` and `add` are called the functions. Functions allow the code to be more structured and sensible.

so typically the command line argument would become,

```
# ./cmdline add 28  24
```

or

```
# ./cmdline sub 28 24
```

NOTE: You do not really need to be root to run this program.

A function would typically look as below..

```
return_type function_name(variable_type var_name, variable_type var_name, ..);
```

Each function has `return_type` and accepts set of variables as its input called `arguments`. The return_type can be a variable type such as `int` or `void`. The function, when it has the return type as `void` meaning that the function returns nothing.

The same program is written in Python below:

```
#!/usr/bin/python

import sys

def add(a, b):
    return int(a) + int(b)

def sub(a, b):
    return int(a) - int(b)

def program_usage():
    print sys.argv[0] + " add/sub num1 num2"

if len(sys.argv) != 4:
    program_usage()
    exit(1)

if sys.argv[1] == "add":
    print "add result: " + str(add(sys.argv[2], sys.argv[3]))
elif sys.argv[1] == "sub":
    print "sub result: " + str(sub(sys.argv[2], sys.argv[3]))
else:
    program_usage()
    exit(1)
```

Compilation stages in a C program

Here are the things that happen when you compile a **C** program.

There are 4 stages in a **C** program compilation.

1. Preprocessor
2. Compiler
3. Assembler
4. Linker

Preprocessor:

The preprocessor is a program that converts the C code into a high level code by removing comments, replacing the header files with the actual content of the header files, replacing the macros into their corresponding code etc. Its job is to parse the C program and convert into something that's understandable by the compiler. Find more on the preprocessor in the macros chapter.

This can be done with the gcc using the -E option.

```
gcc -E basics.c
```

when given with -o option it copies the output to the file after the -o.

```
gcc -E basics.c -o basics.i
```

Compiler:

The compiler is another program that converts the source code into assembly code that is understandable by the assembler.

This can be done with the gcc using the **-S** option. We are passing the **basics.i** file after the preprocessor stage to the compiler. Alternatively you can pass **basics.c** and see what happens.

```
gcc -S basics.i
```

This generates a file called **basics.s** that contains the assembly instructions ready for the assembler.

Assembler:

The assembler program converts the assembly code passed by the compiler into an object code.

This can be done with the gcc using the -c option. We are passing the basics.s file that is produced at the compilation stage. Alternatively you can pass basics.c and see what happens.

```
gcc -c basics.s
```

This generates a file called basics.o that is the object file.

running `file` command on the object file results in the following output on a 64 bit machine.

```
ELF 64-bit LSB relocatable, x86-64, version-1 (SYSV), not stripped
```

Linker:

The linker resolves the references to the functions that are defined else where (such as in libraries) by adding the function's addresses so that the loader program can locate and load the functions if necessary. This part is called the linking process. The executable is created after the linking process. We can do all this with the gcc -o flag. As always, try it out with basics.c and see what happens ! :-)

```
gcc -o basics basics.o
```

The final executable's name is `basics`.

We will go deep into each compilation stage in the following chapters where ever necessary.

loops and conditional statements

The 'C' programming language provides the conditional statements `if`, `else` and `else if` to perform various functions. For ex: run the program until this condition satisfy and exit if it does not. This would be the case for the algorithms. They always check for conditions that satisfy and execute and proceed to the next stages.

The if conditional checks for a truth statement. Any number other than 0 is true in C (Yes it means even the negative numbers).

The if condition looks like the below.

```
if (condition) {
    // execute staement(s)
}
```

if the `condition` is evaluated to true, then the statements inside the `if` are executed. Otherwise the statements enclosed in the `if` are not going to be run.

The **otherwise** part we call it the `else` conditional. The else conditional executed if the conditional in the `if` statement is not executed. Very simple.

The `else` condition looks like the below.

```
if (condition) {
    // execute code path 1
} else {
    // execute code path 2
}
```

Example:

```
int a = -1;

if (a) {
    printf("a value is %d\n", a);
} else {
    printf("a value is zero\n");
}
```

Remember that the `else` conditional always follow the `if`.

The compiler always give an error if the `else` conditional does not follow the `if`.

If there are more than one conditions to be executed based on some truth conditionals, we use `else if`.

The `else if` conditional looks like the below.

```
if (condition1) {
    // execute conde path 1
} else if (condition2) {
    // execute code path 2
} else {
    // execute code path 3
}
```

Example:

```
int a = 2;

if (a == 1) {
    printf("a value is %d\n", a);
} else if (a == 2) {
    printf("a value is %d\n", a);
}
```

An else will always be at the last of the `if else if` conditional.

A series of `if` and `else if` conditional statements is also called an `else if ladder`.

For ex: an `else if ladder` would look as below.

```
if (fruit_name == FRUIT_APPLE) {
    printf("Apple\n");
} else if (fruit_name == FRUIT_ORANGE) {
    printf("Orange\n");
} else if (fruit_name == FRUIT_PINEAPPLE) {
    printf("Pine Apple\n");
} else if (fruit_name == FRUIT_GRAPE) {
    printf("Grape\n");
} else {
    printf("Noooo .. i don't have any Fruits\n");
}
```

Always make sure that an `else` statement is always present in an `if.. else if` conditional. This allows not to miss any case that does not satisfy either `if` or `else if`. Most of the software bugs lie around the `if`, `else if` and `else` parts.

Sometimes, the `else if ladder` is replaced with a `switch` statement. The `switch` statement is often and mostly used for data elements of type `int` or `char`. On the otherhand, `switch` can't be used with `float`, `double` or strings. The compiler cries if any of them are being used in a switch.

For ex: the `switch` statement converted fruit example looks as below.

```
switch (fruit_name) {
    case FRUIT_APPLE:
        printf("Apple\n");
   break;
   case FRUIT_ORANGE:
        printf("Orange\n");
   break;
   case FRUIT_PINEAPPLE:
        printf("Pine Apple\n");
   break;
   case FRUIT_GRAPE:
        printf("Grape\n");
   break;
   default:
        printf("Noooo .. i don't have any Fruits\n");
}
```

With in the `switch` statement, a series of `case` blocks with the `default` block can be used. Consider the `case` blocks as the `if` conditionals followed by the `break` statement causing the next `case` statement NOT to execute. If there is no `break` statement, the next conditions may execute and the result will be entirely different. We will see about this in the following paragraphs. The `default` statement is like an `else` conditional. If any of the `case` statements does not satisfy the condition given in the `switch`, the `default` statement gets executed.

For ex:

```
switch (fruit_name) {
    case FRUIT_APPLE:
        printf("Apple\n");
    case FRUIT_ORANGE:
        printf("Orange"\n");
    break;
    default:
        printf("No... i don't have any Fruits\n");
}
```

When fruit_name is passed as FRUIT_APPLE, The above code would let the cases FRUIT_APPLE and FRUIT_ORANGE execute. This is because the `break` statement is not present after the `case` statement is complete.

The conditionals statements are very important in the coding.

There are cases where the `switch` statement is not always `if .. else if`. Consider the following example from the stackoverflow website here (http://programmers.stackexchange.com/questions/162574/why-do-we-have-to-use-break-in-switch)

There comes a situation where you need to loop until some condition is evaluated to true. You can do with the `while` and `for` statements. There is also `do .. while` statement. We will see about these below.

The `while` statement loops till the condition evaluates to false. It looks like the following..

```
while (condition) {
    // series of statements
}
```

When the condition fails, the `while` conditional will not get executed.

The series of statements are evaulated again and again till the condition becomes invalid.

For ex: let us consider a case where we print 10 number from 1.

```
int i = 1;

while (i <= 10) {
    printf("number %d\n", i);
    i ++;
}
```

The i value before the `while` is 1 and the loop starts printing the numbers from 1 till 10 and when i becomes 11 the condition evaulates to false and the loop stops.

The `for` is similar but it has the following advantages.

`for` statement has initializer, condition evaluation and incrementer.

```
for (init; cond; increment) {
}
```

For ex: the `while` loop is rewritten as

```c
int i;

for (i = 1; i <= 10; i ++) {
    printf("i value %d\n", i);
}
```

The initialiser sets the value of the variable to 1 and it gets executed only once. The condition then executes and if condition becomes satisfied the statements with in the for loop are executed. After the last statement is executed, the increment part of the for loop is executed and then the condition with in the for loop. The initializer will only be executed in the beginning.

There can be cases where one needs to run the program forever. This forever programs we call infinite loops.

The infinite loops can be written as the following:

```c
while (1) {
    // statements;
}

for (;;) {
    // statements;
}
```

An infinite program, when ran, can be stopped via signalling it by the `ctrl + c` combination from the Keyboard. We look at the signals more in the upcoming topics.

Infinite loops cause the program to consume all of the CPU causing the system load go high and the temperature of the system. Often, the daemons use the infinite loops in a controlled way to make the program run forever. The controlled way avoids the heavy CPU loads.

getopt

getopt is an API from the **libc** that is used to parse the command line arguments very easily and effectively. It also provides an interactive input to the user of the commad.

The <getopt.h> header file needs to be included in order to use this API.

Example:

```
#include <stido.h>
#include <unistd.h>
#include <getopt.h> // for getopt and friends

int main(int argc, char **argv)
{
    int opt;
    int time_delay = 0;

    while ((opt = getopt(argc, argv, "t:")) != -1) {
        switch (opt) {
            case 't':
                time_delay = atoi(optarg);
            break;
            default:
                printf("%s -t <time-delay in sec>\n", argv[0]);
                return -1;
        }
    }

    printf("sleeping for %d\n", time_delay);
    sleep(time_delay);

    return 0;
}
```

Example: getopt base example

- We compile and generate the binary as the following:

```
gcc -Wall getopt\_example.c -o getopt\_example
```

```
./getopt_example -t 1
```

The above code sleeps for 1 second and stops the execution

- include files : <getopt.h> and <unistd.h>.
- getopt defines a set of variable that can be used by the program.
    - **opterr**: If the value of this variable is nonzero, then getopt prints an error message to the standard error stream if it encounters an unknown option character or an option with a missing required argument. If you set this variable to zero, getopt does not print any messages, but it still returns the character ? to indicate an error.
        - For ex: if we run the above example as the following `./getopt_example -d`. It would print `./getopt_example: illegal option -- d` on to the screen.
    - **optopt**: When getopt encounters an unknown option character or an option with a missing required argument, it stores that option character in this variable. You can use this for providing your own diagnostic messages.

- **optind**: This variable is set by getopt to the index of the next element of the argv array to be processed. Once getopt has found all of the option arguments, you can use this variable to determine where the remaining non-option arguments begin. The initial value of this variable is 1.
- **optarg**: This variable is set by getopt to point at the value of the option argument, for those options that accept arguments.
    - we get this option in our example of the getopt and use it to convert into an integer that gives us the time to sleep in the code. The `optarg` is the most commonly used argument in any basic to an intermediate level program.

Lets take a look at another example below.

Example:

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

int main(int argc, char *argv[])
{
    int ret;
    int opt_t = 0;

    while ((ret = getopt(argc, argv, "t")) != -1) {
        switch (ret) {
            case 't':
                opt_t = 1;
            break;
            default:
                fprintf(stderr, "<%s> [-t]\n", argv[0]);
                return -1;
        }
    }

    if (opt_t) {
        fprintf(stderr, "opt_t is set\n");
    } else {
        fprintf(stderr, "opt_t is not set\n");
    }

    return 0;
}
```

in the above example, we see that in the `getopt` call, the 3rd argument has the `"t"` mising the `:`.

This means, the argument `-t` does not accept any more values, so the program must be run as (after you have compiled ofcourse..)

```
./a.out -t
```

when you run with `-t` option, we set the option variable `opt_t` to demonstrate that the option t was given at command line.

if not given the `else` statement in the code will be executed and prints `opt_t is not set`.

Here's one more example below that accepts multiple options.. see below, also you can download it from here (https://github.com/DevNaga/gists/blob/master/getopt_args.c)

Example:

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

int main(int argc, char *argv[])
{
    int ret;
    char *filename = NULL;
    int number_a = 0;
    int number_b = 0;
    int add = 0;
    int usage = 0;

    while ((ret = getopt(argc, argv, "f:a:b:Ah")) != -1) {
        switch (ret) {
            case 'a':
                number_a = atoi(optarg);
            break;
            case 'b':
                number_b = atoi(optarg);
            break;
            case 'f':
                filename = optarg;
            break;
            case 'A':
                add = 1;
            break;
            case 'h':
            default:
                fprintf(stderr, "<%s> -a <number_a> -b <number_b> -A [add] -f
<filename> -h [help]\n", argv[0]);
            return -1;
        }
    }

    if (add) {
        fprintf(stderr, "add [%d + %d = %d]\n", number_a, number_b, number_a +
number_b);
    }
    if (filename) {
        fprintf(stderr, "filename is given in command line [%s]\n", filename);
    }
}
```

The above program takes multiple arguments such as a string a set of numbers and option without any arguments that we seen in one more program above. finally the option `-h` prints the help of the program on the screen.

```
./a.out

<./a.out> -a <number_a> -b <number_b> -A [add] -f <filename> -h [help]
```

getopt_long

Some commands accept the input as the following...

```
command --list
```

The two `--` allow to provide a descriptive command from the command line. These options are called long options.

Such options are parsed using the `getopt_long` API provided by the **libc**.

The option data structure looks as below that is passed as argument to `getopt_long`.

The data structure is as follows...

```
struct option {
    const char *name;
    int has_arg;
    int *flag;
    int val;
};
```

Here is the description:

name: name of the long options.

has_arg: `no_argument` (0) if the option does not take an argument.

`required_argument` (1) if the option requires an argument. `optional_argument` (2) if the option takes an optional argument.

flag: usually set to 0.

val: the value to be used in short notation in case getopt is used.

```c
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h> // for getopt_long and friends

int main(int argc, char **argv)
{
    int opt;
    int opt_id = 0;
    static struct option long_opt[] = {
        {"option1", required_argument, 0, 'x'},
        {"option2", required_argument, 0, 'y'},
        {"option3", required_argument, 0, 'z'},
        {"usage", no_argument, 0, 'p'},
        {0, 0, 0, 0},
    };

    while ((opt = getopt_long(argc, argv, "x:y:z:p", long_opt, &opt_id)) != -1)
{
        switch (opt) {
            case 'x':
                printf("option1 given %s\n", optarg);
            break;
            case 'y':
                printf("option2 given %s\n", optarg);
            break;
            case 'z':
                printf("option3 given %s\n", optarg);
            break;
            case 'p':
                printf("%s [option1 (x)] "
                       "[option2 (y)] "
                       "[option3 (z)] "
                       "[usage (p)]\n",
                               argv[0]);
            break;
        }
    }

    return 0;
}
```

Data structures

The data structures are key to any software program. They define the layout of the program. In this section we are going to understand about some of the most commonly used data structures.

We are also going to take a dig at the searching and sorting functions too.. These however are important in the data analysis part of the programs.

1. Linked List

1. Think of this as set of objects in series.
2. Objects can be of any type.
3. Dynamically modified (created, deleted, altered)
4. Number of elements in the list is the length of the list
5. Often more time consuming when looking for an element in the list if there are lots of elements.

Linked lists can be created with the simple data structure below.

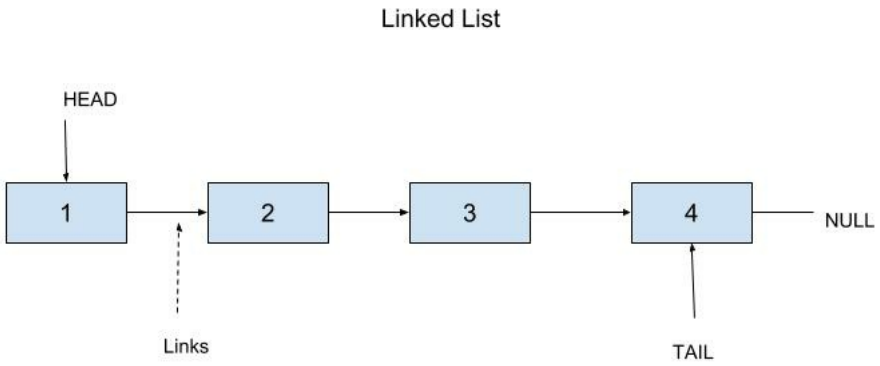
```
struct linked_list {
    void *data;
    struct linked_list *next;
};

struct linked_list *head, *tail;
```

The `data` part is a pointer that pointing to a data structure. A user can point their data structure to this pointer.

The `tail` pointer is used to point to always the end of the list. The end pointer is used to chain the newly created element in the list at the end of the list. The end pointer is then made point to the newly created element.

The following code performs the head and `tail` pointer assignments

```
// if there's no head element add a new head element and point the
// tail to the same so that the next element could be added as
// the next element if the tail such as that in the else part
//
// if the head element is already there, then we have a node in
// the linked list. so let's extend this by linking new element
// as the next element to the tail such as
//
// tail->next = new_node;
if (!head) {
    head = new_node;
    tail = new_node;
} else {
    tail->next = new_node;
    tail = new_node;
}
```



The `if` conditional says if there is no head element, then the new node becomes the head element and also the `tail` node points to the new node. Otherwise, the `new_node` gets added as the next element of the `tail` node and further the `tail` node gets pointed to the `new_node`. Thus if a new element is again getting added, it will always gets added at the end.

The part of the code is keep into a function called `list_add`.

```c
/**
 * @brief - add new element into the linked list at the tail
 */
int list_add(void *elem)
{
    struct linked_list *new_node;

    // allocate new_node
    new_node = calloc(1, sizeof(struct linked_list));
    if (!new_node) {
        return -1;
    }

    new_node->data = elem;

    // follow the same logic as in the above code snippet..
    //
    // if no head, add head and make tail points to it
    //
    // else, add the new node as next element of tail and
    // make tail point to the new node
    if (!head) {
        head = new_node;
        tail = new_node;
    } else {
        tail->next = new_node;
        tail = new_node;
    }

    return 0;
}
```

the function creates a new linked list node and puts the `elem` to the `data` pointer of it.

Then it adds the element at the end of the list or at the beginning of the list (no list head).

The below function adds the elements to the head and so be called `list_add_head`.

```
int list_add_head(void *elem)
{
    struct linked_list *new_node;

    new_node = calloc(1, sizeof(struct linked_list));
    if (!new_node)
        return -1;

    new_node->data = elem;

    if (!head) {
        head = new_node;
        tail = new_node;
    } else {
        new_node->next = head;
        head = new_node;
    }
    return 0;
}
```

The below function adds the elements to the tail or at the end of linked list. The operation is O(1).

```
int list_add_tail(void *elem)
{
    struct linked_list *new_node;

    new_node = calloc(1, sizeof(struct linked_list));
    if (!new_node)
        return -1;

    new_node->data = elem;

    if (!tail) {
        free(new_node);
        return -1;
    }

    tail->next = new_node;
    tail = new_node;

    return 0;
}
```

Before we make the new_node the head, we must assign the next pointer of the new_node to head so that we preserve the entire chain. Next is to make the head pointing to the new_node. Thus keeping the new_node the head.

To get the data pointer of each element, we need to `traverse` each element in the list and retrieve the data pointer.

The following code perform the traversal of the list:

```
    void list_for_each(void (*app_func)(void *data))
    {
        struct linked_list *node;

        for (node = head; node; node = node->next) {
            if (app_func)
                app_func(node->data);
        }
    }
```

The `list_for_each` function goes through each element by advancing into the `next` pointer in the list. At each node, it dereferences the list element and calls the application callback function pointer `app_func`. This is the place where the function pointers come in handy in hiding the details of the linked list layer with the caller.

With the linked list we can delete a particular element in the list. A delete involve the following:

1. find the element in the list.
2. if the element is in the head, make head pointing to the next element after the head. free up the pointer that is at the previous head.
3. else, traverse through each element and if the element is found, make the previous node pointing to the next node. free up the list.

```
int list_delete(void *elem, int (*free_func)(void *data))
{
    struct linked_list *node, *prev;

    // check if the element is at the head node
    if (head->data == elem) {
        node = head;
        head = head->next;
        free_func(node->data);
        free(node);
        return 0;
    }

    node = head;
    prev = head;
    while (node) {
        if (node->data == elem) {
            prev->next = node->next;
            free_func(node->data);
            free(node);
            return 0;
        } else {
            prev = node;
            node = node->next;
        }
    }

    return -1;
}
```

The `free_func` is called when an element is found while we are deleting the node. Before we delete the node, we should make sure whatever the data application is allocated gets freed. Thus we call the `free_func` so that application would get chance to free up the memory that is has allocated.

The free function frees up all the nodes and removes the links. The `free_func` gets called on each linked list node thus notifying the user of the list API to safely free up its context structures.

The below function performs the free job.

```
void list_free(int (*free_func)(void *data))
{
    struct linked_list *prev;
    struct linked_list *node;

    node = head;

    while (node) {
        free_func(node->data);
        prev = node;
        node = node->next;
        free(prev);
    }
}
```

Problem:

1. program to count the number of elements in the list.

---

2. Doubly linked list

1. Similar to the linked list, but the each element contain its own `prev` pointer along with the `next` pointer.
2. Going back into the list is possible.

the doubly linked list looks as the following.

```
struct doubly_linked_list {
    void *data;
    struct doubly_linked_list *prev;
    struct doubly_linked_list *next;
};

struct doubly_linked_list *head, *tail;
```

similar to the linked list, we define the API such as the following.

Doubly linked list API	**description**
dllist_add_tail	add a node at the tail
dllist_add_head	add a node at the head
dllist_for_each_forward	move towards the end of the list from the head
dllist_for_each_reverse	move towards the beginning of the list from the tail
dllist_delte_elem	delete the element from the doubly linked list
dllist_destroy	destroy the created doubly linked list

The `dllist_add_tail` API adds the node to the tail of the list. The code looks as follows.

```
int dllist_add_tail(void *elem)
{
    struct doubly_linked_list *new_node;

    new_node = calloc(1, sizeof(struct doubly_linked_list));
    if (!new_node)
        return -1;

    new_node->data = elem;

    if (!head) {
        head = new_node;
        tail = new_node;
    } else {
        new_node->prev = tail;
        tail->next = new_node;
        tail = new_node;
    }

    return 0;
}
```

This API finds if a node is NULL, adds the node to the head. Otherwise, it points the previous element of the new node to the tail node and makes the tail's next pointer point to the new_node. Then finally makes tail points to the new_node.

The `dllist_add_head` API adds the element to the head. The API looks as follows.

```
int dllist_add_head(void *elem)
{
    struct doubly_linked_list *new_node;

    new_node = calloc(1, sizeof(struct doubly_linked_list));
    if (!new_node)
        return -1;

    new_node->data = elem;

    if (!head) {
        head = new_node;
        tail = new_node;
    } else {
        new_node->next = head;
        head->prev = new_node;
        head = new_node;
    }

    return 0;
}
```

the `dllist_add_head` always adds the node at the beginning of the head. Make the previous pointer of head point to the new_node and new_node next points to the head element and finally making the head node point to the new_node.

the `dllist_for_each_forward` makes the iterator move from the head to the end. At each point, the callback is called and given the user data to the callback. So if the user gives out a callback, the API calls the callback for each node with the userdata.

```
void dllist_for_each_forward(void (*callback)(void *data))
{
    struct dllist *iter;

    for (iter = head; iter; iter = iter->next) {
        callback(iter->data);
    }
}
```

the `dllist_for_each_reverse` makes the iterator move from the tail to the head. At each node we call the callback and pass the data to the user in the callback.

```
void dllist_for_each_reverse(void (*callback)(void *data))
{
    struct dllist *iter;

    for (iter = tail; iter; iter = iter->prev) {
        callback(iter->data);
    }
}
```

```c
int dllist_delete_elem(void *data, void (*callback)(void *data))
{
    struct dllist *elem, *prev;

    if (head->data == data) {
        elem = head->next;
        elem->prev = NULL;
        callback(head->data);
        free(head);
        head = elem;
        return 0;
    }

    elem = head;
    prev = head;

    while (elem) {
        if (elem->data == data) {
            callback(elem->data);
            prev->next = elem->next;
            elem->next->prev = prev;
            free(elem);
            return 0;
        }
        prev = elem;
        elem = elem->next;
    }

    return -1;
}
```

```c
int dllist_destroy(void (*callback)(void *data))
{
    struct dllist *iter, *next;

    iter = head;

    while (iter) {
        callback(iter->data);
        next = iter->next;
        free(iter);
        iter = next;
    }

    return 0;
}
```

3. Circular lists

3.1 Circular linked lists

1. The linked list's last element points to the head instead of NULL. Making it the circular linked list.

the circular linked list looks as the following.

```
struct circular_linked_list {
    void *data;
    struct circular_linked_list *next;
};

struct circular_linked_list *head, *tail;
```

Adding an element to the circular list is same as adding an element to the linked list, but the last element points to the head back.

```
int circular_list_add(void *elem)
{
    struct circular_linked_list *new;

    new = calloc(1, sizeof(struct circular_linked_list));
    if (!new)
        return -1;

    new->data = elem;
    if (!head) {
        head = new;
        tail = new;
    } else {
        tail->next = new;
        tail = new;
    }

    new->next = head;
    return 0;
}
```

The traversal of the list is done as follows, however the check has to be made with the last element not equal to head as the list is circular.

```
int circular_list_for_each(void (*func)(void *data))
{
    struct circular_linked_list *node = head;

    do {
        if (func)
            func(node->data);
        node = node->next;
    } while (node != head);

    return 0;
}
```

The callback function is called at each element traversal.

3.2 Circular doubly lists

1. The doubly linked list's last element points to the head instead of NULL. Making it the circular doubly linked list.

the circular doubly linked list looks as the following.

```
struct ciruclar_doubly_linked_list {
    void *data;
    struct circular_doubly_linked_list *next;
    struct circular_doubly_linked_list *prev;
};

struct circular_doubly_linked_list *head, *tail;
```

4. Queues

1. Same as linked list.
2. new entries are added at the last. Also called enqueue.
3. the elements are retrieved at the front. Also called dequeue.
4. The queue is also called FIFO (first in first out).

```
    struct queue {
        void *data;
        struct queue *next;
    };

    struct queue *front, *rear;
```

The queue front and rear are initialised when a new element is added. The addition of new element is as the following.

```
int enqueue(void *data)
{
    struct queue *node;

    node = calloc(1, sizeof(struct queue));
    if (!node) {
        return -1;
    }

    node->data = data;
    if (!rear) {
        rear = node;
        front = node;
    } else {
        rear->next = node;
        rear = node;
    }

    return 0;
}
```

The dequeue is performed by taking one element out of the front end.

```
int dequeue(void (*dequeue_func)(void *data))
{
    struct queue *node;

    node = front;

    if (!node)
        return -1;

    front = node->next;
    dequeue_func(node->data);

    return 0;
}
```

The queues above are dynamic lists. Meaning, there is no limit on the number of elements that be added. Only the memory limit would be the limitting factor.

5. Stacks

A stack is a container of objects that are inserted and removed in Last in First out (LIFO) manner. Putting the elements into the stack is called push and taking out the elements out of the stack is called pop.

push adds an item into the top of the stack and pop removes the item from the top.

```
struct stack_list {
    void *item;
    struct stack_list *next;
};

struct stack_list *head, *tail;
```

The data structure is similar to the linked list but the elements are keep on adding at the head so that the pop can take the element out of the head and give it to the user.

Here is the push API that adds the element at the head.

```
int push(void *elem)
{
    struct stack_list *new_node;

    new_node = calloc(1, sizeof(struct stack_list));
    if (!new_node)
        return -1;

    new_node->item = elem;
    if (!head) {
        head = new_node;
        tail = new_node;
    } else {
        new_node->next = head;
        head = new_node;
    }

    return 0;
}
```

The pop API takes the element out of the head and calls the user callback.

```
int pop(void (*callback)(void *data))
{
    struct stack_list *new_node;

    if (!head)
        return -1;

    new_node = head->next;

    callback(head->item);
    free(head);
    head = new_node;
    return 0;
}
```

6. Hash tables

A hash table is a data structure used to implement an associative array, a structure that can map keys to values. Hash tables are used to find the index based on the hash value.

To do this, they compute hash of the input and add the value into the array. If more than one input matches to the same hash, it is called collision.

To add an element, we compute the hash out of it using the hashing formula. We then derive the index out of the hash value and then add the element to the index.

The perfect hash function was implemented by `Dan Bernstein`. And it is the following snippet.

```
static int hash = 5381;

hash = ((hash << 5) + hash) + number;
```

the index is then calcuated using the below snippet.

```
index = hash % size;
```

Collisions are the problems in hash tables. When the input matches to the same hash, this is treated as collision. When a collision occur, the elements cannot be added at the right index. When the collision occur, the element is added at the tail of the data. The best hashing algorithm prevents the number of collisions.

Example hash program: You can find it here (https://github.com/DevNaga/gists/blob/master/hash_tables.c)

```c
/***
 * @simple hash table implementation in C
 *
 * Author: Dev Naga <devendra.aaru@gmail.com>
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * List elements
 */
struct list_elem {
    void *spec_data;
    struct list_elem *next;
};

/**
 * Hash table
 */
struct hash_list {
    struct list_elem *item;
};

static int table_size;
static struct hash_list *hlist;

/**
 * Init the hash tables witha pre-defined table
 */
int hlist_init(int hlist_size)
{
    hlist = calloc(hlist_size, sizeof(struct hash_list));
    if (!hlist) {
        return -1;
    }

    table_size = hlist_size;
    return 0;
}
```

```c
/**
 * Hash the data with a simple addition of characters
 */
static int hlist_hash_data(char *hash_str)
{
    int hash = 0;
    int i;

    for (i = 0; i < strlen(hash_str); i ++) {
        hash += hash_str[i];
    }

    return hash % table_size;
}

static int hlist_add_item_new(int index, void *item)
{
    hlist[index].item = calloc(1, sizeof(struct list_elem));
    if (!hlist[index].item) {
        return -1;
    }

    hlist[index].item->spec_data = item;
    hlist[index].item->next = NULL;

    return 0;
}

static int hlist_add_item_append(int index, void *item)
{
    struct list_elem *last, *iter;

    iter = hlist[index].item;
    last = iter;
    while (iter) {
       last = iter;
       iter = iter->next;
    }

    struct list_elem *new;

    new = calloc(1, sizeof(struct list_elem));
    if (!new) {
        return -1;
    }

    new->spec_data = item;
    last->next = new;

    return 0;
}
```

```
/**
 * Find item in the hash tables if exist
 */
int hlist_find_item(void *item, int (*compare)(void *a, void *b))
{
    int i;

    for (i = 0; i < table_size; i ++) {
        struct list_elem *list;
        int res;

        for (list = hlist[i].item; list; list = list->next) {
            res = compare(item, list->spec_data);
            if (res) {
                return 1;
            }
        }
    }

    return 0;
}

/**
 * Add an item in list.. append if already hashed to same data, add new if does
not
 */
int hlist_add_item(void *item, char *hash_str)
{
    int index;

    index = hlist_hash_data(hash_str);

    if (hlist[index].item == NULL) {
        hlist_add_item_new(index, item);
    } else {
        hlist_add_item_append(index, item);
    }

    return 0;
}

/**
 * Print hash table elements
 */
void hlist_print(void (*data_callback)(void *data))
{
   int i;

   for (i = 0; i < table_size; i ++) {
       struct list_elem *list;
```

```
        printf("hash item: <%d>\n", i);
        for (list = hlist[i].item; list; list = list->next) {
            printf("\t list->spec_data <%p>\n", list->spec_data);
            data_callback(list->spec_data);
        }
    }
}

/***
 * Test program
 */
void print_items(void *data)
{
    //printf("item : <%s>\n", data);
}

int compare_item(void *a, void *b)
{
    char *a_1 = a;
    char *a_2 = b;

    return (strcmp(a_1, a_2) == 0);
}

int main(int argc, char **argv)
{
    int i;

    hlist_init(20);

    for (i = 1; i <= argc - 1; i ++) {
        if (!hlist_find_item(argv[i], compare_item)) {
            hlist_add_item(argv[i], argv[i]);
        }
    }

    hlist_print(print_items);

    return 0;
}
```

For more info on the data structures and on C/C++ languages refer this book [here (https://leanpub.com/c_cpp_refman)](https://leanpub.com/c_cpp_refman)

string manipulations

The C library (libc) provides an interface to manipulate with the strings.

A string is a sequence of characters. A sequence of characters are stored in a character array.

```
char char_array[10];
```

The above line defines a char_array of length 10 bytes. To keep characters in the array we could do the following.

```
char_array[0] = 'l';
char_array[1] = 'i';
char_array[2] = 'n';
char_array[3] = 'u';
char_array[4] = 'x';
char_array[5] = '\0';
```

A string is always terminated by a null termination mark '\0'. The null terminator marks the end of the character array or string.

Or another way of assigning the string is the following.

```
char char_array[] = "linux";
```

Here in the char_array, we do not have to specify the size. Such variables are stored in the data section part of the final executable.

The printf or fprintf functions can be used to print the strings. The format specifier %s is used to print the string.

The below code prints the char_array on to the console.

```
printf("%s", char_array);
```

The <ctype.h> provides the character describing API to find out if the character is a number, whitespace, alphabet.

Some of the useful API are

API	description
isalpha	return 1 if character is an alphabet
islower	return 1 if character is a lower case letter
isupper	return 1 if character is an upper case letter
isalpha	return 1 if character is an alphabet
isdigit	return 1 if character is a digit
isalnum	return 1 if character is an alphabet and numeric
isxdigit	return 1 if chacter is a hexadecimal digit
ispunct	return 1 if character is a punctuation character
isspace	return 1 if character is a whitespace character
isblank	return 1 if character is a blank space
isgraph	return 1 if character is a graphic character
isprint	return 1 if character is a printing character
iscntrl	return 1 if character is a control character
isascii	return 1 if character is an ascii character
toupper	convert lowercase letter to the upper case
tolower	convert uppercase letter to the lower case
toascii	convert letter to ASCII

Reading of the strings can be performed by the use of `fgets` function. However, the `scanf` can also be used but is too dangerous to read strings. Thus it is not a preferrable choice when reading the strings from the command line. The sample usage of `fgets` when reading the strings looks as follows...

```
char data[10];

fgets(data, sizeof(data), stdin);
```

We read the string `data` of length 10 from the `stdin`. Where `stdin` is the standard input as we know it.

The standard library provides a string manipulation functions such as the following. Include the header file `<string.h>` for the API declarations.

function	description
strlen(str)	calculate the length of the string
strcpy(dst, src)	copy the string from src to dst
strcat(str2, str1)	concatenate str2 and str1
strcmp(str1, str2)	compare two strings str1 and str2
strchr(str, chr)	find a character chr in the str
strstr(str1, str2)	find the str2 position with in str1

The above API are enough to perform the string manipulation of almost any kind of strings.

Strlen

The `strlen` function usage is demoed below.

prototype: `size_t strlen(const char *s);`

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

int main(int argc, char **argv)
{
    int len;

    if (argc != 2) {
        printf("%s <string name>\n", argv[0]);
        return -1;
    }

    len = strlen(argv[1]);
    printf("length of the string is %d\n", len);

    return 0;
}
```

Alternatively the length of the string can be calculated manually. The length of the string is the count of characters till the null termination mark except the null terminator.

```
int own_strlen(char *string)
{
    int i = 0;

    while (*string != '\0') {
        // increment i if there is a valid character
        i++;
        // move to the next address
        string++;
    }

    return i;
}
```

Strcpy

The `strcpy` function usage is demoed below.

prototype: `char *strcpy(char *dst, const char *src);`

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

int main(int argc, char **argv)
{
    char *dst;
    int dst_len;

    if (argc != 2) {
        printf("%s <string name>\n", argv[0]);
        return -1;
    }

    // find the length of the destination and add 1 for null terminator
    dst_len = strlen(argv[1]) + 1;

    dst = calloc(1, dst_len);
    if (!dst) {
        return -1;
    }

    strcpy(dst, argv[1]);

    printf("Source %s Destination %s\n", argv[1], dst);

    free(dst);

    return 0;
}
```

Alternatively we can write own string copy function as the following.

```
char* own_strcpy(char *src, char *dst)
{
    int i = 0;

    while (*dst != '\0') {
        src[i] = *dst;
        i++;
    }

    return src;
}
```

Strcat

The `strcat` function concatenates or joins two strings together. The second string is joined at the end of the second string.

prototype: `char *strcat(char *s1, const char *s2);`

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

int main(int argc, char *argv[])
{
    char string1[100];

    if (argc != 3) {
        fprintf(stderr, "%s <string1> <string2>\n", argv[0]);
        return -1;
    }

    strcpy(string1, argv[1]);
    strcat(string1, argv[2]);

    printf("Source1 %s Source2 %s Destination %s\n", argv[1], argv[2],
string1);

    return 0;
}
```

Strcmp

The `strcmp` function compares two strings. If the strings are not equal then a non zero value is returned otherwise zero is returned.

prototype: `int strcmp(const char *s1, const char *s2);`

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int ret;
    char *string1, *string2;

    if (argc != 3) {
        fprintf(stderr, "%s <string1> <string2>\n", argv[0]);
        return -1;
    }

    string1 = argv[1];
    string2 = argv[2];

    ret = strcmp(string1, string2);

    printf("strings are %s\n", (ret == 0) ? "Equal" : "Not Equal");
    return 0;
}
```

Strstr

strstr locates the first occurence of the given substring in the main string.

prototype: `char *strstr(const char *s1, const char *s2);`

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int ret;
    char *main_str, *sub_str, *match;

    if (argc != 3) {
        fprintf(stderr, "%s <main string> <sub string>\n", argv[0]);
        return 0;
    }

    main_str = argv[1];
    sub_str = argv[2];

    match = strstr(main_str, sub_str);
    printf("match found %s\n", match ? "Yes": "No");
    if (match) {
        printf("matching address %p string %s\n", match, match);
    }

    return 0;
}
```

Strchr

strchr API locates the first occurence of the given character in the original string and returns the address of it.

prototype: `char *strchr(const char *s, int c);`

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

int main(int argc, char **argv)
{
    int ret;
    char *string, ch, *string_pos;

    if (argc != 3) {
        fprintf(stderr, "%s <string> <character>\n", argv[0]);
        return -1;
    }

    string = argv[1];
    ch = argv[2][0];

    string_pos = strchr(string, ch);
    if (string_pos) {
        fprintf(stderr, "character pos found %p and string %s\n", string_pos,
string_pos);
    } else {
        fprintf(stderr, "character pos not found\n");
    }

    return 0;
}
```

Strcasecmp

strcasecmp compares the two strings and matches them ignoring the case.

to use this API we should include `<strings.h>`.

prototype: `int strcasecmp(const char *s1, const char *s2);`

```c
#include <stdio.h>
#include <strings.h>

int main(int argc, char **argv)
{
    int ret;
    char *string1, *string2;

    if (argc != 3) {
        fprintf(stderr, "%s <string1> <string2>\n", argv[0]);
        return -1;
    }

    string1 = argv[1];
    string2 = argv[2];

    ret = strcasecmp(string1, string2);

    printf("strings are %s\n", (ret == 0) ? "Equal": "Not Equal");
    return 0;
}
```

Strstr

strstr finds a substring inside a string. It returns the address of the first character if the substring has been found and null otherwise.

prototype: `char *strstr(const char *haystack, const char *needle);`

```c
#include <string.h>
#include <stdio.h>

int main(int argc, char **argv)
{
        char *str;
        int ret;

        if (argc != 3) {
                printf("%s <string1> <substring>\n", argv[0]);
                return -1;
        }

        str = strstr(argv[1], argv[2]);
        if (str) {
                fprintf(stderr, "Found substring %s\n", str);
        } else {
                fprintf(stderr, "Substring not found\n");
        }

        return 0;
}
```

Problems:

The following problems are given to encourage the reader to put things that are learnt, into practice.

1. Program to find the frequency of occurence of a character e in the string "frequency".
2. Program to find the length of a string "Meaningful".
3. Program to sort the set of strings in alphabetical order. The strings are "Apple", "Banana", "Pine Apple", "Grapes", "Guava" and "Peach".
4. Given two strings "Apple" and "Banana". Concatenate from the third character of both of the strings. Meaning the final output would be "Appana".
5. Look up a string in a set of strings. Take any large set of strings and look up any string.

bit manipulations

The AND (&), OR (|), XOR (^), Negate (~), Not (!) and bit shifts (<< and >>) operations are possible in C.

The most common uses of the bit manipulation operations are in the device driver programs where the hardware operations are dependent on the bit values in the shift registers etc.

AND operation

The AND operation is used to test if the bit is set or unset. A set is called a 1 and unset or clear is called a 0.

For example consider the following number 128 (in hex 0x80).

```
a = 0x80
MSBit_set = !!(a & 0x80)
if (MSBit_set)
    printf("MSBit is set\n");
else
    printf("MSBit is not set\n");
```

NOT operaton

The above program tests if the Most significant bit (MSbit) of the variable a is set. The test is performed with a mask 0x80. The two Not (!) operators have the following purpose. The Not operator first inverts the value. If the value is not zero, it inverts it into 0 and if the value is 0, it inverts into 1. A Not operation that is performed twice would only give either 1 or 0 only.

Ex: if the Not operation performed on 12, !12 would give 0. and a Not operation again on 0 would give us 1. that is the Not operation results in a binary value. Thus if a value that is non-zero, a twice Not operator would make it 1. If the value that is zero, a twice Not operator would make it 0.

OR operation

An OR operation is used mostly to add or set bits into a byte or group of bytes. Let us take the below example

```
uint8_t var = 0;

var |= 0x01;

printf("0x%02x\n", var);

var |= 0x02;

printf("0x%02x\n", var);
```

The second statement `var |= 0x01` would set 1 in the 0'th bit of the byte 0.
When the third statement is executed it prints the value 0x1.

The bit representation would look as follows.

```
00000001
```

The fourth statement `var |= 0x02` would set 1 in the 1'st bit of the byte 1.
It does not change any other bits except the 1st bit. When the fifth statement is executed it prints the value 0x3.

The bit representation would look as follows.

```
00000011
```

XOR operation

The XOR operation uses the ^. The operation looks as below...

Bit 1	Bit 2	result
0	0	0
0	1	1
1	0	1
1	1	0

The `Bit 1` is XOR'ed with `Bit 2` to give out the result.

When both bits are same the result is a `0`.

When both bits are not same the result is 1.

For ex:

```
a = 10, b = 5;

printf("value %d\n", a ^ b);
```

Left shift operation

The left shift operation uses <<. The operation shifts the bits towards left by the given number of positions. The left shift operation is also a pow(2, x) where x is the left shift number. The resulting number is a multiplication of the original number that is being shifted.

For example:

```
val = 2 << 2;
printf("value %d\n", val);
```

The above example results in the output of **8**.

The original value 2 can be written in binary as

00000010

Shifting by 2 bits left will result in the following.

00001000

The above statement is similar to

```
val = 2 * pow(2,2);
printf("value %d\n", val);
```

Right shift operation

The right shift operation uses '>>'. The operations shifts the bits towards right by the given number of positions. The right shift operation is also a pow(2, x) where x is the right shift number. The resulting number is a division of the original number that is being shifted.

For example:

```
val = 8 >> 2;
printf("value %d\n", val);
```

The above example results in the output of **2**.

The original value 8 can be written in binary as

00001000

Shifting by 2 bits right will result in the following.

00000010

The above statement is similar to

```
val = 8 / pow(2,2);
printf("value %d\n", val);
```

Negate opertaion

The computer stores the digits in binary form unlike we use them either in decimal or hexadecimal.

This is called the base of a number. The binary number is in base 2, decimal is base 10 and hexadecimal is base 16.

atoi and string to number conversion API

The `atoi` function perform an ASCII to integer conversion of a number stored in the string form. If it can't convert it, then it returns simply 0. However, value 0 is also a number and is valid. This means the caller does not know if the conversion was succesful or a failure has occured. This is really ambiguious in a software and the results are unspecified.

So in most of the cases `atoi` is only used for simple numbers and to programs that the input number string expected is always correct. In most of the cases, the `strtol` family of functions are preferred.

We have written our own `atoi` function here. This version detects the errors by checking if a character is non number (`isdigit` macro from the `<ctype.h>`) and fails on an invalid input.

To use `atoi` and `atol` include the header `<stdlib.h>`

The rest of the code here performs a scanning of integer from a string input.

our own string to integer conversion function

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

/**
 * @brief - our own atoi conversion program
 * @param [in] string - string that has the integer value init
 * @param [out] result - integer pointer that will have the final converted
integer
 *
 * @returns returns 0 on success and -1 on failure
 */
int our_atoi(char *string, int *result)
{
    int num;
    int ret;
    int i;
    int len = strlen(string);

    for (i = 0; i < len; i++) {
        if (!isdigit(string[i]))
            return -1;
    }

    ret = sscanf(string, "%d", &num);
    if (ret == 1) {
        // copy over the converted string into result
        *result = num;
        return 0;
    }

    return -1;
}
```

strtol

The strtol converts a string to the `long` type. The prototype looks as follows.

```c
long strtol(const char *str, char **endptr, int base)
```

It returns the converted value from the string `str`.

If the string does not contain the number (or any character that is present in the string), it will store the string from that character into the `endptr`.

The base variable is used to tell the function if the string is in base 10 (i.e decimal numbers) or base 16 (hexadecimal numbers).

For hexadecimal numbers the input can either have the `0x` or does not need to.

The below example shows the usage of `strtol`. It takes an input string from the command line, uses `strtol` and converts it into the decimal number. After the conversion, it prints the number on to the console.

The *strtol* example: (base 10)

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

int main(int argc, char **argv)
{
    int data;
    char *errorp = NULL;

    if (argc != 2) {
        printf("%s integer\n", argv[0]);
        return -1;
    }

    data = strtol(argv[1], &errorp, 10);
    if (errorp && (errorp[0] != '\0')) {
        printf("invalid number %s\n", argv[0]);
        return -1;
    }

    printf("converted %d\n", data);

    return 0;
}
```

With this example, we now write a safe string to integer conversion API instead of using the buggy `atoi` API.

The below function converts a string (the number is in decimal fashion) to an integer.

```c
/**
 * @brief - string to integer converter (in decimal)
 *
 * @param[in] string - input string containing the number
 * @param[out] int_var - integer pointer that will have the final converted
number
 *
 * @returns returns -1 on failure and 0 on success
 */
int string_to_int_d(char *string, int *int_var)
{
    int var;
    char *errorp = NULL;

    var = strtol(string, &errorp, 10);
    if (errorp && (errorp[0] != '\0')) {
        return -1;
    }

    *int_var = var;
    return 0;
}
```

The below function converts a string (the number is in hexadecimal fashion) to an integer.

```c
/**
 * @brief - string to integer converter (in hexadecimal)
 *
 * @param[in] string - input string containing the number
 * @param[out] int_var - integer pointer that will have the final converted
number
 *
 * @returns returns -1 on failure and 0 on success
 */
int string_to_int_h(char *string, int *int_var)
{
    int var;
    char *errorp = NULL;

    var = strtol(string, &errorp, 16);
    if (errorp && (errorp[0] != '\0')) {
        return -1;
    }

    *int_var = var;
    return 0;
}
```

strtoul

The `strtoul` function converts a number in the string into the unsigned long. The prototype of the function is as follows.

```
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

The following example shows the usage of the API.

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

int main(int argc, char **argv)
{
    char *endptr = NULL;
    uint32_t data;
    int base;

    if (argc != 3) {
        printf("%s [number] [base - 10, 16]\n", argv[0]);
        return -1;
    }

    base= atoi(argv[2]);

    data = strtoul(argv[1], &endptr, base);
    if (endptr && (endptr[0] != '\0')) {
        printf("invalid string number %s\n", argv[1]);
        return -1;
    }

    printf("converted value after strtoul %u\n", data);

    return 0;
}
```

strtod

The `strtod` converts a string into a `double`. The prototype looks as follows.

```
double strtod(const char *nptr, char **endptr);
```

Just like other functions, it is always necessary to check the `endptr`.

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

int main(int argc, char **argv)
{
    int ret;
    char *endptr = NULL;
    double val;

    if (argc != 2) {
        printf("%s [double] \n", argv[0]);
        return -1;
    }

    val = strtod(argv[1], &endptr);
    if (endptr && (endptr[0] != '\0')) {
        printf("invalid double value %s\n", argv[1]);
        return -1;
    }

    printf("converted value after strtod %f\n", val);

    return 0;
}
```

Usually, one can write a small set of APIs that use the `strto*` library functions, to make sure the string to type conversion is handled properly.

In, real world, most of the time, the `atoi` is not a preferred solution because the ambiguity of it. Usually, error reporting and analysis is missing in the `atoi` function. I would prefer using the `strto*` family of
functions over the `atoi` when the program involve sufficient complexity so that i do not have to deal with the mistakes that are caused by the `atoi`.

offsetof

offsetof is a macro defined in <stddef.h>## System calls and library functions

The differences are:

1. System call is a function call that involves kernel to execute part of the code and return the value back to the userspace. Library function does not simply perform that operation. It simply performs a job that does not involve going into the kernel and executing on behalf of the user.
2. Example: `system` is a system call that executes a command. The call involve, going into the kernel space, creating the process, executing etc. The `strcpy` is a function call that simply copies source string into the destination. It does not involve system call to go into kernel space and copy the string (because that is un-needed).

when a system call happens, the execution enters from usermode to kernelmode and the kernel executes the called function and returns the output back to the userspace once the execution finishes.

When system calls fail, they also set the `errno` variables accordingly to best describe the problem and why the failure has happened. This is returned as the result of the system call operation from the kernel. The userspace then captures this value and returns the -1 on failure and non negative on success. the error result is then copied to the `errno` variable.

`errno` is a global variable and to be used carefully protected with in the threads. the `errno.h` should be included to use this variable. The `asm-generic/errno.h` contains all the error numbers. (Although one can only include the `errno.h` and not the `asm-generic/errno.h` as the later is more platform specific).

`perror` is a useful function that describes the error in the form of a string and outputs to `stderr`

A small example demostrating the `perror` is shown below. (You can also view / Download here (https://github.com/DevNaga/gists/blob/master/errno_strings.c))

```
#include <stdio.h>
#include <errno.h> //for errno and perror
#include <string.h> // for strerror

int main(int argc, char **argv)
{
    FILE *fp;

    fp = fopen(argv[1], "r");
    if (!fp) {
        fprintf(stderr, "failed to open %s\n", argv[1]);
        perror("fopen:");
        return -1;
    }

    perror("fopen:");
    printf("opened file %s [%p]\n", argv[1], fp);

    fclose(fp);

    return 0;
}
```

Example: perror example

We compile it with `gcc -Wall errno_strings.c`. It will generate an `a.out` file for us.

Then we run our binary with the correct option as below:

```
./a.out errno_string.c

fopen: Success
opened file errno_strings.c [0xeb1018]
```

The `perror` gives us that the file has been opened successfully. The filepointer is then printed on to the screen.

Then we run our binary with the incorrect option as below:

```
./a.out errno_string.c.1

failed to open errno_string.c.1
fopen: No such file or directory
```

Linux provides another API to print the error message based on the error number variable `errno`.

`error` is another API provided by the Glibc. It is declared as follows.

```
void error(int status, int errno, const char *format, ...);
```

The `status` variable is usually set to 0. The `errno` variable is the `errno`.
The format is any message that the program wants to print.

The below example provides an idea of the `error` function.

```
#include <stdio.h>
#include <error.h>
#include <errno.h>

int main(void)
{
    int fd = -1;

    close(fd);
    error(0, errno, "failed to closed fd\n");

    return 0;
}
```

Below is another example of printing error number with using the `strerror`. Download [here (https://github.com/DevNaga/gists/blob/master/strerror.c)](https://github.com/DevNaga/gists/blob/master/strerror.c)

```c
#include <stdio.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void)
{
    char *file = NULL;
    int fd;

    fd = open(file, O_RDWR);
    if (fd < 0) {
        printf("failed to open file: error: %s\n", strerror(errno));
        return -1;
    }
    return 0;
}
```

System() system call

The `system()` system call is used to execute a shell command in a program. Such as the following code

```c
system("ls");
```

Will simply execute the `ls` command.

During the execution of the function, the `SIGCHLD` will be blocked and `SIGINT` and `SIGQUIT` will be ignored. We will go in detail about these signals in the later chapter **signals**.

The `system` function returns -1 on error and returns the value returned by command. The return code is actually the value left shifted by 7. We should be using the `WEXITSTATUS(status)`.

However the `system` system call has some serious vulnerabilities and should not be used. [Here (https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=2130132)](https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=2130132) is the link.

Exec family of calls

The exec family of functions create a new process image from the given binary file or a script and replaces the contents of the original program with the contents of the given binary file or a script. These functions only return in case of a failure such as when the path to the binary / script file is not found or the kernel cannot resever any more memory for the program to execute.

the following are the exec functions.

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *path, const char *arg, ...);
int execle(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

system variables

sysconf is an API to get the current value of a configurable system limit or option variable.

The API prototype is the following

```
long sysconf(int name);
```

symbolic constants for each of the variables is found at include file<unistd.h>. The name argument specifies the system variable to be queried.

sysconf() example on the max opened files:

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

int main(void)
{
    int maxfd;
    maxfd = sysconf(_SC_OPEN_MAX);
    printf("maxfd %d\n", maxfd);
    return 0;
}
```

to get the system virtual memory page size the _SC_PAGESIZE is used. Below example shows the use of the _SC_PAGESIZE. Download here (https://github.com/DevNaga/gists/blob/master/pagesize.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf("sysconf(pagesize) %ld\n", sysconf(_SC_PAGESIZE));
}
```

to get the maximum length of arguments in bytes, passed to a program via command line, use _SC_ARG_MAX. Below example illustrates this. Download here (https://github.com/DevNaga/gists/blob/master/sysconf_argmax.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf("sysconf(_SC_ARG_MAX) %ld\n", sysconf(_SC_ARG_MAX));
}
```

The _SC_CLK_TCK gets the number of clockticks on the system. Defaults to 100. Below is an example, Download here (https://github.com/DevNaga/gists/blob/master/clktick.c)

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

int main()
{
    int clock_tick;

    clock_tick = sysconf(_SC_CLK_TCK);
    printf("clock ticks %d\n", clock_tick);

    return 0;
}
```

max priority of the message queue can be obtained by using the _SC_MQ_PRIO_MAX. Below is an example, Download here (https://github.com/DevNaga/gists/blob/master/mq_prio_max.c)

```c
#include <stdio.h>
#include <limits.h>
#include <unistd.h>

int main()
{
    printf("mq_priority_max %ld\n", sysconf(_SC_MQ_PRIO_MAX));

    return 0;
}
```

to summarise, the following sysconf varialble exists:

variable type	meaning
_SC_OPEN_MAX	max open file descriptors
_SC_PAGESIZE	default virtual memory page size
SC_ARG_MAX	maximum length of command line args in bytes
_SC_CLK_TCK	get the number of clock ticks

resource limits

There are two other APIs to get or set the resource limits on the process.

1. getrlimit
2. setrlimit

The APIs are defined in `<sys/resource.h>`. The prototypes are as follows.

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

The `struct rlimit` is defined as below.

```
struct rlimit {
    rlim_t rlim_cur; // soft limit
    rlim_t rlim_max; // hard limit
}
```

The two APIs return 0 on success and -1 on failure. The corresponding error descriptive functions will be used to describe the return.

The limits are 15 provided by the kernel at asm-generic/resource.h.

Resource name	description
RLIMIT_CPU	maximum CPU time
RLIMIT_FSIZE	maximum file size in bytes
RLIMIT_DATA	maximum data size
RLIMIT_STACK	maximum stack size in bytes
RLIMIT_CORE	maximum core file size in bytes
RLIMIT_RSS	maximum RSS size
RLIMIT_NPROC	maximum number of processes that user may be running
RLIMIT_NOFILE	maximum number of open files
RLIMIT_MEMLOCK	maximum number of bytes a process can lock into memory
RLIMIT_AS	maximum address space size
RLIMIT_LOCKS	maximum file locks held
RLIMIT_SIGPENDING	maximum number of pending signals that to be delivered to the process
RLIMIT_MSGQUEUE	maximum bytes in posix message queue
RLIMIT_NICE	maximum nice priority allowed
RLIMIT_RTPRIO	maximum realtime priority allowed
RLIMIT_RTTIME	timeout of RT tasks in use

The below example illustrates the `rlimit API` uses. Both the `getrlimit` and `setrlimit` are described in the below example.

```
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <sys/time.h>
#include <sys/resource.h>

void get_max_addr_space()
{
    struct rlimit rlim;
```

```
    int ret;

    ret = getrlimit(RLIMIT_AS, &rlim);
    if (ret < 0) {
        fprintf(stderr, "failed to getrlimit\n");
        return;
    }

    printf("soft : %lu hard : %lu\n",
                            rlim.rlim_cur, rlim.rlim_max);
}

void get_max_file_size()
{
    struct rlimit rlim;
    int ret;

    ret = getrlimit(RLIMIT_FSIZE, &rlim);
    if (ret < 0) {
        fprintf(stderr, "failed to getrlimit\n");
        return;
    }

    printf("soft: %lu hard: %lu\n",
                            rlim.rlim_cur, rlim.rlim_max);
}

void get_max_stack_size()
{
    struct rlimit rlim;
    int ret;

    ret = getrlimit(RLIMIT_STACK, &rlim);
    if (ret < 0) {
        fprintf(stderr, "failed to getrlimit\n");
        return;
    }

    printf("soft: %lu hard: %lu\n",
                            rlim.rlim_cur, rlim.rlim_max);
 }

void get_max_cpu_time()
{
    struct rlimit rlim;
    int ret;

    ret = getrlimit(RLIMIT_CPU, &rlim);
    if (ret < 0) {
        fprintf(stderr, "failed to getrlimit\n");
        return;
    }
```

```
    printf("soft: %lu hard: %lu\n",
                            rlim.rlim_cur, rlim.rlim_max);
}

void set_max_stack_size(int n, char **argv)
{
    struct rlimit rlim;
    int ret;

    ret = getrlimit(RLIMIT_STACK, &rlim);
    if (ret < 0) {
        fprintf(stderr, "failed to getrlimit\n");
        return;
    }

    rlim.rlim_cur = strtol(argv[0], NULL, 10);

    ret = setrlimit(RLIMIT_STACK, &rlim);
    if (ret < 0) {
        fprintf(stderr, "failed to setrlimit\n");
        return;
    }
}

struct rlimit_list {
    char *string;
    void (*get_callback)(void);
    void (*set_callback)(int n, char **rem_args);
} rlimit_list[] = {
    {"max_addr_space", get_max_addr_space, NULL},
    {"max_file_size", get_max_file_size, NULL},
    {"max_stack_size", get_max_stack_size, set_max_stack_size},
    {"max_cpu_time", get_max_cpu_time, NULL},
};

int main(int argc, char **argv)
{
    int i;

    if (!strcmp(argv[1], "get")) {
        for (i = 0; i < sizeof(rlimit_list) / sizeof(rlimit_list[0]); i ++) {
            if (!strcmp(rlimit_list[i].string, argv[2])) {
                rlimit_list[i].get_callback();
            }
        }
    } else if (!strcmp(argv[1], "set")) {
        for (i = 0; i < sizeof(rlimit_list) / sizeof(rlimit_list[0]); i ++) {
            if (!strcmp(rlimit_list[i].string, argv[2])) {
                if (rlimit_list[i].set_callback)
                    rlimit_list[i].set_callback(argc - 3, &argv[3]);
            }
```

```
        }
    }

    return 0;
}
```

Static and dynamic libraries

1. The static libraries are denoted with `.a` extension while the dynamic libraries are denoted with `.so` extension.
2. The static libraries, when linked they directly add the code into the resulting executable. Thus allowing the program to directly resolve the function references at the linker time. This also meaning that the program size is greatly increased.
3. The dynamic libraries, when linked they only add the symbol references and addresses where the symbol can be found. So that when the program is run on the target system, the symbols will be resolved at the target system (mostly by the ld loader). Thus, the dynamic library does not add any code to the resulting binary.
4. The dynamic library poses a problem with the un-resolved symbols when the program is run on the target system.
5. The Program binary versions can be changed or incremented irrespective with the dynamic library linkage as long as the dynamic library provides the same APIs to the user program. Thus introducing the modularity.

To create a shared library:

```
gcc -shared -o libshared.so -fPIC 1.c 2.c ..
```

when creating the shared library using the -fPIC is most important. The position independent operation allows the program to load the address at the different address.

To create a static library:

```
ar rcs libstatic.a 1.o 2.o
```

The libdl.so (Dynamic loading of functions)

The dynamic loading allows program to load a library at run time into the memory, retrieve the address of functions and variables, can perform actions and unload the library. This adds an extended functionality to the program and allows methods to inject code into the program.

dlopen: open the dynamic object

dlsym: obtain the address of a symbol from a dlopen object

dladdr: find the shared object containing a given address

include the header file `<dlfcn.h>` to use the dynamic library. The `dladdr` function prototype is as follows:

```
int dladdr(void *addr, Dl_info *dlip);
```

The `Dl_info` looks as the following:

```
typedef struct {
    const char *dli_fname;
    void       *dli_fbase;
    const char *dli_sname;
    void       *dli_saddr;
} Dl_info;
```

`dladdr` shall query the dynamic linker for information about the shared object containing the address `addr`. The information shall be returned in the user supplied data structure referenced by `dlip`.

The `dlopen` is defined as follows:

```
void *dlopen(const char *filename, int flags);
```

The `dlopen` function loads the dynamic library file referenced by `filename` and returns a handle pointer for the libbrary. The `flags` arguments tell the `dlopen` on the way to load the library. Usually `flags` are set to RTLD_LAZY. The RTLD_LAZY performs the lazy binding. This means that it only resolves the symbols when the code calls them.

The opened library handle is then used to get the function addresses with referencing to their names. This is done with `dlsym` function call.

The `dlsym` is defined as follows:

```
void *dlsym(void *handle, const char *symbol);
```

The `symbol` is the function name and `handle` is the return of `dlopen`. The `dlsym` returns the address of the function. It is then captured into a function pointer for further use.

Let us define a file named `lib.c` (You can also download it here (https://github.com/DevNaga/gists/blob/master/lib.c)). It is defined as follows.

```
lib.c:

#include <stdio.h>

int function_a()
{
    printf("function a is defined\n");
    return 0;
}
```

We then compile it as follows.

```
gcc -fPIC -o lib.c
```

The `lib.o` is generated with the above compiler command.

Let us define the code that perform the dynamic loading. (You can also download it here (https://github.com/DevNaga/gists/blob/master/dlopen.c))

```c
#include <stdio.h>
#include <dlfcn.h>

int func(void);

int main()
{
    int (*func)(void);
    void *dl_handle;

    dl_handle = dlopen("./libtest.so", RTLD_LAZY);
    if (!dl_handle) {
        fprintf(stderr, "no handle found \n");
        return -1;
    }

    func = dlsym(dl_handle, "function_a");
    printf("function %p\n", func);

    func();

    return 0;
}
```

We then compile the above program as the following:

```
gcc dlopen.c -ldl -rdynamic
```

The `ldl` and `rdynamic` are used to compile and link the above code.

When we execute the resultant `a.out` file it prints the following:

```
function 0x7f05b82d46e0
function a is defined
```

Makefiles

Makefile is a different language, is a group of instructions that when executed using the command `make` generates the output binaries.

The binary names, how to generate them, the optimizations are specified in the Makefile.

Makefiles are useful to organize and compile a large set of C, C++ or any other source file into different sets in different ways like customization.

The command `make` is available in the `automake` package of any linux distribution.

To install `make` on Ubuntu:

```
sudo apt-get install make automake
```

on Fedora (as root):

```
dnf install make automake
```

To compile our gist example hash.c using the makefile, we put it in a file called, `Makefile`.

`cat Makefile`

```
hash:
    gcc hash.c -o hash
```

a `make` command is issued in the same directory gives us the `hash` binary.

Now, we can add more things to the Makefile as the following.

```
all:
    gcc hash.c -o hash

clean:
    rm -rf hash.o hash
```

We have two subcommands now for the make argument. They are `all` and `clean`.

Issuing `make clean` would remove the binary `hash` and the object file `hash.o`.

Issuing `make all` would compile the binary and generate the `hash`.

Let us define the variable `CC` which can either be assigned with in the makefile to the C compiler `gcc` or passed via a command line argument. The command line argument is mostly used in cases where the compiler can be an architectural dependent compiler (in case of cross compilation) or a proprietary compiler.

The makefile now looks as below.

```
CC=gcc

all:
    $(CC) hash.c -o hash
clean:
    rm -rf hash.o hash
```

We can pass the `CC` variable via the command line as following `make CC=gcc`.

processes

Process is a binary program that is currently running in the linux system. It can be viewed by using the `ps` command.

A binary program can be created just as we create `a.out` files using the gcc compiler. The program when run, becomes the process.

Each process by default, gets three file descriptors that are attached to it. They are `stdin`, the standard input pipe, `stdout`, the standard output pipe, `stderr`, the standard error pipe.

Linux is a multi-process operating system. Each process gets an illusion that it has the access to the full hardware for itself. Each process is interleaved in execution using the concept called scheduling. The scheduling takes many parameters as input to run one of the scheduling algorithms. There are many scheduling algorithms such as roundrobin, FIFO etc. Each of these

scheduling algorithms can be selectable or just built-in directly into the Linux kernel that comes with the Linux OS types such as Fedora or Ubuntu. The scheduler is also a thread or a function that gets a periodic interrupt via the hardware timing pulse. At each pulse the scheduler finds out the next process that is eligible to run on the CPU(s). The scheduler then places the process into the Run queue and wakes it up and places the already running process into wait queue and saves all of the registers that it used and etc. The woken up process will then execute on the CPU. The scheduling is a vast and a large topic to cover and it is only explained here in this paragraph brief.

In linux `stdin` is described as file descriptor 0, `stdout` is described as file descriptor 1, and `stderr` is described as file descriptor 2. Upon the each successful pipe / msgqueue / socket call, the file descriptors will get incremented in number usually (i.e from 3, 4 and so on) most of the time. A file descriptor that is opened must always be closed with a corresponding `close` system call or the similar one. This is a good programming practice.

Each process is identified by a pid (called process identifier). Process ids are 16 bit numbers (Signed numbers). The program can obtain its pid using `getpid()` call. The parent process id is obtained by using `getppid()`.

NOTE: In linux the process 1 is always the init process.

The below program gets the pid and parent pid.

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

int main(void)
{
    printf("process id %d\n", getpid());
    printf("parent process id %d\n", getppid());

    return 0;
}
```

The parent process of the program that we have currently executed is always the shell that we ran the binary.

The below example is for the `ps` command.

```
  PID TTY          TIME CMD
22963 ttys000    0:00.29 -bash
22860 ttys001    0:00.01 -bash
22863 ttys002    0:00.02 -bash
```

more detailed output can be obtained about the current processes can be done using the **-e** option `ps -e`.

```
devnaga@Hanzo:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:02 systemd
    2 ?        00:00:00 kthreadd
    4 ?        00:00:00 kworker/0:0H
    6 ?        00:00:00 mm_percpu_wq
    7 ?        00:00:00 ksoftirqd/0
    8 ?        00:00:06 rcu_sched
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:00 migration/0
   11 ?        00:00:00 watchdog/0
   12 ?        00:00:00 cpuhp/0
   13 ?        00:00:00 kdevtmpfs
   14 ?        00:00:00 netns
   15 ?        00:00:00 khungtaskd
   16 ?        00:00:00 oom_reaper
   17 ?        00:00:00 writeback
   18 ?        00:00:00 kcompactd0
   19 ?        00:00:00 ksmd
   20 ?        00:00:00 khugepaged
   21 ?        00:00:00 crypto
   22 ?        00:00:00 kintegrityd
   23 ?        00:00:00 kblockd
   24 ?        00:00:00 ata_sff
   25 ?        00:00:00 md
   26 ?        00:00:00 edac-poller
   27 ?        00:00:00 devfreq_wq
   28 ?        00:00:00 watchdogd
   30 ?        00:00:16 kworker/0:1
```

The ps is detailed much more in the manual pages (`man ps`). Eitherway, the most useful command is `ps -e`.

pid 0 (and is never shown in the ps command output) is a swapper process and pid 1 is an init process. The init process is the one that is called by the kernel to initialise the userspace.

A process has the following states:

Running - The process is either running or is ready to run.

Waiting - The process is waiting for an event or a resource. Such as socket wait for the recv etc.

Stopped - The process has been stopped with a signal such as Ctrl + Z combination or under a debugger.

When a process is created, the kernel allocates enough resources and stores the information about the process. The kernel also creates an entry about the process in the `/proc` file system. For ex: for a process such as `init` with pid 1, it stores the name of the process into `/proc/<pidname>/cmdline` file, and stores links to the opened files in `/proc/<pidname>/fd/`

directory and it keeps a lot of other information such as memory usage by this process, scheduling statistics etc. When a process is stopped, the information and all the allocated memory and resources will then automatically be freed up by the kernel.

setsid system call

getpriority and setpriority system calls

scheduling system calls

enviornmental variables

The environmental variables allow the system to get some of the items for use in the program.

Environmental variables can be found with the use of `getenv` function call.

Below is an example of `getenv`. Download here (https://github.com/DevNaga/gists/blob/master/getenv.c)

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

int main()
{
    char *path_ = getenv("PATH");
    if (!path_) {
        printf("PATH variable can't be found\n");
    } else {
        printf("PATH %s\n", path_);
    }

    return 0;
}
```

environmental variables are found as well by using the command env in the shell.

for example to define an enviornmental variable within the current shell, use `export` command.

something like the below,

```
export TEST_C_ENV="hello"
```

and doing `env | grep TEST_C_ENV gives,

```
env | grep TEST
TEST_C_ENV=hello
```

with the above program having PATH replaced with TEST_C_ENV, it looks below..

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

int main()
{
    char *path_ = getenv("TEST_C_ENV");
    if (!path_) {
        printf("TEST_C_ENV variable can't be found\n");
    } else {
        printf("TEST_C_ENV %s\n", path_);
    }

    return 0;
}
```

All the export commands done on this variable only apply to this shell, and if a new shell is created the variable however will not be present.

any further export calls to TEST_C_ENV as,

```
export TEST_C_ENV="hello1"
```

will replace the existing value with new value. That means it gets overwritten.

to append the value to the TEST_C_ENV, the value must preceed with the variable itself with a `:`.

```
TEST_C_ENV=$TEST_C_ENV:test1
```

to unset an environmental variable, use `unset`. calling unset on TEST_C_ENV as

```
unset TEST_C_ENV
```

running the code to get TEST_C_ENV again, gives,

```
./a.out
TEST_C_ENV variable can't be found
```

Alternatively, the program can set the environmental variables using `putenv`. Below is an example of `putenv` system call, Download here (https://github.com/DevNaga/gists/blob/master/putenv.c)

The prototype of `putenv` is as follow,

```c
int putenv(const char *var_val);
```

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

int main(int argc, char **argv)
{
    int i = 0;
    char var[20];
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> var=val\n", argv[0]);
        return -1;
    }

    ret = putenv(argv[1]);
    if (ret < 0) {
        return -1;
    }

    printf("Set %s\n", argv[1]);

    memset(var, 0, sizeof(var));

    while (argv[1][i] != '=') {
        if (i >= sizeof(var) - 1) {
            break;
        }

        var[i] = argv[1][i];
        i ++;
    }

    var[i] = '\0';

    printf("get %s=%s\n", var, getenv(var));

    return 0;
}
```

Another function is setenv and the prototype is as follows.

```c
int setenv(const char *variable, const char *value, int override);
```

This simply sets the variable to value to the environment. If `override` is set to 1, then the existing value is overwritten. If there is no variable and `override` is set to 0, then new variable is created.

Below is an example of setenv. Download [here](https://github.com/DevNaga/gists/blob/master/setenv.c)

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

int main(int argc, char **argv)
{
    int i = 0;
    char var[20];
    int ret;

    if (argc != 3) {
        fprintf(stderr, "<%s> var val\n", argv[0]);
        return -1;
    }

    ret = setenv(argv[1], argv[2], 1);
    if (ret < 0) {
        return -1;
    }

    printf("Set %s\n", argv[2]);


    printf("get %s=%s\n", argv[1], getenv(argv[1]));

    return 0;
}
```

users and groups

The password file `/etc/passwd` contain details about each user with name, password (not in clear text) and UID, GID and current working directory.

The system call `getpwnam` is used to get the information for each user.

The `getpwnam` returns what is called `struct passwd`. It looks like the following,

```c
struct passwd {
    char *pw_name; // name of the user
    char *pw_passwd; // passwd
    uid_t pw_uid;
    gid_t pw_gid;
    char *pw_gecos; // user information
    char *pw_dir; // home directory
    char *pw_shell; // shell program
}
```

The prototype of `getpwnam` looks like below,

```c
struct passwd * getpwnam(const char *name);
```

Below is an example of `getpwnam`. Download here (https://github.com/DevNaga/gists/blob/master/getpass.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>

int main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "<%s> username\n", argv[0]);
        return -1;
    }

    struct passwd *pw;

    pw = getpwnam(argv[1]);
    if (!pw) {
        return -1;
    }

    printf("name %s\n", pw->pw_name);
    printf("password %s\n", pw->pw_passwd);
    printf("uid %d\n", pw->pw_uid);
    printf("gid %d\n", pw->pw_gid);
    printf("user informatino %s\n", pw->pw_gecos);
    printf("home %s\n", pw->pw_dir);
    printf("shell %s\n", pw->pw_shell);

    return 0;
}
```

fork

The system call fork is used to create processes. The most important realworld use of fork is in the daemon processes where in which the daemon has to get attached to the pid 1 (the init process).

The process that calls fork and creates another process is called a parent process and the process that is created by the fork system call is called child process.

Fork returns twice unlike any other function in C. The fork call is executed by the kernel and it returns once in the parent process and once in the child process.

Fork returns the pid of the child process in the parent, and returns 0 in the child. If creation of a process was unsuccessful then it returns -1.

The below program gives an example of `fork` system call. Download here (https://github.com/DevNaga/gists/blob/master/fork.c)

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

int main()
{
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        printf("failed to fork..\n");
    } else if (pid == 0) {
        printf("child process ..\n");
    } else if (pid > 0) {
        printf("parent process..\n");
    }

    sleep(2);
    return 0;
}
```

Example: demonstration of fork system call

notice that when you run the above program several times, the print statements "parent process" and "child process" never come exactly one after the another for ex: parent first and then the child. This is because of the scheduling job that is done internally in the kernel.

After a child process is created the parent can continue its execution of the remaining code or can exit by calling the `exit` system call. When the parent exits before the child could run, the child becomes an orphan. This process is then parented by the init process (pid 1). This method is one of the step in creating the system daemon.

In an other case where in the child gets stopped or exited before the parent. In this case, the parent must cleanup the resources that are allocated to the child by calling one of the functions `wait`, `waitpid`. In some cases, when the parent process does not perform the task of cleanup, the child process will then go into a state called zombie state. In the zombie state, the child process although is not running, its memory is never get cleaned up by the parent process. Notice the "Z" symbol in the `ps -aux` command.

The system call `getpid` returns the process id of the process. When applied to the above example code, prints out the process ids of both parent and child.

Below is an example of `getpid`. Download [here](https://github.com/DevNaga/gists/blob/master/getpid.c) (https://github.com/DevNaga/gists/blob/master/getpid.c)

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

int main()
{
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        printf("failed to fork..\n");
    } else if (pid == 0) {
        printf("child process .. %d\n", getpid());
    } else if (pid > 0) {
        printf("parent process.. %d\n", getpid());
    }

    sleep(2);
    return 0;
}
```

fork system call is mostly used in the creation of system daemons.

wait

The system call `wait` lets the parent process wait for the child process to finish.

The prototype of the `wait` systel call is below:

```c
pid_t wait(int *status);
```

The `status` is the status of the child process that is received when child process finishes its execution. The wait returns the pid of the child process. There are more than one status codes ORed together with in the `status`.

1. The `WIFEXITED` macro describe the exit status of the child. Use `WIFEXITED` macro with the `status` variable and then exit status is found with `WEXITSTATUS` macro.

Below is the example of the `wait` system call. Download [here (https://github.com/DevNaga/gists/blob/master/fork_wait.c)](https://github.com/DevNaga/gists/blob/master/fork_wait.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;

    pid = fork();
    if (pid < 0) {
        printf("failed to fork..\n");
    } else if (pid == 0) {
        printf("child process ..\n");
        int i;

        double f = 1.0;
        for (i = 0; i < 10000000; i ++) {
            f *= i;
            f /= i;
        }
        //uncomment the below line and observe the behavior of the value at
WEXITSTATUS in the parent section
        //exit(1);
    } else if (pid > 0) {
        printf("parent process..\n");
        printf("wait for child to complete the task..\n");

        int status = 0;

        pid_t pid_ = wait(&status);
        if (pid_ == pid) { // child pid
            if (WIFEXITED(status)) {
                printf("child terminated normally..\n");
                printf("exit %d\n", WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("signal on child\n");
                printf("term signal %d on child \n", WTERMSIG(status));
                printf("coredump %d\n", WCOREDUMP(status));
            } else if (WIFSTOPPED(status)) {
                printf("child stopped by delivery of signal\n");
                printf("stop signal %d\n", WSTOPSIG(status));
            }
        }
    }

    return 0;
}
```

in the above program, the parent process is in the parent portion of the code, the code we handle using the `pid > 0` check. Within which the exit statuses are printed at each execution stage. The program simply adds a busy maths loop to delay the execution of the child process.

The exit status from the child is 0 on a normal condition. When the code below the child having the `exit` library function called with exit status of 1, the parent process receives the exit status as the `wait` system call output.

The status is then collected by checking `WIFEXITED` macro and with the `WEXITSTATUS` macro.

the status field is checked upon with the following macros.

macro	meaning
`WIFEXITED`	returns true if the child terminated normally
`WEXITSTATUS`	returns the exit status of the child process, use it only when `WIFEXTITED` set to to true
`WIFSIGNALED`	returns true if child process is terminated by a signal
`WTERMSIG`	return signal number. use it if `WIFSIGNALED` returns true
`WCOREDUMP`	returns true if child produced core dump. use it when `WIFSIGNALEd` returns true
`WIFSTOPPED`	returns true if child process stopped by delivery of the signal
`WSTOPSIG`	return signal number.. use it when `WIFSTOPPED` returns true

waitpid

the `waitpid` system call has the below definition.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

the `pid` argument is of type pid of the child process. if set to -1, the `waitpid` will wait indefinitely for any child process.

The status of the child process is copied into the `status`. Status is similar to the `wait` status values.

the options contain the folowing values. OR combination of one or more of the below values

value	meaning
`WNOHANG`	return immediately if no child has exited
`WUNTRACED`	return if a child has stopped
`WCONTIUED`	return if a stopped child has continued after delivery of SIGCONT

waiting for all the child processes

waiting for multiple child processes can be done using the `wait` system call.

Below is the example. Download [here (https://github.com/DevNaga/gists/blob/master/allchild.c)](https://github.com/DevNaga/gists/blob/master/allchild.c)

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    int ret;
    int i;

    for (i = 0; i < 10; i ++) {
        pid = fork();
        if (pid != 0) {
        } else {
            printf("starting %d\n", getpid());
            _exit(1);
        }
    }

    for (i = 0; i < 10; i ++) {
            pid_t child = wait(NULL);
            printf("reap %d\n", child);
    }

    return 0;
}
```

typically, in larger programs, the child processes does not expect to die soon often. sometimes, it may happen to be that the child process is died because of unexpected reasons, and thus may require a restart. Before restarting, the child process must be cleaned by using the `wait` or `waitpid` system call.

There is another way to do the same waiting. by creating pipes.

below is the procedure:

```
for each child a  pipe is created, the parent closes the write end of the pipe
and the child closes the read end of the pipe.

parent then sets the read end of the file descriptor to the `fd_set` for each
children.

parent waits in `select` system call to monitor the `fd_set`.

if any children die, reported as a close of socket that is write end on the
child end, since parent is on the `read` end, the `select` will wake up causing
the parent to call `waitpid`.
```

Below is the source code for the above algorithm. Download [here](https://github.com/DevNaga/gists/blob/master/selfpipe.c)

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/select.h>

int maxfd_all(int pipefd[10][2], int len)
{
    int maxfd = 0;
    int i;

    for (i = 0; i < len; i ++) {
        if (maxfd < pipefd[i][0])
            maxfd = pipefd[i][0];
    }

    return maxfd;
}

int main()
{
    pid_t pid;
    int i;
    int ret;
    int pipefd[10][2];
    fd_set allfd;
    fd_set rdset;
    int maxfd = 0;

    FD_ZERO(&rdset);

    for (i = 0; i < 10; i ++) {
        ret = pipe(pipefd[i]);

        printf("Read end %d max fd %d\n", pipefd[i][0], maxfd);
        FD_SET(pipefd[i][0], &rdset);
        if (maxfd < pipefd[i][0])
            maxfd = pipefd[i][0];

        pid = fork();
        if (pid != 0) {
            close(pipefd[i][1]);
        } else {
            close(pipefd[i][0]);
            for (i = 0; i < 100000000; i ++) ;
            sleep(1);
            _exit(1);
        }
    }

    while (1) {
        allfd = rdset;
```

```
        maxfd = maxfd_all(pipefd, 10);

        if (maxfd <= 0) {
            break;
        }

        ret = select(maxfd + 1, &allfd, 0, 0, NULL);
        if (ret < 0) {
            perror("select");
            return -1;
        } else {
            for (i = 0; i < 10; i ++) {
                if (FD_ISSET(pipefd[i][0], &rdset)) {
                    printf("reap child.. %d \n", pipefd[i][0]);
                    waitpid(-1, NULL, WNOHANG);
                    //close(pipefd[i][0]);
                    FD_CLR(pipefd[i][0], &rdset);
                    pipefd[i][0] = -1;
                }
            }
        }
    }

    return 0;
}
```

[self pipe trick (http://cr.yp.to/docs/selfpipe.html)]

signals

Introduction

Signals are used to notify a process about some event. The event condition may be for ex: packet received on a network interface, packet sent on a network interface, watchdog timer expiry, floating point exception, invalid memory address read / write (segfault, alignment fault) etc. The linux provides 64 different signals range from `SIGHUP` to `SIGRTMAX`. The normal signals range from `SIGHUP` to `SIGSYS` and the real time signals start from `SIGRTMIN` to `SIGRTMAX`.

The command `kill -l` on the bash would give us the following.

SIGHUP	SIGINT	SIGQUIT	SIGILL	SIGTRAP
SIGABRT	SIGBUS	SIGFPE	SIGKILL	SIGUSR1
SIGSEGV	SIGUSR2	SIGPIPE	SIGALRM	SIGTERM
SIGSTKFLT	SIGCHLD	SIGCONT	SIGSTOP	SIGTSTP
SIGTTIN	SIGTTOU	SIGURG	SIGXCPU	SIGXFSZ
SIGVTARLM	SIGPROF	SIGWINCH	SIGIO	SIGPWR
SIGSYS	SIGRTMIN	SIGRTMIN + 1	SIGRTMIN + 2	SIGRTMIN + 3
SIGRTMIN + 4	SIGRTMIN + 5	SIGRMIN + 6	SIGRTMIN + 7	SIGRTMIN + 8
SIGRTMIN + 9	SIGRTMIN + 10	SIGRTMIN + 11	SIGRTMIN + 12	SIGRTMIN + 13

SIGRTMIN + 14 SIGRTMIN + 15 SIGRTMAX - 14 SIGRTMAX - 13 SIGRTMAX - 12
SIGRTMAX - 11 SIGRTMAX - 10 SIGRTMAX - 9 SIGRTMAX - 8 SIGRTMAX - 6
SIGRTMAX - 6 SIGRTMAX - 5 SIGRTMAX - 4 SIGRTMAX - 3 SIGRTMAX - 2
SIGRTMAX - 1 SIGRTMAX

The `SIGINT` and `SIGQUIT` are familiar to us as from the keyboard as we usually perform the `ctrl + c` (`SIGINT`) or `ctrl + \` (`SIGQUIT`) combination to stop a program.

Signals are also delivered to a process (running or stopped or waiting) with the help of `kill` command. The manual page (`man kill`) of kill command says that the default and easier version of kill command is the `kill pid`. Where `pid` is the process ID that is found via the `ps` command. The default signal is `SIGTERM (15)`. Alternatively a signal number is specified to the `kill` command such as `kill -2 1291` making a delivery of `SIGINT(2)` signal to the process ID `1291`.

The linux also provide a mechanism for sysadmins to control the processes via two powerful and unmaskable signals `SIGKILL` and `SIGSTOP`. They are fatal and the program terminates as soon as it receives them. This allows admins to kill the offending or bad processes from hogging the resources.

The linux also provide us a set of system calls API to use the signals that are delivered to the process. Some of the most important API are `sigaction` and `signal`. Lately `signalfd` is introduced that delivers the signals in a synchronous fashion to safely control the signals that occur at unknown or surprisingly.

handling of the signals:

1. ignore the signal
2. perform the handler function execution
3. default action

Signals are asynchornous and must be handled. So the system call interface provides an API to handle the signals. Below described are some of the system calls.

signal and sigaction

`signal` is a system call, that allows the program to handle the signal when it occurs.

prototype:

```
sighandler_t signal(int signum, sighandler_t handler);
```

Sigaction

Key differences

A very good [stackoverflow question (http://stackoverflow.com/questions/231912/what-is-the-difference-between-sigaction-and-signal)](http://stackoverflow.com/questions/231912/what-is-the-difference-between-sigaction-and-signal) here tells us why `sigaction` is better than `signal`. The key differences are:

1. the `signal()` does not necessarily block other signals from arriving while the current signal is being executed. Thus when more than one signal occur at the same time, it becomes more problematic to understand and perform actions. If it is on the same

data, this might even get more complex. The `sigaction()` blocks the other signals while the handler is being executed.

2. As soon as the `signal()` handler is executed, the `signal()` sets the default handler to `SIG_DFL` which may be `SIGINT / SIGTERM / SIGQUIT` and the handler must reset the function back to itself so that when the signal occur again, the signal can be handled back. However, with the signal() allowing the handler to be called even though the handler is already executing, this will implicate a serious problem where in which the small window between the register and default would let the program go into `SIG_DFL`.

sigwaitinfo

sigtimedwait

sigwait

signalfd

signalfd is a new system call introduced by the linux kernel. The signalfd system call returns a file descriptor associated with the signal. This file descriptor is then used to wait on the select, poll or epoll system calls. Unlike the `signal` or `sigaction` the signals are queued in the socket buffer and can be read on the socket.

The prototype of the `signalfd` is as follows.

```
int signalfd(int fd, const sigset_t *mask, int flags);
```

To use the `signalfd` one must include `<sys/signalfd.h>`.

If the fd argument is -1, the `signalfd` returns a new file descriptor. If fd argument is not -1, then the fd that is returned from previous calls to the `signalfd` must be given as an argument.

The mask argument is similar to the one that we pass to the `sigprocmask` system call. This allows the `signalfd` to create an `fd` out for the mask. As the mask is created, the signals in the mask should be blocked with the `sigprocmask`. This allows the correct functionality of the `signalfd`.

The `flags` argument is usually set to 0. It is much similar to the O_NONBLOCK flag options of other system calls.

Include `<sys/signalfd.h>` to use the `signalfd` system call.

Below is an example of the `signalfd` system call. Download [here (https://github.com/DevNaga/gists/blob/master/sigfd.c)](https://github.com/DevNaga/gists/blob/master/sigfd.c)

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/signalfd.h>
#include <string.h>

int main(int argc, char **argv)
{
    int sfd;
    sigset_t mask;
```

```
    sigemptyset(&mask);
    sigaddset(&mask, SIGQUIT);
    sigaddset(&mask, SIGINT);

    if (sigprocmask(SIG_BLOCK, &mask, NULL) < 0) {
        printf("afiled to sigprocmask\n");
        return -1;
    }

    sfd = signalfd(-1, &mask, 0);
    if (sfd < 0) {
        printf("failed to get signalfd\n");
        return -1;
    }

    while (1) {
        struct signalfd_siginfo sf;
        int ret;

        memset(&sf, 0, sizeof(sf));
        ret = read(sfd, &sf, sizeof(sf));
        if (ret != sizeof(sf)) {
            printf("invalid length of siginfo %d received\n", ret);
            return -1;
        }

        printf("pid %d signal value: %d signal code: %d\n",
                        sf.ssi_pid,
                        sf.ssi_signo,
                        sf.ssi_code);

        if (sf.ssi_signo == SIGQUIT) {
            printf("received termination signal\n");
        } else if (sf.ssi_signo == SIGINT) {
            printf("received interrupt\n");
        } else {
            printf("invalid signal %d\n", sf.ssi_signo);
        }
    }

    close(sfd);

    return 0;
}
```

The signals are first masked by the `sigaddset` system call and then blocked with `sigprocmask`. The mask is then given to the `signalfd` system call. The signals are then queued to the socket fd returned. This can be waited upon the `read` or `select` system call.

The kernel returns a variable of the form `signalfd_siginfo` upon read.The structure then contains the signal that is occured. Here's some contents in the `signalfd_siginfo`.

```
struct signalfd_siginfo {
    uint32_t ssi_signo;
    ...
    uint32_t ssi_pid;
    ....
};
```

The `ssi_signo` contains the signal number that is occured. The `ssi_pid` is the process that sent this signal.

Example: signalfd basic example

sigaddset

`sigaddset` adds the signal to a set. The prototype is as follows.

```
int sigaddset(sigset_t *set, int signo);
```

The `signo` gets added to the `set`. Multiple calls of the `sigaddset` on the same `set` would add the signals to the set. The function is mostly used in generating a signal mask for the `sigprocmask`. See more about `sigprocmask` in the below sections.

sigfillset

`sigfillset` initializes the `set` to full, including all the signals. The prototype is as follows.

```
int sigfillset(sigset_t *set);
```

sigemptyset

sigismember

sigdelset

sigemptyset

sigprocmask

waiting for child processes

When a child process stops the parent process must reap it to prevent it from becoming a zombie. The zombie meaning that the process is not taken out from process table but they dont execute and do not utilize the memory or CPU.

When a parent process stops before the child stops, then the child process is called an `orphan` process. Often some process adapts the child process and becomes its parent. This process often is the `init` process.

When a parent waits for the child process by some means and reaps it, the zombie processes will not happen.

Waiting for a child process can be performed in the following ways.

1. calling `wait` and finding the child's pid as its return value.
2. calling `waitpid` on a specific child.

wait

The `wait` system call suspends the execution of the parent process until one of its child processes terminates. On success returns the process ID of the child and -1 on failure.

The prototype is as follows.

```
pid_t wait(int *status);
```

waitpid

The `waitpid` system call suspends the execution of the parent process until a child process specified by the `pid` argument has changed state.

The prototype is as follows.

```
pid_t waitpid(pid_t pid, int *status, int options);
```

By default, the `waitpid` waits only for the terminated children. With the `options` argument the behavior is changed.

pid value	meaning
-1	wait for any child process
> 0	wait for the child with the given process ID

Most of the time the `options` argument is either 0 or `WNOHANG`. With `WNOHANG` the `waitpid` returns immediately if there is no child exits.

The `status` value is kept into the `waitpid` if the passed argument of `status` is non `NULL`. The value can be interpreted with the following macros.

exit status	description
`WIFEXITED(status)`	returns true if the child is exited normally
`WEXITSTATUS(status)`	returns the exit status of the child. call this only if `WIFEXITED(status)` returns true

file operations

1. C file handling

The function call set `fopen`, `fclose`, `fgets`, `fputs`, `fread` and `fwrite` can be used to perform basic to intermediate file handling.

The FILE is the handle returned by the `fopen` call.

the `fopen` takes the following form:

```
FILE *fopen(const char *file, char *mode);
```

the fopen returns the pointer of type FILE. This also is called as file handle. The mode argument is one of "r", "w", "a" etc. The "r" argument is used to perform "read" on the file, the "w" argument is used to perform "write" on the file and the "a" argument is used to perform "append" on the file.

```
FILE *fp;

fp = fopen("file", "w");
```

the `fopen` returns `NULL` if the file can't be opened.

The opened file can be used to perform operations on the file. The `fgets` is used to perform the read operation and `fputs` is used to perform the write operation on the file respectively.

Contents of the file can be read character by character using the `fgetc` and can be written character by character using the `fputc` call.

The `stdin`, `stdout` and `stderr` are the standard file descriptors for the standard input stream, output stream and error stream.

A call to `fgets` will read the contents of the string into the buffer of given length terminated by the '\n\0'.

So when performing `fgets`, the buffer should be stripped with `\n`. such as the following

```
buf[strlen(buf) - 1] = '\0'; // terminate the new line with '\0' character.
```

The below example reads the file given as argument from the command line and prints the contents on to the screen.

```
#include <stdio.h>

#define LINE_LEN 512

int main(int argc, char **argv)
{
    char buf[LINE_LEN];
    FILE *fp;

    fp = fopen(argv[1], "r");
    if (!fp) {
        fprintf(stderr, "failed to open %s for reading\n", argv[1]);
        return -1;
    }

    while (fgets(buf, sizeof(buf), fp)) {
        fprintf(stderr, "%s", buf);
    }

    fclose(fp);
}
```

Example: file system reading

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    FILE *fp;

    fp = fopen(argv[1], "r");
    if (!fp) {
        fprintf(stderr, "cannot open file %s for reading\n", argv[1]);
        return -1;
    }

    do {
        int ch;

        ch = fgetc(fp);
        if (ch == EOF)
            break;

        printf("%c", ch);
    } while (1);

    fclose(fp);

    return 0;
}
```

Example: file read character by character

```c
#include <stdio.h>

#define LINE_LEN 512

int main(int argc, char **argv)
{
    FILE *fp;
    int ret;
    char buf[LINE_LEN];

    fp = fopen(argv[1], "w");
    if (!fp) {
        fprintf(stderr, "failed to open %s for writing\n", argv[1]);
        return -1;
    }

    do {
        fgets(buf, sizeof(buf), stdin);
        if (buf[0] == '\n')
            break;
        fputs(fp, buf);
    } while (1);

    fclose(fp);

    return 0;
}
```

Example: basic file writing

###2. OS file handling

The system calls open, `close`, read and `write` perform the file system handling in the Linux.

Underlying kernel buffers the bytes that are written to or read from the file. The buffering is of kernel page size (8k or 4k) and only if the buffer size is over written, the kernel writes the contents of the file to the disk. When reading, the kernel reads the file contents before hand, and buffers them so that future reads will only happen from the buffer but not from the file (this is to save extra operations of the file to disk).

the open system call returns a file descriptor. The file descriptor is used as a handle in the kernel, to map to a specific portions such as inode and then to the file etc. The open system call prototype is as follows.

```c
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

file mode permissions	description
O_CREAT	create file
O_RDWR	open in read write mode
O_RDONLY	read only
O_WRONLY	write only

O_APPEND	append only
O_EXCL	exclusive operation on the file when used with O_CREAT

The two prototypes of the open system tells that its a variable argument function.

The first prototype is used when opening a file in read/write mode. The second prototype is used when opening a new file and that's where the mode comes into picture.

Opening a file in read/write mode would look as below.

```
int file_desc;

file_desc = open(filename, O_RDWR);
```

The O_RDWR tells the kernel that the file must be opened in read and write mode.

Opening a new file would look as below.

```
int file_desc;

file_desc = open(filename, O_RDWR | O_CREAT, S_IRWXU);
```

The O_CREAT tells the kernel that the file must be created and the S_IRWXU means that the file must have read (R), write (W) and executable (X) permissions.

The open system call on success returns a file descriptor to the file. Using this, the read or write operations can be performed on the file. On failure, the open returns -1 and sets the error number, indicating the cause of failure. The most possible failure cases can be that the permissions to open a file in the directory (EACCESS), too large filename (ENAMETOOLONG), or invlaid filename pointer.

Below example demonstrates the open system call with the O_EXCL feature. Download here (https://github.com/DevNaga/gists/blob/master/open_excl.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int fd;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_CREAT | O_EXCL | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        fprintf(stderr, "failed to open file error: %s\n", strerror(errno));
        return -1;
    }

    fprintf(stderr, "opened file %s fd %d\n", argv[1], fd);

    close(fd);

    return 0;
}
```

The above example, the `O_CREAT` is used to create a file, the `O_EXCL` is used to check if the file exist, if it does, then the file open will fail. if the file does not exist, the file open succeeds and a file descriptor `fd` is returned.

when the program is run the following way:

```
./a.out open_excl
opened file open_excl fd 3

./a.out open_excl
failed to open file error: File exists
```

the first execution, `./a.out open_excl` returns a file descriptor because the file `open_excl` is not available. The same command when executed one more time, the failure happens with an error of `File exists`.

The `O_EXCL` is used to validate the existence of the file and only creates if it does not exist.

The flags `S_IRUSR` and `S_IWUSR` are permission bits on the file. They are described more in the below sections.

the below example describe the O_APPEND flag of the open system call. Download here (https://github.com/DevNaga/gists/blob/master/open_excl.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int fd;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_WRONLY | O_APPEND);
    if (fd < 0) {
        perror("");
        return -1;
    }

    while (1) {
        char bytes[1000];
        int ret;

        fprintf(stderr, "enter something to send to the file %s\n", argv[1]);

        ret = read(1, bytes, sizeof(bytes));
        if (ret <= 0) {
            break;
        }

        if (bytes[0] == '\n') {
            break;
        }

        write(fd, bytes, ret);
    }

    close(fd);

    return 0;
}
```

the above example opens a file in append mode for writing, taking the input from the `fd` 1 that is `stdin` and writing the output to the file descriptor opened. The program keep writing to the file unless the read returns -1 or 0 or the user pressed a new line `\n`, this checked in the first byte and the loop breaks.

Always the file is opened in append mode, no matter how many times the program has run, so the contents will be appended to the same file. Observe that the file open is performed without creating it. So the program expects that some file is already there.

The `read` and `write` operations on the file are performed using the file descriptor returned by the `open`.

The `read` system call prototype is defined as follows.

```
size_t read(int fd, void *buf, size_t count);
```

the `read` system call returns the numbers of bytes read from the file descriptor `fd`. The `buf` is an input buffer passed to `read` and must be a valid pointer. the call fills in the buffer with the specified `count` bytes.

return `errno` values:

1. EBADF - bad fd
2. EFAULT - invalid buf pointer
3. EINTR - signal occured while reading

some of the common errors include:

1. If the file has read shorter than count bytes but more than 0 bytes, then it may be because the end of the file might have been reached.
2. The `read` system call returns 0 if the end of the file is reached and might return -1 if the calling process does not have permissions to read from the file descriptor `fd` or that the `fd` is not a valid `fd`.

A close of the file descriptor can also have the same effect of `read` returning 0, this is described later in the chapters.

the `write` system call prototype is defined as follows.

```
size_t write(int fd, const void *buf, size_t count);
```

the `write` system call returns the number of bytes written to the file pointed by `fd` from the `buf` of `count` bytes. Writes normally does not happen directly to the underlying disk, but happens at delayed intervals when the i/o is ready. The kernel caches the writes and at suitable times (when there is an i/o with definitive cache size) writes the buffers to the disk.

return `errno` values:

1. EBADF - bad fd
2. EFAULT - invalid buf
3. EPERM - insufficient privileges

some of the most common errors include:

1. disk is full
2. bad fd (someone closed it / programmer error)
3. no permissions to write to the fd

the write returns number of bytes written on success and -1 on error.

The `close` system call closes the file descriptor associated with the file. This is to be called when we are finished all the operations on the file.

the prototype of `close` system call is as follows

```
int close(int fd);
```

return `errno` values:

1. EBADF - bad fd

`close` system call returns 0 on success -1 on failure.

The numbers 0, 1 and 2 are for the standard input (stdin), standard output (stdout) and standard error (stderr).

The below examples give a basic idea about the file system calls under the Linux OS. Download here (https://github.com/DevNaga/gists/blob/master/open.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define LINE_LEN 512

int main(int argc, char **argv)
{
    int fd;
    int ret;
    char buf[LINE_LEN];

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s for reading\n", argv[1]);
        return -1;
    }

    do {
        ret = read(fd, buf, sizeof(buf));
        if (ret <= 0)
            break;
        write(2, buf, ret);
    } while (1);

    close(fd);

    return 0;
}
```

The above example calls open in read only (The `O_RDONLY` flag) mode. If the file is not found, then the open fails and returns -1 and thus printing the failed to open file error on the console.

If the file is opened successfully, the `read` operation is called upon the file descriptor and the `read` calls returns number of bytes read from the file. At each read, the file contents on printed on the screen's `stderr` file descriptor that is 2, using `write` system call.

Once the read completes, it either returns -1 or 0. This condition is checked upon always, The loop breaks and the file is closed.

Example: basic file read via the OS calls

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define LINE_LEN 512

int main(int argc, char **argv)
{
    int fd;
    int ret;
    char buf[LINE_LEN];

    fd = open(argv[1], O_CREAT | O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s for writing\n",
                                argv[1]);
        return -1;
    }

    do {
        ret = read(0, buf, sizeof(buf));
        if (ret <= 0) {
            break;
        }

        if (buf[0] == '\n') {
            break;
        }

        write(fd, buf, ret);
    } while (1);


    close(fd);

    return 0;
}
```

just like the `fwrite` and `fread`, a series of data structures or binary data can be written to the file directly using the `read` and `write` system calls. Below is one example of how to do. Download [here (https://github.com/DevNaga/gists/blob/master/write_bins.c)](https://github.com/DevNaga/gists/blob/master/write_bins.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```c
#include <errno.h>
#include <string.h>

struct bin_data {
    int a;
    double b;
} __attribute__ ((__packed__));

int main(int argc, char **argv)
{
    int fd;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s\n", strerror(errno));
        return -1;
    }

    struct bin_data b[10];
    int i;

    for (i = 0; i < 10; i ++) {
        b[i].a  = i;
        b[i].b = i + 4.4;
    };

    write(fd, b, sizeof(b));

    close(fd);

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "failed to open file %s\n", strerror(errno));
        return -1;
    }

    struct bin_data d[10];

    read(fd, d, sizeof(d));

    for (i = 0; i < 10; i ++) {
        fprintf(stderr, "a %d b %f\n", d[i].a, d[i].b);
    }

    close(fd);

    return 0;
```

```
}
```

in the above example, the file is created and a data structure is then setup and writes are made to the file. The file is then re-opened in read only and the same datastructures read back with the same size. The contents are then printed for a proof validity that the data structures infact are valid.

Example: Basic file write via OS calls

Practical example: File copy program

Here we are going to write a file copy program using the `open`, `read` and `write` system calls.

A file copy program takes a source filename on the command line and a destination file to which the source file contents to be copied.

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int fd1, fd2;
    int ret;

    if (argc != 3) {
        fprintf(stderr, "%s <source file> <destination file>\n", argv[0]);
        return -1;
    }

    fd1 = open(argv[1], O_RDONLY);
    if (fd1 < 0) {
        fprintf(stderr, "failed to open %s for reading\n", argv[1]);
        return -1;
    }

    fd2 = open(argv[2], O_CREAT | O_RDWR, S_IRWXU);
    if (fd2 < 0) {
        fprintf(stderr, "failed to create %s for writing\n", argv[2]);
        return -1;
    }

    while (1) {
        char c;

        ret = read(fd1, &c, sizeof(c));
        if (ret > 0) {
            write(fd2, &c, sizeof(c));
        } else if (ret <= 0) {
            close(fd1);
            close(fd2);
            break;
        }
    }

    return 0;
}
```

lseek system call

`lseek` system call allows to seek with in the file to a position specified as an argument.

The `lseek` system call prototype is defined as follows.

```
off_t lseek(int fd, off_t offset, int whence);
```

lseek returns an offset in the file pointed to by file descriptor fd. The offset and whence are related to each other.

the offset argument specifies the position in the file. the position is of the form off_t and whence is one of the following.

type	definition
SEEK_SET	set the offset to current given off_t bytes
SEEK_CUR	set the offset to current location + given off_t bytes
SEEK_END	set the offset to end of file + given off_t bytes

the return error codes are :

1. EBADF - bad fd
2. EINVAL - whence argument is invalid
3. EOVERFLOW - the file size offset cannot be represented in off_t type

Below example describe the use of lseek SEEK_SET attribute. Download here (https://github.com/DevNaga/gists/blob/master/lseek.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        return -1;
    }

    while (1) {
        char bytes[1000];

        ret = read(fd, bytes, sizeof(bytes));
        if (ret <= 0) {
            // reached eof .. set to beginning of the file
            lseek(fd, 0, SEEK_SET);

            fprintf(stderr, " reached EOF.. setting to the beginning of
file\n");

            sleep(1);
        } else {
            write(2, bytes, ret);
        }
    }

    close(fd);

    return 0;
}
```

in the above example, the read returns a -1 or 0 when it reaches an end of file, and there, the position of the file is moved to the beginning by setting the offset bytes to 0 and using SEEK_SET as it sets the position to given bytes. The `sleep` call is added to let the display slowly and repeatedly print the file contents using the below `write` system call. The `write` call is performed on the `stderr` file descriptor that is 2.

Below is another example of `lseek` that is used to find the file size. Download [here (https://github.com/DevNaga/gists/blob/master/lseek_size.c)](https://github.com/DevNaga/gists/blob/master/lseek_size.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int fd;
    off_t offset;
    struct stat s;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        return -1;
    }

    offset = lseek(fd, 0, SEEK_END);

    close(fd);

    printf("file size offset %ld\n", offset);

    ret = stat(argv[1], &s);
    if (ret < 0) {
        return -1;
    }

    printf("file size from stat %ld\n", s.st_size);

    return 0;
}
```

The above example, the open system call gives out an fd for a given valid file, then the `lseek` is called upon the `fd` with the offset 0 bytes and the `SEEK_END`. The `lseek` returns the offset currently in the file, basically this is starting at the beginning of the file till the last point, effectively counting the bytes in the file.

The same is verified by the `stat` system call with the file name as argument, the `stat` returns the `struct stat` which then contain the `st_size` member variable that outputs the file size.

running the above example prints,

```
./a.out lseek_size.c
file size offset 608
file size from stat 608
```

truncate and ftruncate

the `truncate` and `ftruncate` system calls are useful in truncating the file to a given size without adding anything into the file. These are very useful in coming sections where a memory map is used to write to files than the regular file io calls. See `mmap` section for more details on where `truncate` or `ftruncate` are being used.

The prototype of `truncate` is as follow.

```
int truncate(const char *path, off_t bytes);
```

the prototype of `ftruncate` is as follow.

```
int ftruncate(int fd, off_t bytes);
```

the `ftruncate` is different from `truncate` in the sense that it accepts only the `fd` than the filename.

Below example describe the use of `truncate` and `ftruncate`. Download [here (https://github.com/DevNaga/gists/blob/master/truncate.c)](https://github.com/DevNaga/gists/blob/master/truncate.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

long filesize(char *filename)
{
    struct stat st;
    int ret;

    ret = stat(filename, &st);
    if (ret < 0) {
        return -1;
    }

    return st.st_size;
}

int main(int argc, char **argv)
{
    long size;
    int fd;
    int ret;
```

```
    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    ret = truncate(argv[1], 1024 * 1024);
    if (ret < 0) {
        fprintf(stderr, "failed to truncate file to 1M error :%s \n",
strerror(errno));
        return -1;
    }

    size = filesize(argv[1]);

    fprintf(stderr, "truncated file %s file size %ld\n", argv[1], size);

    fprintf(stderr, "unlinking %s\n", argv[1]);

    ret = unlink(argv[1]);
    if (ret < 0) {
        fprintf(stderr, "failed to unlink %s\n", strerror(errno));
        return -1;
    }

    fd = open(argv[1], O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (fd < 0) {
        fprintf(stderr, "failed to open file %s\n", argv[1]);
        return -1;
    }

    ret = ftruncate(fd, 1024 * 1024);
    if (ret < 0) {
        fprintf(stderr, "failed to ftruncate file to 1M error: %s\n",
strerror(errno));
        return -1;
    }

    size = filesize(argv[1]);

    fprintf(stderr, "truncated file %s filesize %ld\n", argv[1], size);

    return 0;
}
```

Random number generator

The linux kernel provides a device interface to the psuedo randomnumber generator. The device is called **/dev/urandom**.

This device is opened like any other file in the linux. A read call on the device with the given length would give that length data back in random.

This random number or array is then used as a seed to a random number generator.

The sample program is located here (https://github.com/DevNaga/gists/blob/master/dev_random.c), and is also printed below for your reference.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int ret;
    int rand_fd;

    rand_fd = open("/dev/urandom", O_RDONLY);
    if (rand_fd < 0) {
        fprintf(stderr, "cannot open /dev/urandom for reading\n");
        return -1;
    }

    unsigned char randbytes[16];

    read(rand_fd, randbytes, sizeof(randbytes));

    int i;

    for (i = 0; i < sizeof(randbytes); i++) {
        printf("%02x", randbytes[i]);
    }

    printf("\n");

    return 0;
}
```

with the above example code, one can write a simple C++ class for various uses. Such as getting a byte, two bytes, an integer, a stream of random data etc. Below is an example of such class, Download here (https://github.com/DevNaga/gists/blob/master/random.cpp)

```
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

class RandomGen {
    private:
        int Fd_;
```

```
    public:
        RandomGen();
        int getRandBytes(char *bytes);
        int getRandBytes(short *bytes);
        int getRandBytes(int *bytes);
        int getRandBytes(uint8_t *bytes, int len);
        ~RandomGen();
};

RandomGen::RandomGen()
{
    Fd_ = open("/dev/urandom", O_RDONLY);
    if (Fd_ < 0) {
        return;
    }
}

RandomGen::~RandomGen()
{
    if (Fd_ > 0) {
        close(Fd_);
    }
}

int RandomGen::getRandBytes(char *bytes)
{
    return read(Fd_, bytes, sizeof(char));
}

int RandomGen::getRandBytes(short *bytes)
{
    return read(Fd_, bytes, sizeof(short));
}

int RandomGen::getRandBytes(int *bytes)
{
    return read(Fd_, bytes, sizeof(int));
}

int RandomGen::getRandBytes(uint8_t *bytes, int len)
{
    return read(Fd_, bytes, len);
}
```

In the above example, the RandomGen class has a constructor opening a file descriptor to the `/dev/urandom` in read only mode. The class then defines a overloaded function `getRandBytes` that can be called for one bytes, two, four or a series of bytes to receive for random data.

The class destructor is called upon, when the class goes out of scope , and it closes the file descriptor.

This class demonstrates an easy and alternate way of getting random numbers than using the highly unsecure `rand` or `srand` functions.

POSIX standard defines `_LARGEFILE64_SOURCE` to support reading or writing of files with sizes more than 2GB.

stat system call

The `stat` system call is very useful in knowing the information about the file. The information such as file last access time (read and write), created time, file mode, permissions, file type, file size etc are provided by the stat system call. These are all stored in the `struct stat` data structure. This we need to pass when calling the stat().

The `stat` prototype looks like below:

```
int stat(const char *path, struct stat *s);
```

`stat` system call accepts a file specified in `path` argument and outputs the attributes into the `struct stat` structure. The pointer `s` must be a valid pointer.

include the following header files when using this system call.

1. `<sys/stat.h>`
2. `<sys/types.h>`
3. `<unistd.h>`

`stat` returns 0 on success. So the structures are only accessed if it returns 0.

The stat data structure would look as the following.

```
struct stat {
    dev_t st_dev;          // id of the device
    ino_t st_ino;          // inode number
    mode_t st_mode;        // protection
    nlink_t st_nlink;      // number of hardlinks
    uid_t st_uid;          // user id of the owner
    gid_t st_gid;          // group id of the owner
    dev_t st_rdev;         // device id (if special file)
    off_t st_size;         // total size in bytes
    blksize_t st_blksize;  // blocksize for filesystem io
    blkcnt_t st_blocks;    // number of 512B blocks allocated
    struct timespec st_atim; // time of last access
    struct timespec st_mtim; // time of last modification
    struct timespec st_ctim; // time of last status change

    #define st_atime st_atim.tv_sec
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
};
```

stat data structure taken from the manual page of stat system call

The most common usage of `stat` is to know if the given file is a regular file or directory.

```
char *path = "/home/dev/work/test.c"
struct stat s;

if (stat(path, s) < 0) {
    fprintf(stderr, "failed to stat %s\n", s);
    perror("stat:");
    return -1;
}

if (s.st_mode & S_IFREG) {
    fprintf(stderr, "regular file\n");
} else {
    fprintf(stderr, "unknown or un-tested file type\n");
}
```

below example provide a detailed description of the file type checking with `stat` system call. Download here (https://github.com/DevNaga/gists/blob/master/stat_file.c)

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    struct stat s_;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    ret = stat(argv[1], &s_);
    if (ret < 0) {
        fprintf(stderr, "failed to stat %s\n", argv[1]);
        return -1;
    }

    if (S_ISREG(s_.st_mode)) {
        fprintf(stderr, "[%s] is regular file\n", argv[1]);
    } else if (S_ISDIR(s_.st_mode)) {
        fprintf(stderr, "[%s] is directory\n", argv[1]);
    } else if (S_ISCHR(s_.st_mode)) {
        fprintf(stderr, "[%s] is character device\n", argv[1]);
    } else if (S_ISBLK(s_.st_mode)) {
        fprintf(stderr, "[%s] is block device\n", argv[1]);
    } else if (S_ISFIFO(s_.st_mode)) {
        fprintf(stderr, "[%s] is a fifo\n", argv[1]);
    } else if (S_ISLNK(s_.st_mode)) {
        fprintf(stderr, "[%s] is a symlink\n", argv[1]);
    } else if (S_ISSOCK(s_.st_mode)) {
        fprintf(stderr, "[%s] is a socket\n", argv[1]);
    }

    return 0;
}
```

Running on the following files gives:

Regular files:

```
./a.out stat_file.c
[stat_file.c] is regular file
```

Directory:

```
./a.out .
[.] is directory
```

Character device:

```
./a.out /dev/null
[/dev/null] is character device
```

Example: basic stat example

Below is another example of `stat` system call getting the size of a file in bytes. Download here (https://github.com/DevNaga/gists/blob/master/stat_filesize.c)

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

int main(int argc, char **argv)
{
    int ret;
    struct stat s;

    if (argc != 2) {
        printf("%s [file name]\n", argv[0]);
        return -1;
    }

    ret = stat(argv[1], &s);
    if (ret) {
        printf("failed to stat: %s\n", strerror(errno));
        return -1;
    }

    printf("file %s size %ld\n", argv[1], s.st_size);
    return 0;
}
```

The `st_atime` is changed by file accesses, for ex: read calls.

The `st_mtime` is changed by file modifications, for ex: `write` calls.

The `st_ctime` is changed by writing or by setting inode, for ex: permissions, mode etc.

The below example provides the file's last access time, modification times etc. Download here (https://github.com/DevNaga/gists/blob/master/stat_access.c)

```c
#include <stdio.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
    struct stat s;
    int ret;

    if (argc != 2) {
        printf("%s [filename] \n", argv[0]);
        return -1;
    }

    ret = stat(argv[1], &s);
    if (ret) {
        printf("failed to stat %s\n", strerror(errno));
        return -1;
    }


    printf("last accessed: %ld, \n last modified: %ld, \n last status changed:
%ld\n",
                            s.st_atime,
                            s.st_mtime,
                            s.st_ctime);
    return 0;
}
```

A more detailed timestamp information can be obtained by using the `asctime` and `localtime` system calls on the `s.st_atime`, `s.st_mtime`, and `s.st_ctime` attributes. Below example shows the details, Download here (https://github.com/DevNaga/gists/blob/master/stat_timestamp.c)

```c
#include <stdio.h>
#include <errno.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>

int main(int argc, char **argv)
{
    struct stat s;
    int ret;

    if (argc != 2) {
        printf("%s [filename] \n", argv[0]);
        return -1;
    }

    ret = stat(argv[1], &s);
    if (ret) {
        printf("failed to stat %s\n", strerror(errno));
        return -1;
    }

    printf("last accessed : %s\n", asctime(localtime(&s.st_atime)));
    printf("last modified : %s\n", asctime(localtime(&s.st_mtime)));
    printf("last status change : %s\n", asctime(localtime(&s.st_ctime)));

    return 0;
}
```

The `stat` system call is often used in conjunction with the `readdir` system call, to find if the path contains a file or a directory.

More about the `st_mode` field and the way the files are created is described here..

the open system call as described before, has permission bits when calling in `O_CREAT`. The call for reference looks as below,

```c
int open(const char *filename, int flags, mode_t mode);
```

the `mode` bits are only valid if the file opened in `O_CREAT`.

The mode bits of type `mode_t`, can be one of the following types.

type	description
S_IRUSR	read only for the current user
S_IWUSR	write only for the current user
S_IXUSR	execute only for the current user
S_IROTH	read only others
S_IWOTH	write only others

S_IXOTH execute only others
S_IRGRP read only group
S_IWGRP write only group
S_IXGRP execute only group

below example describe the permission bits and their effects on the program. Download here (https://github.com/DevNaga/gists/blob/master/open_trait.c)

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    struct stat s;
    int fd;
    int ret;

    if (argc != 3) {
        fprintf(stderr, "<%s> filename mode\n", argv[0]);
        fprintf(stderr, "mode is one of user, user_group, user_others\n");
        return -1;
    }

    mode_t mask = 0;

    mask = umask(mask);

    printf("creation of old mask %o\n", mask);

    mode_t mode = 0;

    if (!strcmp(argv[2], "user")) {
        mode = S_IRUSR | S_IWUSR;
    } else if (!strcmp(argv[2], "user_group")) {
        mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;
    } else if (!strcmp(argv[2], "user_others")) {
        mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;
    }

    if (mode == 0) {
        return -1;
    }

    printf("mode bits %o\n", mode);
    fd = open(argv[1], O_CREAT | O_WRONLY | O_TRUNC, mode);
    if (fd < 0) {
        return -1;
    }
```

```
    ret = stat(argv[1], &s);
    if (ret < 0) {
        return -1;
    }

    printf("file %s opened, mode %o\n", argv[1], s.st_mode & 0xfff);

    close(fd);
    return 0;
}
```

Above example uses what is called umask system call. For most of the users the umask is bits 002 and on some systems the umask is set to 022. This umask is then negated with file creation bits as in case of open system call (the mode bits of type mode_t) to get resulting permission bits for the file.

The umask system call returns the previous permission bits for this process and sets the permissions given as mode_t argument.

In the above example, umask permission bits are cleared off with mode bits setting all permissions to 0. This means that if a file is created with flags 0666 that is S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH, the umask bits are 000 and the resulting operation that is 0666 ~ 000 produce the permission bits as 0666.

That is if the umask is not cleared off and the previous umask of the process is 002 then the permission bits of the resulting operation that is 0666 ~ 002 produce 0664. That is the file always be opened in S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH than the default asked for. the S_IWOTH is missed from the new file.

lstat system call

fstat system call

ioctl system call

manual page of linux ioctl_list lists down a various set of linux supported ioctls.

Directory manipulation

Directories are mapped as well into inodes. Linux supports the nesting of directories.

Reading / Writing Directories programmatically under linux:

Under the linux, directory is read by using the ls command. The ls command performs the listing of files and directories. The mkdir command performs the creation of directories. The touch command creates a file. The ls, mkdir and touch run in a shell (such as bash or ash or sh).

The ls command performs directory reading to list out the contents. The contents can be files or directories. There are many types of files with in the linux.

The opendir system call opens up a directory that is given as its first argument. It returns a directory tree node using which we can effectively traverse the directory using the readdir system call. The readdir call is repeatedly called over the directory tree node until the return value of the readdir becomes NULL.

The manual page of the opendir gives us the following prototype.

```
DIR * opendir(const char *name);
```

The call returns the directory tree node of type DIR as a pointer.

The manual page of the readdir gives us the following prototype.

```
struct dirent * readdir(DIR *dir);
```

The readdir takes the directory tree node pointer of type DIR that is returned from the opendir call. The readdir call is repeatedly called until it returns NULL. Each call to the readdir gives us a directory entry pointer of type struct dirent. From the man pages this structure is as follows:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* not an offset; see NOTES */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                           by all filesystem types */
    char           d_name[256]; /* filename */
};
```

The d_name and d_type elements in the structure are the most important things to us. The d_name variable gives us the file / directory that is present under the parent directory. The d_type tells us the type of the d_name. If the d_type is DT_DIR, then the d_name contains a directory, and if the d_type is DT_REG, then the d_name is a regular file.

An opendir call must follow a call to closedir if the opendir call is successful. If there is an opendir call, and no closedir is performed, then it results in a memory leak. From the man pages, the closedir call looks below:

```
int closedir(DIR *dirp);
```

The closedir will close the directory referenced by dirp pointer and frees up any memory that is allocated by the opendir call.

The below example is a basic ls command that perform the listing of the directory contents. It does not perform the listing of symbolic links, permission bits, other types of files.

You can download the example from [here](https://github.com/DevNaga/gists/blob/master/basic_listdir.c)

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <dirent.h>

int main(int argc, char **argv)
{
    DIR *dir;
    struct dirent *entry;
    char *dirname = NULL;

    if (argc == 1)
        dirname = ".";
    else
        dirname = argv[1];

    dir = opendir(dirname);
    if (!dir) {
        fprintf(stderr, "failed to open %s\n", dirname);
        return -1;
    }

    while (entry = readdir(dir)) {
        switch (entry->d_type) {
            case DT_DIR:
                fprintf(stderr, "dir         ");
            break;
            case DT_REG:
                fprintf(stderr, "reg         ");
            break;
            default:
                fprintf(stderr, "unknown     ");
            break;
        }

        fprintf(stderr, "%s\n", entry->d_name);
    }

    closedir(dir);

    return 0;
}
```

Example: Basic `ls` command example

The above example simply lists down the files and directories. It never lists down the properties of the files / directories, such as the permission bits, timestamps etc. By taking this as an example, we can solve the below programming problems.

1. Sort the contents of the directory and print them.
2. Recursively perform reading of the directories with in the parent directories until there exist no more directories. The directories "." and ".." can be ignored.

Some file systems does not set the `entry->d_type` variable. Thus it is advised to perform the `stat` system call on the `entry->d_name` variable. Usually the `entry->d_name` is only an absolute name and does not contain a full path, it is advised to append the full path before the `entry->d_name`.

The following example shows how a `stat` system call is used to find out the filetype.

```
struct stat s;
char buf[1000];

while (entry = readdir(dirp)) {
    memset(buf, 0, sizeof(buf));

    // append the directory before dir->d_name
    strcpy(buf, directory);
    strcat(buf, "/");
    strcpy(buf, dir->d_name);

    if (stat(buf, &s)) {
        printf("failed to stat %s\n", buf);
        continue;
    }

    if (S_ISREG(s.st_mode)) {
        printf("regular file %s\n", buf);
    } else if (S_ISDIR(s.st_mode)) {
        printf("directory %s\n", buf);
    } else if (S_ISLNK(s.st_mode)) {
        printf("link %s\n", buf);
    }
}
```

Chdir system call

The `chdir` system call changes the current working directory of the program. The `getcwd` system call gets the current directory the program is under.

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        int ret;

        if (argc != 2) {
                fprintf(stderr, "%s <directory name>\n", argv[0]);
                return -1;
        }

        ret = chdir(argv[1]);
        if (ret) {
                fprintf(stderr, "Failed changing the directory to %s, error:
%s\n",
                                                argv[1], strerror(errno));
                return -1;
        }

        printf("directory change successful\n");

        return 0;
}
```

Few of the examples:

Permission denied on a directory user requested:

```
devnaga@hanzo:~$ ./a.out /proc/1/fd
Failed changing the directory to /proc/1/fd, error: Permission denied
```

invalid directory being passed as input:

```
devnaga@hanzo:~$ ./a.out /proc/8a
Failed changing the directory to /proc/8a, error: No such file or directory
```

valid directory with valid permissions:

```
devnaga@hanzo:~$ ./a.out /proc
directory change successful
```

The `chdir` affects only the calling program .

Creating directories with `mkdir`

The `mkdir` also a system call that creates a directory. The command `mkdir` with option `-p` would recursively create the directories. However, the `mkdir` system call would only create one directory.

The below program demonstrates a series of calls that manipulate or get the information from the directories. You can also view / download it here (https://github.com/DevNaga/gists/blob/master/mkdir.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    char dirname[40];
    int ret;

    ret = mkdir(argv[1], 0755);
    if (ret < 0) {
        perror("failed to mkdir: ");
        return -1;
    }

    printf("successfully created %s\n", argv[1]);

    printf("going into %s\n", argv[1]);

    ret = chdir(argv[1]);
    if (ret < 0) {
        perror("failed to chdir: ");
        return -1;
    }

    printf("inside %s\n", getcwd(dirname, sizeof(dirname)));

    return 0;
}
```

However, when we compile and run the above program with good inputs as the following:

```
./mkdir_program test/
successfully created test/
going into test/
inside test/
```

and when the program exits, we are still in the directory where the program is compiled and run.

This is because the program does not affect the current directory of the shell (Shell is however a program too). The directory change is only affected to the program but not to anything else in the userspace if they are not related.

scandir

The `scandir` scans the directory and calls the `filter` and `compar` functions and returns a list of `struct dirent` datastructures and returns the length of them.

The prototype of the `scandir` looks as below..

```
int scandir(const char *dir, struct dirent ***list,
            int (*filter)(const struct dirent *),
            int (*compar)(const struct dirent **, const struct dirent **));
```

The compare function can be a sorting function that arranges the files in an order. The `Glibc` has `alphasort` API to call as `compar` function. On success it returns the number of directory entries selected. On failure, it returns -1. Below is one of the example..

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

int main(int argc, char **argv)
{
    struct dirent **list;
    int n, i;

    if (argc != 2) {
        printf("%s [directory] \n", argv[0]);
        return -1;
    }

    n = scandir(argv[1], &list, NULL, alphasort);
    if (n < 0) {
        printf("failed to scandir %s\n", argv[1]);
        return -1;
    }

    for (i = 0; i < n; i ++) {
        printf("name %s\n", list[i]->d_name);
        free(list[i]);
    }
    free(list);

    return 0;
}
```

rmdir

`rmdir` system call removes the directory. On failure it returns a corresponding error code. The following program demos an `rmdir` call and the example of the such.

```c
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>

int main(int argc, char **argv)
{
    int ret;

    if (argc != 2) {
        printf("%s [directory name]\n", argv[0]);
        return -1;
    }

    ret = rmdir(argv[1]);
    if (ret) {
        printf("failed to rmdir %s\n", strerror(errno));
        return -1;
    }

    printf("rmdir success\n");
    return 0;
}
```

Example: rmdir demo

Here shows the various runs of rmdir call:

```
root@4a032360f5c5:~/books# ./a.out test_file
failed to rmdir No such file or directory
root@4a032360f5c5:~/books# ./a.out test_directory
failed to rmdir No such file or directory
root@4a032360f5c5:~/books# ./a.out /proc/
failed to rmdir Device or resource busy
root@4a032360f5c5:~/books# ./a.out /proc/1/
failed to rmdir Operation not permitted
root@4a032360f5c5:~/books# ./a.out /proc/1/fd
failed to rmdir Permission denied
root@4a032360f5c5:~/books# mkdir pool
root@4a032360f5c5:~/books# ./a.out pool/
rmdir success
root@4a032360f5c5:~/books#
```

This completes the directory manipulation chapter in linux. Mostly next things in here is going to be some adventurous and interesting programs that provides the practical experience.

Programming problems

statvfs and fstatvfs

`statvfs` and `fstatvfs` system calls provide the information of a mounted file system.

`statvfs` prototype is as follows.

```
int statvfs(const char *path, struct statvfs *vfs);
```

`fstatvfs` prototype is as follows.

```
int fstatvfs(int fd, struct statvfs *vfs);
```

the `fstatvfs` is same as `statvfs` but operates on the file descriptor pointed to by the file.

The `statvfs` and `fstatvfs` system calls return the file system information of a given `path` into the `vfs` structure. The `vfs` must be a valid pointer. The path can be a file under a directory or a directory or a mount point itself.

Below is an example of `statvfs`. Download [here (https://github.com/DevNaga/gists/blob/master/statvfs.c)](https://github.com/DevNaga/gists/blob/master/statvfs.c)

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/statvfs.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
    struct statvfs vfs_;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    ret = statvfs(argv[1], &vfs_);
    if (ret != 0) {
        fprintf(stderr, "failed to statvfs [%s] error : %s\n",
                            argv[1], strerror(errno));
        return -1;
    }

    fprintf(stderr, "<statvfs> :\n");
    fprintf(stderr, "\t block size %lu\n", vfs_.f_bsize);
    fprintf(stderr, "\t fragment size %lu\n", vfs_.f_frsize);
    fprintf(stderr, "\t blocks %lu\n", vfs_.f_blocks);
    fprintf(stderr, "\t number of free blocks %lu\n", vfs_.f_bfree);
    fprintf(stderr, "\t number of free blocks for unprivileged user %lu\n",
vfs_.f_bavail);
    fprintf(stderr, "\t number of inodes %lu\n", vfs_.f_files);
    fprintf(stderr, "\t number of free inodes %lu\n", vfs_.f_ffree);
    fprintf(stderr, "\t number of free inodes for unprivileged user %lu\n",
vfs_.f_favail);
    fprintf(stderr, "\t file system id %lu\n", vfs_.f_fsid);
    fprintf(stderr, "\t mount flags %lx\n", vfs_.f_flag);
    fprintf(stderr, "\t filename max len %lu\n", vfs_.f_namemax);

    return 0;
}
```

```
./a.out /home
<statvfs> :
     block size 4096
     fragment size 4096
     blocks 231975294
     number of free blocks 130668897
     number of free blocks for unprivileged user 118879457
     number of inodes 58933248
     number of free inodes 53036742
     number of free inodes for unprivileged user 53036742
     file system id 7389426947721209490
     mount flags 1000
     filename max len 255
```

Below is an example of fstatvfs. Download [here](https://github.com/DevNaga/gists/blob/master/fstatvfs.c) (https://github.com/DevNaga/gists/blob/master/fstatvfs.c)

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/statvfs.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
    struct statvfs vfs_;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s\n", argv[1]);
        return -1;
    }

    ret = fstatvfs(fd, &vfs_);
    if (ret != 0) {
        fprintf(stderr, "failed to fstatvfs [%s] error : %s\n",
                            argv[1], strerror(errno));
        return -1;
    }

    fprintf(stderr, "<fstatvfs> :\n");
```

```
    fprintf(stderr, "\t block size %lu\n", vfs_.f_bsize);
    fprintf(stderr, "\t fragment size %lu\n", vfs_.f_frsize);
    fprintf(stderr, "\t blocks %lu\n", vfs_.f_blocks);
    fprintf(stderr, "\t number of free blocks %lu\n", vfs_.f_bfree);
    fprintf(stderr, "\t number of free blocks for unprivileged user %lu\n",
vfs_.f_bavail);
    fprintf(stderr, "\t number of inodes %lu\n", vfs_.f_files);
    fprintf(stderr, "\t number of free inodes %lu\n", vfs_.f_ffree);
    fprintf(stderr, "\t number of free inodes for unprivileged user %lu\n",
vfs_.f_favail);
    fprintf(stderr, "\t file system id %lu\n", vfs_.f_fsid);
    fprintf(stderr, "\t mount flags %lx\n", vfs_.f_flag);
    fprintf(stderr, "\t filename max len %lu\n", vfs_.f_namemax);

    close(fd);

    return 0;
}
```

the file descriptor however, can only be given a file but not a directory as given in `statvfs`. However, the information from `fstatvfs` is same as that of `statvfs`.

`statvfs` can be used to find out the disk usage. Below is an example, Download [here (https://github.com/DevNaga/gists/blob/master/statvfs_disk.c)](https://github.com/DevNaga/gists/blob/master/statvfs_disk.c)

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/statvfs.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
    struct statvfs vfs_;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    ret = statvfs(argv[1], &vfs_);
    if (ret != 0) {
        fprintf(stderr, "failed to statvfs [%s] error : %s\n",
                            argv[1], strerror(errno));
        return -1;
    }

    double total_disk = (vfs_.f_blocks * vfs_.f_frsize * 1.0) / (1024 * 1024 *
1024);
    double available = (vfs_.f_bfree * vfs_.f_frsize * 1.0) / (1024 * 1024 *
1024);

    fprintf(stderr, "total %f GB available %f GB used %f GB\n", total_disk,
available, total_disk - available);
    fprintf(stderr, "used disk size [%s] %f\n", argv[1], ((total_disk -
available) / total_disk) * 100);
    fprintf(stderr, "free disk size [%s] %f\n", argv[1], ((available) /
total_disk) * 100);
    return 0;
}
```

The `total_disk` is the number of blocks and the fragment size of the block and divide it with GB unit. The `available` or the `free_disk` is the free blocks multiplied by the size of the fragments and divide with GB unit to get the units in terms of GigaBytes.

The used is the `total - free` disk space. The usage `%` is calculate by dividing the result with `total` and multiply by 100.

Fourth chapter

Sockets and Socket programming

1. Socket API

In linux, sockets are the pipes or the software wires that are used for the exchange of the data between two or more machines, or even between the processes with in the same machine, using the TCP/IP protocol stack.

The server program opens a sockets, waits for someone to connect to it. The socket can be created to communicate over the TCP or the UDP protocol and the underlying networking layer can be IPv4 or IPv6. Often sockets are used to provide interprocess communication between the programs with in the OS.

The **socket** API is the most commonly used API in a network oriented programs. This is the starting point to create a socket that can be used for further communication either with in the OS in a computer or between two computers.

In the Linux systems programming, the TCP protocol is denoted by a macro called **SOCK_STREAM** and the UDP protocol is denoted by a macro called **SOCK_DGRAM**. Either of the above macros are passed as the second argument to the **socket** API.

Below are the most commonly used header files in the socket programming.

1. `<sys/socket.h>`
2. `<arpa/inet.h>`
3. `<netinet/in.h>`

The protocol IPv4 is denoted by **AF_INET** and the protocol IPv6 is denoted by **AF_INET6**. Either of these macros are passed as the first argument to the **socket** API.

The socket API usually takes the following form.

```
socket (Address family, transport protocol, IP protocol);
```

for a TCP socket:

```
socket(AF_INET, SOCK_STREAM, 0);
```

for a UDP socket:

```
socket(AF_INET, SOCK_DGRAM, 0);
```

type	**socket type**
TCP	SOCK_STREAM
UDP	SOCK_DGRAM
RAW	SOCK_RAW

The return value of the **socket** API is the actual socket connection. The below code snippet will give an example of the usage:

```
int sock;

// create a TCP socket
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    perror("failed to open a socket");
    return -1;
}

printf("the socket address is %d\n", sock);
```

The kernel allocates a socket structure and returns an entry corresponding this structure. Each socket structure then linked to the process. So a process owns the socket fd and is private to it.

The returned socket address is then used as the communication channel.

Each socket returned is one more than the max file descriptor that is opened currently by the program. The program usually have 3 file descriptors when opened, 0 - stdin, 1 - stdout and 2 - stderr.

To create a server, we must use a **bind** system call to tell others that we are running at some port and ip. Like naming the connection and allowing others to talk with us by referring to the name.

```
bind(Socket Address, Server Address Structure, length of the Server address
structure);

ret = bind(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
```

sometimes the `bind` systemcall might fail, because when a socket is closed, its references are not really removed from the kernel, until unless some of the tasks that the socket been waiting for, have been complete. Thus, usually it is advise to use some of the socket based options to control this behavior. The system call `setsockopt` allows to control socket options and `getsockopt` gets the socket options.

It is advised to perform the `setsockopt` call with `SO_REUSEADDR` option after a call to the `socket`. This is described nicely here (http://www.unixguide.net/network/socketfaq/4.5.shtml).

In brief, if you stopped the server for some time and started back again quickly, the `bind` may fail. This is because the OS still contain the context associated to your server (ip and port) and does not allow others to connect with the same information. The context gets cleared with the `setsockopt` call with the `SO_REUSEADDR` option before the bind.

The setsockopt option would look like the below.

```
int setsockopt(int sock_fd, int level, int optname, const void *optval,
socklen_t optlen);
```

The basic and most common usage of the `setsockopt` is like the below:

```
int reuse_addr = 1;   // turn on the reuse address operation in the OS

ret = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse_addr,
sizeof(reuse_addr));
```

More on the setsockopt and getsockopt is described later in this chapter.

A server must register to the OS that it is ready to perform accepting the connections by using the listen system call. It takes the below form:

```
int listen(int sock_fd, int backlog);
```

The sock_fd is the socket address returned by the socket system call. The backlog defines the number of pending connections on to the socket. If the backlog connections cross the limit of backlog, the client will get a connection refused error. The usual call to the listen for an in-system and embedded server would be as follows:

```
ret = listen(sock, 10);     // server will only perform replies to a max of 10 clients
```

The accept system call is followed after the call to the listen system call.

The accept system call takes the below form:

```
int accept(int sock_fd, struct sockaddr *addr, socklen_t *addrlen);
```

The sock_fd is the socket address returned by the socket system call. the addr argument is filled by the OS and gives us the address of the neighbor. the addrlen is also filled by the OS and gives us the type of the data structure that the second argument contain. Such as if the length is of size struct sockaddr_in then the address is of the IPv4 type and if its of size struct sockaddr_in6 then the address is of the IPv6 type.

The accept function most commonly can be written as:

```
struct sockaddr_in remote;
socklen_t remote_len;

ret = accept(sock, (struct sockaddr *)&remote_addr, &remote_len);
if (ret < 0) {
    return -1;
}
```

The accept system call returns a file descriptor of the client that is communicating with. Any communication with the client must be performed over the file descriptor returned from the accept call. It is advised to store the client file descriptor till the client closes the socket or shutsdown connection.

In case of a client, we do not have to call the bind, listen and accept system calls but call the connect system call.

The connect system call takes the following form:

```
int connect(int sock_fd, const struct sockaddr *addr, socklen_t addrlen);
```

The connect system call allows the client to connect to a peer defined in the addr argument and the peer's length in addrlen argument.

The connect system call most commonly takes the following form:

```
char server_addr[] = "127.0.0.1"
int server_port = 45454;

struct sockaddr_in server = {
    .family                 = AF_INET,
    .sin_addr.s_addr        = inet_addr(server_addr),
    .sin_port               = htons(server_port),
};

ret = connect(sock_fd, (struct sockaddr *)&server, sizeof(server));
if (ret < 0) {
    return -1;
}
```

The address `127.0.0.1` is called the loopback address. Most server programs that run locally with in the computer for IPC use this address for communication with the clients.

Always the port assignment must happen in network endian. This conversion is carried out by `htons`.

inet_addr

```
in_addr_t inet_addr(const char *cp);
```

inet_aton

```
int inet_aton(const char *cp, struct in_addr *inp);
```

inet_ntoa

```
char *inet_ntoa(struct in_addr in);
```

inet_ntop

```
const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
```

inet_pton

```
int inet_pton(int af, const char *src, void *dst);
```

The below sample programs describe about the basic server and client programs. The server programs creates the TCP IPv4 socket, sets up the socket option SO_REUSEADDR, binds, adds a backlog of 10 connections and waits on the accept system call. The server ip and port are given as the command line arguments.

The accept returns a socket that is connected to this server. The below program simply prints the socket address onto the screen and stops the program.

Download here (https://github.com/DevNaga/gists/blob/master/server.c)

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

```c
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SAMPLE_SERVER_CONN 10

int main(int argc, char **argv)
{
    int ret;
    int sock;
    int conn;
    int set_reuse = 1;
    struct sockaddr_in remote;
    socklen_t remote_len;

    if (argc != 3) {
        fprintf(stderr, "%s [ip] [port]\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("failed to socket\n");
        return -1;
    }

    ret = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &set_reuse,
sizeof(set_reuse));
    if (ret < 0) {
        perror("failed to setsockopt\n");
        close(sock);
        return -1;
    }

    remote.sin_family = AF_INET;
    remote.sin_addr.s_addr = inet_addr(argv[1]);
    remote.sin_port = htons(atoi(argv[2]));

    ret = bind(sock, (struct sockaddr *)&remote, sizeof(remote));
    if (ret < 0) {
        perror("failed to bind\n");
        close(sock);
        return -1;
    }

    ret = listen(sock, SAMPLE_SERVER_CONN);
```

```
    if (ret < 0) {
        perror("failed to listen\n");
        close(sock);
        return -1;
    }

    remote_len = sizeof(struct sockaddr_in);

    conn = accept(sock, (struct sockaddr *)&remote, &remote_len);
    if (conn < 0) {
        perror("failed to accept\n");
        close(sock);
        return -1;
    }

    printf("new client %d\n", conn);

    close(conn);

    return 0;
}
```

Example: Sample server program

The client program is described below. It creates a TCP IPv4 socket, connect to the server, on a successful connection, it prints the connection result and stops the program.

Download here (https://github.com/DevNaga/gists/blob/master/client.c)

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SAMPLE_SERVER_CONN 10

int main(int argc, char **argv)
{
    int ret;
    int sock;
    struct sockaddr_in remote;
    socklen_t remote_len;

    if (argc != 3) {
        fprintf(stderr, "%s [ip] [port]\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("failed to socket\n");
        return -1;
    }

    remote.sin_family = AF_INET;
    remote.sin_addr.s_addr = inet_addr(argv[1]);
    remote.sin_port = htons(atoi(argv[2]));

    remote_len = sizeof(struct sockaddr_in);

    ret = connect(sock, (struct sockaddr *)&remote, remote_len);
    if (ret < 0) {
        perror("failed to accept\n");
        close(sock);
        return -1;
    }

    printf("connect success %d\n", ret);

    close(sock);

    return 0;
}
```

Example: Sample client program

2. Sending and Receiving over the Sockets

We have seen the server and client connect to each other over sockets. Now that connection is established, the rest of the steps are the data-transfer. The data-transfers are performed using the simple system calls, `recv`, `send`, `recvfrom` and `sendto`.

The `recv` system call receives the data over the TCP socket, i.e. the socket is created with **SOCK_STREAM** option. The `recvfrom` system call receives the data over the UDP socket, i.e. the socket is created with **SOCK_DGRAM** option.

The `send` system call sends the data over the TCP socket and the `sendto` system call sends the data over the UDP socket.

The `recv` system call takes the following form:

```
ssize_t recv(int sock_fd, void *buf, size_t len, int flags);
```

The `recv` function receives data from the sock_fd into the buf of length len. The options of `recv` are specified in the flags argument. Usually the flags are specified as 0. However, for a non blocking mode of socket operation **MSG_DONTWAIT** option is used.

The example `recv`:

```
recv_len = recv(sock, buf, sizeof(buf), 0);
if (recv_len <= 0) {
    return -1;
}
```

The `recv_len` will return the length of the bytes received. `recv_len` is 0 or less than 0, meaning that the other end of the socket has closed the connection and we must close the connection. Otherwise, the `recv` function call will be restarted by the OS repeatedly causing heavy CPU loads. The code snipped shows the handling.

```
int ret;

ret = recv(sock, buf, sizeof(buf), 0);
if (ret <= 0) {
    close(sock);
    return -1;
}
```

The above code snippet checks for `recv` function return code for 0 and less than 0 and handles socket close.

The `recvfrom` system call is much similar to the `recv` and takes the following form:

```
ssize_t recvfrom(int sock_fd, void *buf, size_t len, int flags, struct sockaddr
*addr, socklen_t *addrlen);
```

The `recvfrom` is basically `recv` + `accept` for the UDP. The address of the sender and the length are notified in the addr and addrlen arguments. The rest of the arguments are same.

The example `recvfrom`:

```
struct sockaddr_in remote;
socklen_t remote_len = sizeof(remote);

recv_len = recvfrom(sock, buf, sizeof(buf), 0, (struct sockaddr *)&remote,
&remote_len);
if (recv_len < 0) {
    return -1;
}
```

The recv_len will return the length of bytes received. The address of the sender is filled in to the remote and the length in the remote_len.

The send system call takes the following form:

```
ssize_t send(int sock_fd, const void *buf, size_t len, int flags);
```

The send will return the length of bytes sent over the connection. The buffer buf of length len is sent over the connection. The flags are similar to that of recv and most commonly used flag is the MSG_DONTWAIT.

The example send:

```
sent_bytes = send(sock, buf, buf_len, 0);
if (sent_bytes < 0) {
    return -1;
}
```

The sent_bytes less than 0 means an error.

The sendto system call takes the following form:

```
ssize_t sendto(int sock_fd, const void *buf, size_t len, int flags, const
struct sockaddr *dest_addr, socklen_t dest_len);
```

The sendto is same as send with address.

The client program now performs a send system call to send "Hello" message to the server. The server program then receives over a recv system call to receive the message and prints it on the console.

With the recv and send system calls the above programs are modified to look as below.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SAMPLE_SERVER_CONN 10
```

```c
int main(int argc, char **argv)
{
    int ret;
    int sock;
    int conn;
    int set_reuse = 1;
    struct sockaddr_in remote;
    socklen_t remote_len;
    char buf[1000];

    if (argc != 3) {
        fprintf(stderr, "%s [ip] [port]\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("failed to socket\n");
        return -1;
    }

    ret = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &set_reuse,
sizeof(set_reuse));
    if (ret < 0) {
        perror("failed to setsockopt\n");
        close(sock);
        return -1;
    }

    remote.sin_family = AF_INET;
    remote.sin_addr.s_addr = inet_addr(argv[1]);
    remote.sin_port = htons(atoi(argv[2]));

    ret = bind(sock, (struct sockaddr *)&remote, sizeof(remote));
    if (ret < 0) {
        perror("failed to bind\n");
        close(sock);
        return -1;
    }

    ret = listen(sock, SAMPLE_SERVER_CONN);
    if (ret < 0) {
        perror("failed to listen\n");
        close(sock);
        return -1;
    }

    remote_len = sizeof(struct sockaddr_in);

    conn = accept(sock, (struct sockaddr *)&remote, &remote_len);
    if (conn < 0) {
        perror("failed to accept\n");
```

```
        close(sock);
        return -1;
    }

    printf("new client %d\n", conn);

    memset(buf, 0, sizeof(buf));

    printf("waiting for the data ... \n");
    ret = recv(conn, buf, sizeof(buf), 0);
    if (ret <= 0) {
        printf("failed to recv\n");
        close(conn);
        return -1;
    }

    printf("received %s \n", buf);

    close(conn);

    return 0;
}
```

Example: Tcp server with recv calls

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SAMPLE_SERVER_CONN 10

int main(int argc, char **argv)
{
    int ret;
    int sock;
    struct sockaddr_in remote;
    socklen_t remote_len;
    char buf[1000];

    if (argc != 3) {
        fprintf(stderr, "%s [ip] [port]\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if (sock < 0) {
        perror("failed to socket\n");
        return -1;
    }

    remote.sin_family = AF_INET;
    remote.sin_addr.s_addr = inet_addr(argv[1]);
    remote.sin_port = htons(atoi(argv[2]));

    remote_len = sizeof(struct sockaddr_in);

    ret = connect(sock, (struct sockaddr *)&remote, remote_len);
    if (ret < 0) {
        perror("failed to accept\n");
        close(sock);
        return -1;
    }

    printf("connect success %d\n", ret);

    printf("enter something to send\n");

    fgets(buf, sizeof(buf), stdin);

    ret = send(sock, buf, strlen(buf), 0);
    if (ret < 0) {
        printf("failed to send %s\n", buf);
        close(sock);
        return -1;
    }

    printf("sent %d bytes\n", ret);

    close(sock);

    return 0;
}
```

Example: Tcp client with send calls

Unix domain sockets

The unix domain sockets used to communicate between processes on the same machine locally. The protocol used is `AF_UNIX`. The unix domain sockets can have `SOCK_STREAM` or `SOCK_DGRAM` protocol families.

The example socket call can be the below..

```
int tcp_sock = socket(AF_UNIX, SOCK_STREAM, 0);  // -> for TCP

int udp_sock = socket(AF_UNIX, SOCK_DGRAM, 0);   // -> for UDP
```

The `struct sockaddr_un` data structure is used for the unix domain sockets. It is defined as follows.

```
struct sockaddr_un {
    sa_family_t sun_family;
    char        sun_path[108];
};
```

The data structure is defined in `sys/un.h`.

The code snippet for the bind call can be as below..

```
int ret;

char *path = "/tmp/test_server.sock"

struct sockaddr_un addr;

addr.sun_family = AF_UNIX;
unlink(path);
strcpy(addr.sun_path, path);

ret = bind(sock, (struct sockaddr *)&addr, sizeof(struct sockaddr_un));
if (ret < 0) {
    // handling the failure here
    printf("failed to bind\n");
    return -1;
}
```

The `unlink` call is used before the `bind` to make sure the path is not being used by any other service. This is to make sure the `bind` call would succeed. Otherwise bind fails.

The sample UNIX domain TCP server and client are shown below...

These are the steps at the server:

1. open a socket with `AF_UNIX` and `SOCK_STREAM`.
2. unlink the `SERVER_PATH` before performing the `bind`.
3. call `listen` to setup the socket into a listening socket.
4. accept single connection on the socket.
5. loop around in the `read` call for the data. When the `read` call returns 0, this means that the client has closed the connection. Alternatively the `select` system call can be used.
6. stop the server and quit the program.

These are the steps at the client:

1. open a socket with `AF_UNIX` and `SOCK_STREAM`.
2. connect to the server at `SERVER_PATH`.
3. after a successful `connect` call perform a write on the socket.
4. quit the program (thus making kernel's Garbage Collector cleanup the connection).

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

```c
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>

#define SERVER_PATH "/tmp/unix_sock.sock"

void server()
{
    int ret;
    int sock;
    int client;

    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        printf("failed to create socket\n");
        return;
    }

    struct sockaddr_un serv;
    struct sockaddr_un cli;

    unlink(SERVER_PATH);
    strcpy(serv.sun_path, SERVER_PATH);
    serv.sun_family = AF_UNIX;

    ret = bind(sock, (struct sockaddr *)&serv, sizeof(serv));
    if (ret < 0) {
        close(sock);
        printf("failed to bind\n");
        return;
    }

    ret = listen(sock, 100);
    if (ret < 0) {
        close(sock);
        printf("failed to listen\n");
        return;
    }

    socklen_t len = sizeof(serv);

    client = accept(sock, (struct sockaddr *)&cli, &len);
    if (client < 0) {
        close(sock);
        printf("failed to accept\n");
        return;
    }

    char buf[200];
```

```
    while (1) {
        memset(buf, 0, sizeof(buf));
        ret = read(client, buf, sizeof(buf));
        if (ret <= 0) {
            close(client);
            close(sock);
            printf("closing connection..\n");
            return;
        }
        printf("data %s\n", buf);
    }

    return;
}

void client()
{
    int ret;
    char buf[] = "UNIX domain client";
    int sock;

    sock = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock < 0) {
        printf("Failed to open socket \n");
        return;
    }

    struct sockaddr_un serv;

    strcpy(serv.sun_path, SERVER_PATH);
    serv.sun_family = AF_UNIX;

    ret = connect(sock, (struct sockaddr *)&serv, sizeof(serv));
    if (ret < 0) {
        close(sock);
        printf("Failed to connect to the server\n");
        return;
    }

    write(sock, buf, sizeof(buf));
    return;
}


int main(int argc, char **argv)
{
    int ret;

    if (argc != 2) {
        printf("%s [server | client]\n", argv[0]);
        return -1;
```

```
    }

    if (!strcmp(argv[1], "server")) {
        server();
    } else if (!strcmp(argv[1], "client")) {
        client();
    } else {
        printf("invalid argument %s\n", argv[1]);
    }

    return 0;
}
```

However, for a UNIX domain UDP sockets, we have to perform the `bind` call on both the sides... i.e. at the server and at the client. This is because when the server performs a `sendto` back to the client, it needs to know exactly the path of the client ... i.e. a name. Thus needing a `bind` call to let the server know about the client path.

So in our code example above, for a UNIX UDP socket, we need to change the `SOCK_STREAM` to `SOCK_DGRAM`, perform `bind` on the server as well as client and replace `read` and `write` calls with `sendto` and `recvfrom`.

the socket call to a UNIX UDP socket is as follows,

```
socket(AF_UNIX, SOCK_DGRAM, 0);
```

at server end, the server must call bind to inform the kernel about the name. A bind call is follows.

```
struct sockaddr_un serv;

// remove any existing path
unlink(SERV_NAME);

strcpy(serv.sun_path, SERV_NAME);
serv.sun_family = AF_UNIX;

bind(sock, (struct sockaddr *)&serv, sizeof(serv));
```

at client end, the client must also call bind to its own address and NOT the server address. A bind calla t the client is as follows.

```
struct sockaddr_un client_addr;

// remove any existing path
unlink(CLIENT_PATH);

strcpy(client_addr.sun_path, CLIENT_PATH);
client_addr.sun_family = AF_UNIX;

bind(sock, (struct sockaddr *)&client_addr, sizeof(client_addr));
```

The client must as well setup the sockaddr_un structure before it can do sendto to a server. An example of sendto is as follows.

```
struct sockaddr_un server_addr;

strcpy(server_addr.sun_path, SERV_NAME);
server_addr.sun_family = AF_UNIX;

sendto(sock, msg, msg_len, 0, (struct sockaddr *)&server_addr,
sizeof(server_addr));
```

Below is one of the example of the unix domain server and client in UDP. Download [here (https://github.com/DevNaga/gists/blob/master/af_unix_udp.cpp)](https://github.com/DevNaga/gists/blob/master/af_unix_udp.cpp)

```
#include <iostream>
#include <string>
#include <chrono>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/un.h>
#include <unistd.h>

#define SERV_NAME "/tmp/unix_serv.sock"
#define CLI_NAME "/tmp/unix_client.sock"

class server {
    public:
        server(std::string name)
        {
            int ret;

            fd_ = socket(AF_UNIX, SOCK_DGRAM, 0);
            if (fd_ < 0) {
                return;
            }
```

```
            struct sockaddr_un serv;

            unlink(name.c_str());
            strcpy(serv.sun_path, name.c_str());
            serv.sun_family = AF_UNIX;

            ret = bind(fd_, (struct sockaddr *)&serv, sizeof(serv));
            if (ret < 0) {
                return;
            }
        }
        int recvFrom(char *str, int str_size)
        {
            return recvfrom(fd_, str, str_size, 0, NULL, NULL);
        }
    private:
        int fd_;

};

class client {
    public:

        client(std::string name, std::string servName)
        {
            int ret;

            fd_ = socket(AF_UNIX, SOCK_DGRAM, 0);
            if (fd_ < 0) {
                return;
            }

            struct sockaddr_un clientAddr;

            unlink(name.c_str());
            strcpy(clientAddr.sun_path, name.c_str());
            clientAddr.sun_family = AF_UNIX;

            ret = bind(fd_, (struct sockaddr *)&clientAddr,
sizeof(clientAddr));
            if (ret < 0) {
                return;
            }

            strcpy(serv_.sun_path, servName.c_str());
            serv_.sun_family = AF_UNIX;
        }
        int sendTo(const char *str, int str_size)
        {
            return sendto(fd_, str, str_size, 0, (struct sockaddr *)&serv_,
sizeof(serv_));
```

```
        }

    private:
        struct sockaddr_un serv_;
        int fd_;
};

int main(int argc, char **argv)
{
    int ret;

    if (argc != 2) {
        std::cerr << argv[0] << " server / client " << std::endl;
        return -1;
    }

    if (std::string(argv[1]) == "server") {
        server s(SERV_NAME);

        while (1) {
            char string[100];
            ret = s.recvFrom(string, sizeof(string));
            if (ret < 0) {
                break;
            }
            std::cerr << "server: " << string << std::endl;
        }
    } else if (std::string(argv[1]) == "client") {
        client c(CLI_NAME, SERV_NAME);

        while (1) {
            std::string msg = "client says hi";
            sleep(1);

            std::cerr << "sending Hi .." << std::endl;
            c.sendTo(msg.c_str(), msg.length());
        }
    }
}
```

socketpair

`socketpair` creates an unnamed pair of sockets. Only the `AF_UNIX` is supported. Other than creating the two sockets, it is much similar to the two calls with `AF_UNIX`. The prototype is as follows..

```
int socketpair(int domain, int type, int protocol, int sv[2]);
```

include <sys/types.h> and <sys/socket.h> for the API.

```c
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(int argc, char **argv)
{
    int sv[2];
    int ret;

    ret = socketpair(AF_UNIX, SOCK_STREAM, 0, sv);
    if (ret < 0) {
        printf("Failed to create socketpair\n");
        return -1;
    }

    printf("socketpair created %d %d\n", sv[0], sv[1]);

    close(sv[0]);
    close(sv[1]);

    return 0;
}
```

Example: socketpair

3. Getsockopt and Setsockopt

The `getsockopt` and `setsockopt` are the two most commonly used socket control and configuration APIs.

The prototypes of the functions look as below.

```c
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

There are lots of possible socket options with different socket levels.

socket level	option name
SO_ACCEPTCONN	Check if a socket is marked to accept connections. returns 1 if the socket is capable of accepting the connections. returns 0 if the socket is not.
SO_BINDTODEVICE	Bind to a particular network device as specified as the option. If on success, the packets only from the device will be received and processde by the socket.
SO_RCVBUF	Sets or gets the socket receive buffer in bytes. The kernel doubles the value when set using the `setsockopt`.
SO_REUSEADDR	Reuse the local address that is used previously by the other program which has been stopped. Only used in the server programs before the `bind` call.
SO_SNDBUF	Sets or gets the maximum socket send buffer in bytes. The kernel doubles this values when set by using the `setsockopt`.

An Example of the `SO_ACCEPTCONN` looks as below.

The below example shows that there is no `listen` call so that the socket will not be able to perform the accept of connections.

Thus the accept connection is set to "No" in the kernel on this socket.

```c
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/socket.h>

int main(int argc, char **argv)
{
        int val;
        int optlen;
        int ret;
        int sock;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                fprintf(stderr, "failed to open socket\n");
                return -1;
        }

        struct sockaddr_in serv = {
                .sin_family = AF_INET,
                .sin_addr.s_addr = inet_addr("127.0.0.1"),
        };

        ret = bind(sock, (struct sockaddr *)&serv, sizeof(serv));
        if (ret < 0) {
                fprintf(stderr, "failed to bind\n");
                close(sock);
                return -1;
        }

        optlen = sizeof(val);

        ret = getsockopt(sock, SOL_SOCKET, SO_ACCEPTCONN, &val, &optlen);
        if (ret < 0) {
                fprintf(stderr, "failed to getsockopt\n");
                close(sock);
                return -1;
        }

        printf("accepts connection %s\n", val ? "Yes": "No");

        close(sock);

        return 0;
}
```

The below example shows that there is `listen` call so that the socket will be able to perform the accept of connections.

Thus the accept connection is set to "Yes" in the kernel on this socket.

```c
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/socket.h>

int main(int argc, char **argv)
{
        int val;
        int optlen;
        int ret;
        int sock;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                fprintf(stderr, "failed to open socket\n");
                return -1;
        }

        struct sockaddr_in serv = {
                .sin_family = AF_INET,
                .sin_addr.s_addr = inet_addr("127.0.0.1"),
        };

        ret = bind(sock, (struct sockaddr *)&serv, sizeof(serv));
        if (ret < 0) {
                fprintf(stderr, "failed to bind\n");
                close(sock);
                return -1;
        }

        ret = listen(sock,  100);
        if (ret < 0) {
                fprintf(stderr, "failed to listen\n");
                close(sock);
                return -1;
        }

        optlen = sizeof(val);

        ret = getsockopt(sock, SOL_SOCKET, SO_ACCEPTCONN, &val, &optlen);
        if (ret < 0) {
                fprintf(stderr, "failed to getsockopt\n");
                close(sock);
                return -1;
        }

        printf("accepts connection %s\n", val ? "Yes": "No");
```

```
        close(sock);

        return 0;
}
```

select and epoll

1. select system call

`select` system call is used to wait on multiple file descriptors at the same time. The file descriptors can be a file, socket, pipe, message queue etc.

`FD_SET` is used to set the corresponding file descriptor in the given `fd_set`.

`FD_ISSET` is used to test if the corresponding file descriptor is set in the given `fd_set`.

`FD_ZERO` resets a given `fd_set`.

`FD_CLR` clears an fd in the `fd_set`.

`select` can also be used to perform millisecond timeout. This can also be used to selective wait for an event or a delay.

the `select` from the manual page looks like below:

`int select(int nfd, FD_SET *rdfd, FD_SET *wrfd, FD_SET *exfd, struct timeval *timeout);`

The select returns greater than 0 and sets the file descriptors that are ready in the 3 file descriptor sets. and returns `0` if there is a timeout.

A TCP server using the select loop is demonstrated below:

The select accepts 3 sets of file descriptor sets. Read fdset, Write fdset, Except fdset. Of all we only use read fdset. We do not really need write fdset and except fdset in most of the cases.

The select when returned (and with no error), the above fdsets should be tested with the FD_ISSET with the list of FDs that we are interested in.

The below two programs can be downloaded here: Server (https://github.com/DevNaga/gists/blob/master/tcp_select_server.c) and Client (https://github.com/DevNaga/gists/blob/master/tcp_client.c)

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```c
#define TCP_SERVER_PORT 21111

int main(int argc, char **argv)
{
    int ret;
    fd_set rdset;
    struct sockaddr_in serv_addr = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = inet_addr("127.0.0.1"),
        .sin_port = htons(TCP_SERVER_PORT),
    };

    int max_fd = 0;
    int set_reuse = 1;
    int serv_sock;

    int client_list[100];
    int i;

    for (i = 0; i < sizeof(client_list) / sizeof(client_list[0]); i ++) {
        client_list[i] = -1;
    }

    serv_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (serv_sock < 0) {
        return -1;
    }

    ret = setsockopt(serv_sock, SOL_SOCKET, SO_REUSEADDR, &set_reuse,
sizeof(set_reuse));
    if (ret < 0) {
        return -1;
    }

    ret = bind(serv_sock, (struct sockaddr *)&serv_addr, sizeof(struct
sockaddr_in));
    if (ret < 0) {
        return -1;
    }

    ret = listen(serv_sock, 100);
    if (ret < 0) {
        return -1;
    }

    if (max_fd < serv_sock)
        max_fd = serv_sock;

    FD_ZERO(&rdset);

    FD_SET(serv_sock, &rdset);
```

```
    while (1) {
        int clifd;
        fd_set allset = rdset;

        ret = select(max_fd + 1, &allset, 0, 0, NULL);
        if (ret > 0) {
            if (FD_ISSET(serv_sock, &allset)) {
                clifd = accept(serv_sock, NULL, NULL);
                if (clifd < 0) {
                    continue;
                }

                for (i = 0; i < sizeof(client_list) / sizeof(client_list[0]); i
++) {
                    if (client_list[i] == -1) {
                        client_list[i] = clifd;
                        FD_SET(clifd, &rdset);
                        if (max_fd < clifd)
                            max_fd = clifd;
                        printf("new fd %d\n", clifd);
                        break;
                    }
                }
            } else {
                for (i = 0; i < sizeof(client_list) / sizeof(client_list[0]); i
++) {
                    if ((client_list[i] != -1) &&
                        (FD_ISSET(client_list[i], &allset))) {
                        char buf[100];

                        printf("client %d\n", client_list[i]);
                        ret = recv(client_list[i], buf, sizeof(buf), 0);
                        if (ret <= 0) {
                            printf("closing %d\n", client_list[i]);
                            FD_CLR(client_list[i], &rdset);
                            close(client_list[i]);
                            client_list[i] = -1;
                            continue;
                        }

                        printf("read %s\n", buf);
                    }
                }
            }
        }
    }

    return 0;
}
```

The tcp sample client is defined below:

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define TCP_SERVER_PORT 21111

int main()
{
    int cli_sock;
    int ret;

    struct sockaddr_in serv_addr = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = inet_addr("127.0.0.1"),
        .sin_port = htons(TCP_SERVER_PORT),
    };

    cli_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (cli_sock < 0) {
        return -1;
    }

    ret = connect(cli_sock, (struct sockaddr *)&serv_addr, sizeof(struct
sockaddr_in));
    if (ret < 0) {
        return -1;
    }

    char msg[] = "sending data to the server";

    send(cli_sock, msg, strlen(msg) + 1, 0);

    close(cli_sock);

    return 0;
}
```

The above two programs are only demonstratable programs. They have many errors. Finding and spotting the errors is going to be your task here. And the solution to it makes us to become a better programmer.

The timeout argument used for the timer events or used as a timer callback.

```
struct timeval timeout = {
    .tv_sec = 0,
    .tv_usec = 250 * 1000,
};

select(1, NULL, NULL, NULL, &timeout);
```

The above code waits for the timeout of 250 milliseconds and the `select` call returns 0. The `select` may not wait for the exact timeout and for this, more accurate timing APIs must be used. such as the `timer_create`, `setitimer`, or `timerfd_create` set of system calls. We will read more on the timers in the **Time and Timers** chapter.

We can use this mechanism to program a timer along with the sockets. We are going to demostrate this feature in the event library section of this book.

The `select` system call cannot serve maximum connections more than `FD_SETSIZE`. On some systems it is 2048. This limits the number of connections when the server program use this call. Thus the `select` call is not a perferred approach when using with a large set of connections that are possibly for the outside of the box.

2. poll system call

3. epoll system call

`epoll` is another API that allows you to selectively wait on a large set of file descriptors. Epoll solves the very less number of client connections with `select` .

The `epoll` is similar to `poll` system call. The system call is used to monitor multiple file descriptors to see if I/O is possible on them.

The `epoll` API can be used either as an edge triggered or level triggered interface and scales well as the file descriptors increase.

The `epoll` API is a set of system calls.

API	description
`epoll_create1`	create an epoll context and return an fd
`epoll_ctl`	register the interested file descriptors
`epoll_wait`	wait for the I/O events

`epoll_create1` creates an epoll context. The context returned is the file descriptor. The file descriptor is then used for the next set of
API. On failure the `epoll_create1` return -1.

The prototype of the `epoll_create1` is as below.

```
int epoll_create1(int flags);
```

The `flags` argument is by default set to 0.

The `epoll_ctl` is a control interface that is used to add, modify or delete a file descriptor. The prototype is as follows.

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

The epfd is the file descriptor returned from the epoll_create1 system call.

The op argument is used to perform the add, modify or delete operations. It can be one of the following.

1. EPOLL_CTL_ADD
2. EPOLL_CTL_DEL
3. EPOLL_CTL_MOD

The event object is used to store the context pointer of the caller.

the struct epoll_event is as follows.

```
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct epoll_event {
    uint32_t events;  // epoll events
    epoll_data_t data; // user data
}
```

The events member can be one of the following.

EPOLLIN: The file descriptor is available for reading.
EPOLLOUT: The file descriptor is available for writing.

The epoll_wait system call waits for events on the returned epoll fd.

The epoll_wait prototype is as follows.

```
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

The timeout argument specifies the timeout in milliseconds to wait.

On success the number of file descriptors ready for the requested I/O are returned, or 0 if none of them are ready during the timeout. On error -1 is returned.

The example of the epoll command is shown below. You can also download it [here (https://github.com/DevNaga/gists/blob/master/epoll_test.c)](https://github.com/DevNaga/gists/blob/master/epoll_test.c)

```c
#include <stdio.h>
#include <string.h>
#include <sys/epoll.h>
#include <unistd.h>

int main(int argc, char **argv[])
{
    int ret;
    int stdin_fd = 0;
    int ep_fd;

    ep_fd = epoll_create1(EPOLL_CLOEXEC);
    if (ep_fd < 0)
        return -1;

    struct epoll_event ep_events;

    memset(&ep_events, 0, sizeof(struct epoll_event));

    ep_events.events = EPOLLIN;
    ep_events.data.fd = stdin_fd;

    if (epoll_ctl(ep_fd, EPOLL_CTL_ADD, stdin_fd, &ep_events) < 0)
        return -1;

    for (;;) {
        int fds;
        struct epoll_event evt;

        fds = epoll_wait(ep_fd, &evt, 1, -1);
        printf("fds %d\n", fds);

        if (evt.data.fd == stdin_fd) {
            printf("read from stdin\n");
            char buf[100];

            memset(buf, 0, sizeof(buf));
            read(stdin_fd, buf, sizeof(buf));
            printf("input %s\n", buf);
        }
    }

    return 0;
}
```

Error codes

Linux system calls returns errors in the form of code stored in `errno` variable. This variable is referenced using the header file <`errno.h`>. The variable is global and be used in safe if the program is using threads.

The error codes are crucial for programs that access critical resources or very important tasks. Depending on the error code returned by the kernel, the userspace can accommodate for the failures.

Here are some of the descriptions of the error codes. As usual, we use the `perror` or `strerror` to describe the error in a string format.

error code	description
EPERM	Permission denied. Special privileges are needed.
ENOENT	The file does not exist.
EIO	Input / Output error occured.
EINTR	Interrupted system call. Signal occured before the system call was allowed to finish.
ENOEXEC	Invalid executable file format. This is detected by the exec family of functions.
ENOMEM	No memory is available. Out of memory situation can get this error.
EBUSY	System resource that can't be shared and already in use.
ENOTDIR	Not a directory. File is given as an argument instead of the directory.
ENODEV	Given device is not found under /dev/ etc.
EEXIST	File exists. A file tried to open in write mode and is already there.
EFAULT	Segmentation fault. Invalid pointer access is detected.
EACCESS	Permission denied. The file permissions on the file does not allow the user to access.
EMFILE	Too many files are opened and can't open anymore.
EISDIR	File is a directory.
EFBIG	File too big.
EADDRINUSE	socket address is already in use.
ENETDOWN	a socket operation failed because the network was down.
ENETUNREACH	a socket operation failed because the network is not reachable.
EADDRNOTAVAIL	socket address that is requested is not available.
EOPNOTSUPP	operation not supported. mostly happens with server and client socket type and family mismatches.
EINPROGRESS	Connect syscall is in in progress to connect with a server.
EAGAIN	Resource temporarily blocked. Try again the call may succeed.
EPIPE	Broken pipe. to ignore it, signal(SIGPIPE, SIG_IGN). Usually a broken pipe means that the other end of the connnection is closed.
EMLINK	too many links
ESPIPE	Invalid seek operation.
ENOTTY	Inappropriate ioctl operation for the tty.
EBADF	File descriptor is bad. The file descriptor is used on the read, write, select, epoll calls is invalid.

Below is a useful error lookup table to map the errors against the strings (Which is what **strerror** usually do)

```c
static struct str_errno_lookup {
    int error;
    char *string;
} table[] = {
    {EPERM, "Permission denied"},
    {ENOENT, "No entry"},
    {EIO, "I/O Error"},
    {EINTR, "Interrupted system call"},
    {ENOEXEC, "Invalid exec format"},
    {ENOMEM, "Out of memory"},
    {EBUSY, "System resource busy"},
    {ENOTDIR, "Not a directory"},
    {ENODEV, "No such device"},
    {EEXIST, "File exist"},
    {EFAULT, "Segmentation Violation"},
    {EACCESS, "Access / Permission denied"},
    {EMFILE, "Too many opened files"},
    {EISDIR, "File is a directory"},
    {EFBIG, "File too big"},
    {EADDRINUSE, "Address already in use"},
    {ENETDOWN, "Network down"},
    {ENETUNREACH, "Network not reachable"},
    {EADDRNOTAVAIL, "Address not available"},
    {EOPNOTSUPP, "Operation not supported"},
    {EINPROGRESS, "Connect system call in progress"},
    {EAGAIN, "Resource temporarily unavailable. Restart the system call"},
    {EPIPE, "Broken pipe"},
    {EMLINK, "Too many symbolic links"},
    {ESPIPE, "Invalid seek operation"},
    {ENOTTY, "Inappropriate ioctl operation for the tty"},
    {EBADF, "Bad file descriptor"}
}

/**
 * @brief - get string based on the error number
 */
char *get_str_lookup(int errno)
{
    int i;

    for (i = 0; i < sizeof(table)/ sizeof(table[0]); i ++) {
        if (table[i].error == errno) {
            return table[i].string;
        }
    }

    return NULL;
}
```

Examples:

Below are some of the useful examples based off of the above described error codes.

1. bad file descriptor: link (https://github.com/DevNaga/gists/blob/master/ebadf.c)

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main(int argc, char **argv)
{
    char hello[] = "hello";
    int fd = -1;

    // writing on a file descriptor thats invalid
    write(fd, hello, sizeof(hello));
    printf("error %d : %s\n", errno, strerror(errno));

    return 0;
}
```

The output looks like the following

```
devnaga@Hanzo:~/gists$ ./a.out
error 9 : Bad file descriptor
devnaga@Hanzo:~/gists$
```

2. permission denied as well address already in use: link (https://github.com/DevNaga/gists/blob/master/ealreadyinuse.c)

```c
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

int main()
{
    int sock;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        return -1;
    }

    int ret;
    struct sockaddr_in serv = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = INADDR_ANY,
        .sin_port = htons(22),
    };

    ret = bind(sock, (struct sockaddr *)&serv, sizeof(serv));
    if (ret < 0) {
        printf("error: %d %s\n", errno, strerror(errno));
        return -1;
    }

    return 0;
}
```

Running the above program without **sudo**:

```
devnaga@Hanzo:~/gists$ ./a.out
error: 13 Permission denied
devnaga@Hanzo:~/gists$
```

Running the above program with **sudo**:

```
devnaga@Hanzo:~/gists$ sudo ./a.out
[sudo] password for devnaga:
error: 98 Address already in use
devnaga@Hanzo:~/gists$
```

network ioctls

Linux operating system supports a wide variety of network ioctls. There are many applications that use these ioctls to perform some very useful operations. One such command is `ifconfig`.

A sample example of an `ifconfig` command looks as below

```
enp0s25: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 68:f7:28:9a:b9:6d  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xf0600000-f0620000
```

The `ifconfig` command also performs a series of ioctls and other operations to get this data.

Let us have a look at the `flags` parameter. The `flags` talk about the state of the network device and the network parameters.

To get the `flags` one must to `SIOCGIFFLAGS` ioctl.

The struct ifreq: This is the base structure that we give out to the kernel for either get / set of network parameters.
This is also what we are going to use right now.

The below is the definition of the `struct ifreq`.

```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* Interface name */
    union {
        struct sockaddr ifr_addr;
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr;
        struct sockaddr ifr_netmask;
        struct sockaddr ifr_hwaddr;
        short           ifr_flags;
        int             ifr_ifindex;
        int             ifr_metric;
        int             ifr_mtu;
        struct ifmap    ifr_map;
        char            ifr_slave[IFNAMSIZ];
        char            ifr_newname[IFNAMSIZ];
        char           *ifr_data;
    };
};
```

below is the example to get the interface flags.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
```

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int sock;
    struct ifreq ifr;

    if (argc != 2) {
        fprintf(stderr, "%s <ifname>\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        return -1;

    memset(&ifr, 0, sizeof(struct ifreq));
    memcpy(ifr.ifr_name, argv[1], IFNAMSIZ - 1);

    if (ioctl(sock, SIOCGIFFLAGS, &ifr) < 0) {
        fprintf(stderr, "failed to get interface flags for %s\n", argv[1]);
        return -1;
    }

    fprintf(stderr, "interface flags %x\n", ifr.ifr_flags);

    fprintf(stderr, "ifflags details: \n");
    if (ifr.ifr_flags & IFF_UP) {
        fprintf(stderr, "\t Up\n");
    }

    if (ifr.ifr_flags & IFF_BROADCAST) {
        fprintf(stderr, "\t Broadcast\n");
    }

    if (ifr.ifr_flags & IFF_MULTICAST) {
        fprintf(stderr, "\t Multicast\n");
    }

    close(sock);

    return 0;
}
```

The above program opens up a DGRAM socket and performs an ioctl to the kernel with the `SIOCGIFFLAGS`
command. This will then be unpacked in the kernel and passed to the corresponding subsystem i.e. the
networking subsystem. The networking subsystem will then validate the fields and the buffers and fills
the data into the buffer i.e `struct ifreq`. Thus the ioctl returns at the userspace with the data.

We then use the `struct ifreq` to unpack and find out the interface flags data. The interface flags
data is stored in the `ifr_flags` field in the `struct ifreq`.

A sample output of the above programs shows us the following:

```
[devnaga@localhost linux]$ ./a.out enp0s25
interface flags 1003
ifflags details:
         Up
         Broadcast
         Multicast
```

Now we have gotten the method to get the interface flags, let's move to `MTU` field in the
`ifconfig` output.

MTU is the largest size of the packet or frame specified in octets. For ethernet, it is mostly 1500
octets.

The below program shows a method to get the MTU of the device.

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int sock;
    struct ifreq ifr;

    if (argc != 2) {
        fprintf(stderr, "%s <ifname>\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        return -1;

    memset(&ifr, 0, sizeof(struct ifreq));
    memcpy(ifr.ifr_name, argv[1], IFNAMSIZ - 1);

    if (ioctl(sock, SIOCGIFMTU, &ifr) < 0) {
        fprintf(stderr, "failed to get MTU for %s\n", argv[1]);
        return -1;
    }

    fprintf(stderr, "MTU of the device is %d\n", ifr.ifr_mtu);

    close(sock);

    return 0;
}
```

In the above program we used SIOCGIFMTU ioctl flag similar to the SIOCGIFFLAGS. We used the same DGRAM
socket and the ioctl returns the data into the `ifr_mtu` field of the `struct ifreq`.

A sample output is shown below:

```
[devnaga@localhost linux]$ ./a.out enp0s25
MTU of the device is 1500
```

Now let us look at the mac address field of the `ifconfig` output. The mac address is the unique 6 byte address that represents
a device in the Layer2. To get this field we should perform `SIOCGIFHWADDR` ioctl.

The below example shows the usage of the `SIOCGIFHWADDR` ioctl.

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int sock;
    struct ifreq ifr;

    if (argc != 2) {
        fprintf(stderr, "%s <ifname>\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        return -1;

    memset(&ifr, 0, sizeof(struct ifreq));
    memcpy(ifr.ifr_name, argv[1], IFNAMSIZ - 1);

    if (ioctl(sock, SIOCGIFHWADDR, &ifr) < 0) {
        fprintf(stderr, "failed to get HWAddress for %s\n", argv[1]);
        return -1;
    }

    uint8_t *hwaddr = ifr.ifr_hwaddr.sa_data;

    fprintf(stderr, "HWAddress: %02x:%02x:%02x:%02x:%02x:%02x\n",
                        hwaddr[0],
                        hwaddr[1],
                        hwaddr[2],
                        hwaddr[3],
                        hwaddr[4],
                        hwaddr[5]);

    close(sock);

    return 0;
}
```

The above program does the similar top level jobs such as opening a socket, closing a socket and using the `struct ifreq`.
The ifr_hwaddr contains the hardware address of the network device. The `ifr_hwaddr` is of type `struct sockaddr`.

The `struct sockaddr` contains its data member `sa_data` which inturn is the hardware address that the kernel copied.

The sample output of the program is shown below:

```
[devnaga@localhost linux]$ ./a.out enp0s25
HWAddress: 68:f7:28:9a:b9:6d
```

The set of hardware address is also possible via the `SIOCSIFHWADDR` ioctl. Here is the example

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if_arp.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int sock;
    int ret;
    struct ifreq ifr;

    if (argc != 3) {
        fprintf(stderr, "%s <ifname> <HWAddress>\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        return -1;

    memset(&ifr, 0, sizeof(struct ifreq));

    int hwaddr_i[6];

    ret = sscanf(argv[2], "%02x:%02x:%02x:%02x:%02x:%02x",
                           &hwaddr_i[0],
                           &hwaddr_i[1],
                           &hwaddr_i[2],
                           &hwaddr_i[3],
                           &hwaddr_i[4],
                           &hwaddr_i[5]);

    strcpy(ifr.ifr_name, argv[1]);

    ifr.ifr_hwaddr.sa_data[0] = hwaddr_i[0];
```

```
    ifr.ifr_hwaddr.sa_data[1] = hwaddr_i[1];
    ifr.ifr_hwaddr.sa_data[2] = hwaddr_i[2];
    ifr.ifr_hwaddr.sa_data[3] = hwaddr_i[3];
    ifr.ifr_hwaddr.sa_data[4] = hwaddr_i[4];
    ifr.ifr_hwaddr.sa_data[5] = hwaddr_i[5];

    ifr.ifr_addr.sa_family = AF_INET;
    ifr.ifr_hwaddr.sa_family = ARPHRD_ETHER;

    ret = ioctl(sock, SIOCSIFHWADDR, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set HWAddress for %s\n", argv[1]);
        perror("ioctl");
        return -1;
    }

    close(sock);

    return 0;
}
```

The above program reads the interface name and the mac address from the command line. It then copies into the
`struct ifreq`. The interface name is copied to the `ifr_name` flag, the family type is put into `ifr.ifr_addr.sa_family`
and is `AF_INET`. We also need to set the family in the `ifr.ifr_hwaddr.sa_family` member as the same `AF_INET`. The mac is
then copied into `ifr.ifr_hwaddr.sa_data` member.

The sample command to set the mac is below.

```
[root@localhost devnaga]# ./a.out enp0s25 00:ff:31:ed:ff:e1

[root@localhost devnaga]# ifconfig enp0s25
enp0s25: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 00:ff:31:ed:ff:e1  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xf0600000-f0620000
```

To get the interface index of a particular network interface, `SIOCGIFINDEX` is used.

Below is an example of getting an index from a particular interface. Download [here (https://github.com/DevNaga/gists/blob/master/getifindex.c)](https://github.com/DevNaga/gists/blob/master/getifindex.c)

```c
#include <stdio.h>
#include <sys/socket.h>
#include <net/if.h>
#include <linux/ioctl.h>
#include <errno.h>
#include <string.h>
#include <sys/ioctl.h>

int main(int argc, char **argv)
{
    int fd;

    if (argc != 2) {
        fprintf(stderr, "<%s> interface name\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    struct ifreq ifr;

    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);

    int ret;

    ret = ioctl(fd, SIOCGIFINDEX, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to get ifindex %s\n", strerror(errno));
        return -1;
    }

    printf("interface index for [%s] is %d\n", argv[1], ifr.ifr_ifindex);

    return 0;
}
```

Below example is another one of SIOCGIFFLAGS. Download [here](https://github.com/DevNaga/gists/blob/master/siocgifflags.c)

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
```

```c
#include <net/if.h>

int main(int argc, char **argv)
{
    struct ifreq ifr;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(fd, SIOCGIFFLAGS, &ifr);
    if (ret < 0) {
        return -1;
    }

    printf("ifname : %s\n", argv[1]);
    if (ifr.ifr_flags & IFF_UP) {
        printf("UP\n");
    }

    if (ifr.ifr_flags & IFF_BROADCAST) {
        printf("BROADCAST\n");
    }

    if (ifr.ifr_flags & IFF_LOOPBACK) {
        printf("LO\n");
    }

    if (ifr.ifr_flags & IFF_RUNNING) {
        printf("RUNN\n");
    }

    if (ifr.ifr_flags & IFF_PROMISC) {
        printf("PROMISC\n");
    }

    if (ifr.ifr_flags & IFF_MULTICAST) {
        printf("MCAST\n");
    }
```

```
    return 0;
}
```

there is another way to get the interface index. This can be done using the `if_nametoindex` function.

Below is an example of `if_nametoindex`. Download here (https://github.com/DevNaga/gists/blob/master/ifnametoindex.c)

```
#include <stdio.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int id;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    id = if_nametoindex(argv[1]);
    printf("index %d\n", id);

    return 0;
}
```

The function `if_nametoindex` is declared in `net/if.h`.

The ioctl `SIOCGIFBRDADDR` allows to get the broadcast address of network interface.

Below is an example. Download here (https://github.com/DevNaga/gists/blob/master/siocgifbrdaddr.c)

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <errno.h>

int main(int argc, char **argv)
{
    struct ifreq ifr;
    char *braddr;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        fprintf(stderr, "failed to socket %s\n", strerror(errno));
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));
    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(fd, SIOCGIFBRDADDR, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to ioctl %s\n", strerror(errno));
        return -1;
    }

    braddr = inet_ntoa(((struct sockaddr_in *)&(ifr.ifr_broadaddr))->sin_addr);
    if (!braddr) {
        fprintf(stderr, "failed to inet_ntoa %s\n", strerror(errno));
        return -1;
    }

    printf("broadcast %s\n", braddr);

    close(fd);

    return 0;
}
```

when the network interface does not have an IP address the above ioctl might fail with an error `cannot assign requested address`. It may be useful in cases of diagnostics.

The ioctl `SIOCGIFNETMASK` is used to get the network mask of an interface.

Below is an example of `SIOCGIFNETMASK`. Download [here (https://github.com/DevNaga/gists/blob/master/siocgifnetmask.c)](https://github.com/DevNaga/gists/blob/master/siocgifnetmask.c)

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>

int main(int argc, char **argv)
{
    char *netmask;
    struct ifreq ifr;
    int fd;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> ifname\n", argv[0]);
        return -1;
    }

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    memset(&ifr, 0, sizeof(ifr));
    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(fd, SIOCGIFNETMASK, &ifr);
    if (ret < 0) {
        return -1;
    }

    netmask = inet_ntoa(((struct sockaddr_in *)&(ifr.ifr_netmask))->sin_addr);
    if (!netmask) {
        return -1;
    }

    printf("netmask %s\n", netmask);

    close(fd);

    return 0;
}
```

There is also a way to change the interface name. This is done using the SIOCSIFNAME ioctl.

Below is the example that demonstrates the SIOCSIFNAME. Download [here (https://github.com/DevNaga/gists/blob/master/setifname.c)](https://github.com/DevNaga/gists/blob/master/setifname.c)

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if_arp.h>
#include <net/if.h>

int main(int argc, char **argv)
{
    int sock;
    int ret;
    struct ifreq ifr;

    if (argc != 3) {
        fprintf(stderr, "%s <ifname> <new-ifname>\n", argv[0]);
        return -1;
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        return -1;

    memset(&ifr, 0, sizeof(struct ifreq));

    strcpy(ifr.ifr_name, argv[1]);

    ret = ioctl(sock, SIOCGIFFLAGS, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to get interface flags for %s\n", argv[1]);
        return -1;
    }

    ifr.ifr_flags &= ~IFF_UP;

    ret = ioctl(sock, SIOCSIFFLAGS, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set interface flags for %s\n", argv[1]);
        return -1;
    }

    strcpy(ifr.ifr_newname, argv[2]);

    ret = ioctl(sock, SIOCSIFNAME, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set interface name for %s\n", argv[1]);
        perror("ioctl");
```

```
        return -1;
    }

    strcpy(ifr.ifr_name, argv[2]);

    ifr.ifr_flags |= IFF_UP;

    ret = ioctl(sock, SIOCSIFFLAGS, &ifr);
    if (ret < 0) {
        fprintf(stderr, "failed to set interface flags for %s\n", argv[1]);
        return -1;
    }

    close(sock);

    return 0;
}
```

The programs makes the interface go down, otherwise we cannot change the name of the interface.
The interface is made down using the `SIOCSIFFLAGS` ioctl and sets up the interface name and makes the interface up again using the `SIOCSIFFLAGS`.

wireless ioctls

The package [wireless-tools (http://www.labs.hpe.com/personal/Jean_Tourrilhes/Linux/Tools.html#latest)](http://www.labs.hpe.com/personal/Jean_Tourrilhes/Linux/Tools.html#latest) provides the needed API to perform many wireless functions.

time

Linux reads time from the RTC hardware clock if its available. The clock runs indefinitely as long as the battery is giving the power (even though the system is powered down). Otherwise it starts the system from JAN 1 2000 00:00 hrs. (As i saw it from the 2.6.23 kernel) (Needs updating)

The resolution is in seconds from the RTC hardware.

The linux kernel then keeps the read time from the RTC into the software and keeps it ticking till it gets a reboot or shutdown signal or the power down interrupt.

The kernel's time management system provides a clock resolution of nano seconds.

Kernel maintains a software timer called jiffies that is measured in ticks. The jiffies are the number of ticks that have occurred since the system booted.

The system call to get the current time in seconds is `time`. The data type `time_t` is available to store the time.

The below code gets the current time in seconds since 1970 UTC JAN 1.

```
time_t now;

now = time(0);
```

the now variable holds the current time in seconds. The `time_t` is typecasted from `long` type. Thus it is printable as the long type.

```
printf("the current system time in sec %ld\n", now);
```

The header file to include when using the `time` system call is `time.h`.

The system call to get the current date and time in human readable format is the `gmtime` and friends (`asctime` and `localtime`)

The `gmtime` takes a variable of type `time_t` as an address and returns a data structure of type `struct tm`. The time value returned in the form of UTC from 1970 Jan 1.

The `struct tm` contains the below fields

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

A call to the gmtime involve getting the time from the `time` API.

the field `tm_sec` will have a range of 0 to 59.
the field `tm_min` will have a range of 0 to 59.
the field `tm_hour` will have a range of 0 to 23.
the field `tm_mday` is day of the month will have a range of 0 to 31. and depending on the month (for Feb or for leap year).
the field `tm_mon` will have a range of 0 to 11. 0 being january and 11 being december.
the field `tm_year` will be subtracted by 1900. usually if its 2018, it will be shown as 118.
the field `tm_wday` is day of the week.

the field `tm_isdst` represents if the current time has a DST (Day light savings) in use.

the below example gives an idea on how to use the gmtime function in a more basic way.

```c
struct tm *t;
time_t cur;

cur = time(0); // current time

t = gmtime(&cur);
if (!t) {
    perror("failed to gmtime");
    return -1;
}

printf("t->year %d\n"
       "t->mon %d\n",
       "t->date %d\n"
       "t->hour %d\n"
       "t->min %d\n"
       "t->sec %d\n",

       t->tm_year + 1900, // the year is addded with 1900 to get to current
year
       t->tm_mon + 1, // month starts from 0
       t->tm_mday, // day of the month, the Calendar
       t->tm_hour,
       t->tm_min,
       t->tm_sec);
```

Below example provide the gmtime API usage. Download [here (https://github.com/DevNaga/gists/blob/master/gmtime.c)](https://github.com/DevNaga/gists/blob/master/gmtime.c)

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

int main()
{
    time_t now;
    struct tm *t;

    now = time(0);

    t = gmtime(&now);
    if (!t) {
        printf("failure to gmtime %s\n", strerror(errno));
        return -1;
    }

    printf("%04d:%02d:%02d-%02d:%02d:%02d\n",
                    t->tm_year + 1900,
                    t->tm_mon + 1,
                    t->tm_mday,
                    t->tm_hour,
                    t->tm_min,
                    t->tm_sec);

    return 0;
}
```

`gmtime` has the structure `struct tm` as static in its code. It is usually not safe to call `gmtime` with threads. A more safer approach is to call the reentrant version of `gmtime`, i.e. `gmtime_r`.

Below example provide the `gmtime_r` API. Download here (https://github.com/DevNaga/gists/blob/master/gmtime1.c)

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

int main()
{
    time_t now;
    struct tm t1;
    struct tm *t;

    now = time(0);

    t = gmtime_r(&now, &t1);
    if (!t) {
        printf("failure to gmtime %s\n", strerror(errno));
        return -1;
    }

    printf("%04d:%02d:%02d-%02d:%02d:%02d\n",
                    t->tm_year + 1900,
                    t->tm_mon + 1,
                    t->tm_mday,
                    t->tm_hour,
                    t->tm_min,
                    t->tm_sec);

    return 0;
}
```

`ctime` is another API that can return time in calendar time printable format (string) according to the time size.

`ctime` prototype is as follows.

```c
char *ctime(const time_t *t);
```

Below is an example of `ctime` API. Download [here](https://github.com/DevNaga/gists/blob/master/ctime.c)

```c
#include <stdio.h>
#include <time.h>

int main()
{
    time_t now = time(NULL);

    printf("current cal time %s\n", ctime(&now));
    return 0;
}
```

mktime is an API that converts the time in `struct tm` format into `time_t`. The prototype is as follows.

```
time_t mktime(struct tm *t);
```

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

int main(int argc, char **argv)
{
    int year, month, date, hr, min, sec;
    struct tm t;
    time_t result;

    if (argc != 7) {
        printf("%s [year] [month] [date] [hour] [minute] [second]\n", argv[0]);
        return -1;
    }

    year = atoi(argv[1]);
    month = atoi(argv[2]);
    date = atoi(argv[3]);
    hr = atoi(argv[4]);
    min = atoi(argv[5]);
    sec = atoi(argv[6]);

    if (month < 0 || month > 12) {
        printf("out of range month %d\n", month);
        return -1;
    }

    if (date < 0 || date > 31) {
        printf("out of range date %d\n", date);
        return -1;
    }

    if (hr < 0 || hr > 24) {
        printf("out of range hour %d\n", hr);
        return -1;
    }

    if (min < 0 || min > 60) {
        printf("out of range minute %d\n", min);
        return -1;
    }

    if (sec < 0 || sec > 60) {
        printf("out of range second %d\n", sec);
        return -1;
    }
```

```
    t.tm_year = year - 1900;
    t.tm_mon = month - 1;
    t.tm_mday = date;
    t.tm_hour = hr;
    t.tm_min = min;
    t.tm_sec = sec;
    t.tm_isdst = -1;

    result = mktime(&t);
    if (result == -1) {
        printf("Failed to get mktime\n");
        return -1;
    }

    printf("res %ld\n", result);

    return 0;
}
```

Example: mktime

The `t.tm_isdst` is set to -1 as we do not know the timezone.

A more resolution timeout can be obtained from the `gettimeofday` API. The API looks like below:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

The second argument is usually passed NULL for getting the time since UTC.

The `gettimeofday` is used to get the microsecond resolution time as well as the timezone. The `gettimeofday` returns 0 on success and -1 on failure.

The value is returned into `struct timeval`. The `struct timeval` is as follows.

```
struct timeval {
    time_t  tv_sec;
    suseconds_t tv_usec;
};
```

where the `tv_sec` represents the seconds part of the time and `tv_usec` represents the microseconds part of the time.

We simply use the below code to get the current time in seconds and micro seconds resolution.

```
struct timeval cur_time;
int ret;

ret = gettimeofday(&cur_time, NULL);
if (ret < 0) { // most unlikely the call will fail
    perror("failed to gettimeofday");
    return -1;
}

printf("cur time sec: %ld, usec: %ld\n",
                    cur.tv_sec, cur.tv_usec);
```

The most common use of `gettimeofday` is to put the call above and below a function cal, analyze how much time the function call would take to execute.

An example would look as follows.

```
void function()
{
    ...
}

void analysis()
{
    long diff;
    struct timeval before, after;

    gettimeofday(&before, 0);
    function();
    gettimeofday(&after, 0);

    diff = (((after.tv_sec * 1000) - (before.tv_sec * 1000)) +
               (after.tv_usec / 1000) - (before.tv_sec / 1000))

    printf("delta %ld\n", diff);
}
```

The above calls would get the current 'wallclock' time. Meaning they are affected by the changes in the time due to clock drift and adjustments. The most important factors include the GPS setting the time into the system, NTP changing the system time syncing with the NTP servers. This would affect programs depending on these API. For example: the timers using the above API would either expire quickly (due to time moving forward) or wait forever (due to time moving backwards to a larger value).

The header file `sys/time.h` also provides a macro called `timersub`. The `timersub` accepts two `timeval` structures and produces the delta in the third varible that is also of type `timeval`.

Below is the `timersub` prototype looks like,

```
timersub(struct timeval *stop, struct timeval *start, struct timeval *delta);
```

with the timersub the above diff calculation can be done simply as below, (diff can be replaced with timersub)

```
struct timeval before, after;
struct timeval delta;

...

timersub(&after, &before, &delta);

printf("latency %ld sec %ld usec\n", delta.tv_sec, delta.tv_usec);
```

The settimeofday API is used to set the system time. The prototype is as follows..

```
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

just like gettimeofday, the settimeofday tz argument can be set to NULL by default.

The settimeofday API can fail in the following cases:

EPERM:

if the user is not privileged user and tries to call this API.

ENOSYS:

If the tz pointer in the call is not null and the os does not supports it.

The following code snippet describes the usage of the settimeofday system call.

```
struct timeval tv;
int ret;

ret = gettimeofday(&tv, 0);
if (ret < 0) {
    perror("failed to gettimeofday");
    return -1;
}

// set one sec in future
tv.tv_sec += 1;
ret = settimeofday(&tv, 0);
if (ret < 0) {
    perror("failed to settimeofday");
    return -1;
}
```

The problem with the settimeofday is that the time can go abruptly forward or abruptly backwards. This might affect some programs as we have discussed above that programs using wallclock time might misbehave with the abrupt change of time. To avoid this process we need to use the adjtime API which is described as follows.

The adjtime looks as follows

```
int adjtime(const struct timeval *delta, struct timeval *olddelta);
```

The adjtime API speeds up or slows down the time in monotonically. If the delta argument is positive, then the system time is speeded up till the delta value and if the delta argument is negative, then the system time is slowed down till the delta value.

The below code sample shows the usage of adjtime.

```
int ret;
struct timeval delta;

delta.tv_sec = 1;
delta.tv_usec = 0;

ret = adjtime(&delta, NULL);
if (ret < 0) {
    perror("failed to adjtime");
    return -1;
}
```

When we are programming timers, we should avoid any calls to the above API as they are not monotonic or steadily moving forward in the future.

The time.h provides a macro called difftime that is used to find the difference of time between two variables of type time_t (Although one can subtract the two from each other on linux system).

The difftime looks as follows.

```
double difftime(time_t time0, time_t time1);
```

clock is another API that measures the CPU time perfectly. As we looked at one of the usage of gettimeofday call, we provided an example of using the gettimeofday to measure the time it takes to execute the function. However, this incurs the scheduling and other jobs with in the system and is not the effective way to find out only the CPU time. clock function provides us to do just this.

The clock looks as follows.

```
clock_t clock();
```

The clock returns the number of ticks. To convert it, divide it by CLOCKS_PER_SEC. If the processor time used is not represented, it returns -1. The clock return value can wrap around every 72 minutes. On a 32 bit system the CLOCKS_PER_SEC is 1,000,000.

The sample code is as below.

```
clock_t start;
clock_t end;

start = clock();
func_call();
end = clock();

printf("ticks %d\n", end - start);
```

There is another API that is used to get the CPU times, called `times`.

The prototype is as follows

```
clock_t times(struct tms *buf);
```

The `times` API stores the information into the `struct tms`. The structure looks as below.

```
struct tms {
    clock_t tms_utime; // user time
    clock_t tms_stime; // system time
    clock_t tms_cutime; // user time of children
    clock_t tms_cstime; // system time of children
};
```

`tms_utime` is the amount of time spent in executing the instructions in user space.
`tms_stime` is the amount of time spent in executing the instructions in system.
`tms_cutime` is the amount of time spent by the children executing the instructions in user space.
`tms_cstime` is the amount of time spent by the children executing the instructions in system.

All the above times are units of clock ticks.

Timer APIs

Linux supports the following timer API.

```
1. setitimer
2. timer_create
3. timerfd_create
```

The command `hwclock` is very useful to get or set time to the system RTC hardware clock.

`hwclock` is also used to correct time drifts with the UTC. A periodic (deterministic timeout) set would allow the system to be in sync with the UTC time.

`hwclock` command has the following functions:

```
-r, --show

Read the hardware clock and print the time on standard output.

--hctosys

set the system time from the hardware clock.

--systohc

Set the system time to the hardware clock.
```

timers

A timer counts down to 0 from a specific value. In operating systems, the timer upon expiry, allows a program to perform specific actions. There are two timers, specifically one shot and periodic.

A one shot timer runs only once. A periodic timer repeats itself upon every expiration. Some programs need the timer to be tick at smaller intervals and with lesser resolutions. the `time_t` variable can be used for this purpose. The `time_t` is of resolution in seconds.

alarm

`alarm()` arranges for a `SIGALRM` signal to be delivered to the calling process. The `alarm` can be thought of as one shot timer.

Below example provides a oneshot timer implementation with the `alarm`. The weblink to the program is [here (https://github.com/DevNaga/gists/blob/master/alarm_book.c)](https://github.com/DevNaga/gists/blob/master/alarm_book.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>

void sigalrm_handler(int sig)
{
    printf("Alarm signal\n");
}

int main(int argc, char **argv)
{
    signal(SIGALRM, sigalrm_handler);
    alarm(2);
    while(1);
    return 0;
}
```

compile and run the program as follows.

```
root@b516cef12271:~/books# gcc alarm.c
root@b516cef12271:~/books# ./a.out
Alarm signal
```

Removing the `while (1);` would make the program stop instead of waiting in the alarm. This means that the alarm function registers a timer in the kernel and returns. The kernel, upon a timer expiry, triggers a SIGALRM signal that, when handled by a program, the signal handler is invoked.

The waiting is done in the `while` statement so as to allow the kernel to trigger the timer for this running process.

setitimer

The `setitimer` API provides either a one shot or an interval timer. When the timer fires, the OS activates the SIGALRM signal for each expiry. Before the `setitimer` we register a signal handler for the SIGALRM. The `setitimer` thus invokes the signal handler indirectly upon each expiry.

the `setitimer` API prototype is as follows.

```
int setitimer(int which,
              const struct itimerval *cur,
              struct itimerval *old);
```

the `which` argument takes one of the three aguments which are `ITIMER_REAL`, `ITIMER_VIRTUAL` and `ITIMER_PROF`. Generally, for a system timer we use `ITIMER_REAL`.

The second argument is the structure pointer of type `struct itimerval`. The structure contains the following.

```
struct itimerval {
    struct timeval it_interval;
    struct timeval it_value;
};

struct timeval {
    time_t tv_sec;
    suseconds_t tv_usec;
};
```

The structure contains an it_value member describing the initial value for the timer and an it_interval member that is used as a repeatable value. The timer is initialized with the it_value and when the timer expires, the it_interval is loaded as a next expiry value and is repeated again and again.

for ex:

```
struct itimerval it = {
    .it_interval.tv_sec = 1;
    .it_interval.tv_usec = 0;
    .it_value.tv_sec = 2;
    .it_value.tv_usec = 0;
};
```

The it_value is initialized to 2 seconds and so after the 2 secs the timer expires and calls the signal handler that is registered. upon the expiry, the it_interval is loaded into the timer as the new expiry time that is 1 sec. At every expiry the 1 sec timeout value is loaded back as a next triggering timeout.

The `setitimer` behaves as a one shot timer when the it_interval argument is 0.

Below is the example of the `setitimer`.

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>

void signal_handler(int sig)
{
    struct timeval tv;

    gettimeofday(&tv, 0);
    printf("timer interrupt %ld.%ld\n", tv.tv_sec, tv.tv_usec / 1000);
}

int main(int argc, char **argv)
{
    int ret;
    int timerval;
    int repeat = 0;

    struct sigaction new_sig;

    if (argc != 3) {
        fprintf(stderr, "%s <msec timeout value> <repeat enable (1 / 0)>\n",
argv[0]);
        return -1;
    }

    timerval = atoi(argv[1]);
    repeat = atoi(argv[2]);

    memset(&new_sig, 0, sizeof(new_sig));

    new_sig.sa_handler = signal_handler;
```

```
    ret = sigaction(SIGALRM, &new_sig, NULL);
    if (ret != 0) {
        fprintf(stderr, "failed to setup SIGACTION for SIGALRM\n");
        return -1;
    }

    if (timerval != 0) {
        struct itimerval iv;

        memset(&iv, 0, sizeof(iv));

        iv.it_value.tv_sec = 0;
        iv.it_value.tv_usec = timerval * 1000;
        if (repeat) {
            iv.it_interval.tv_sec = 0;
            iv.it_interval.tv_usec = timerval * 1000;
        }

        ret = setitimer(ITIMER_REAL, &iv, NULL);
        if (ret != 0) {
            perror("setitimer");
            fprintf(stderr, "failed to setitimer\n");
            return -1;
        }
    }

    while (1);

    return 0;
}
```

The program first registers the SIGALRM signal via `sigaction` and registers the `setitimer` with the given input values. The periodicity of the timeout is controlled via the repeat argument.

timer_create

timerfd

1. timerfd_create

`timerfd_create` notifies the timer events via the file descriptors. The returned file descriptor is watchable via `select` or `epoll` calls.

The prototype is as follows.

```
int timerfd_create(int clockid, int flags);
```

The `clockid` refers to the clock that can be used to mark the timer progress. It is one of the `CLOCK_REALTIME` or `CLOCK_MONOTONIC`. Monotonic clock is a non settable clock that progresses constantly over the time. Meaning that the clock will stay constant although there are changes in the system time.

The flags argument is usually kept 0 by default.

2. timerfd_settime

The timerfd_settime arms or disrams the timer value referred by the file descriptor.

The timerfd_settime prototype is as follows.

```
int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
                    struct itimerspc *old_value);
```

The itimerspec looks as below.

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};

struct itimerspec {
    struct timerspec it_interval;
    struct timerspec it_value;
}
```

The it_value is taken as initial value of the timer. As soon as the timer is fired, the it_interval is stored into the timer as the new value. If it_interval is set to 0, then the timer becomes a oneshot timer.

The below program explains the timerfd_ calls.

```

#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>
#include <stdint.h>
#include <time.h>
#include <stdlib.h>
#include <sys/timerfd.h>
#include <sys/select.h>

int main(int argc, char **argv)
{
    int time_intvl;
    int ret;
    int fd;

    if (argc != 2) {
        printf("%s [timer interval in msec]\n", argv[0]);
        return -1;
    }

    time_intvl = atoi(argv[1]);

    fd = timerfd_create(CLOCK_MONOTONIC, 0);
```

```
    if (fd < 0) {
        printf("failed to timerfd_create\n");
        return -1;
    }

    struct itimerspec it = {
            .it_value = {
                .tv_sec = 0,
                .tv_nsec = 1000 * 1000ULL * time_intvl,
            },
            .it_interval = {
                .tv_sec = 0,
                .tv_nsec = 1000 * 1000ULL * time_intvl,
            },
    };

    ret = timerfd_settime(fd, 0, &it, 0);
    if (ret < 0) {
        printf("failed to timerfd_settime\n");
        return -1;
    }

    printf("fd %d\n", fd);
    struct timeval tv;
    fd_set rdfd;

    while (1) {
        FD_ZERO(&rdfd);
        FD_SET(fd, &rdfd);

        ret = select(fd + 1, &rdfd, NULL, NULL, NULL);
        if (ret > 0) {
            if (FD_ISSET(fd, &rdfd)) {
                uint64_t expiration;

                ret = read(fd, &expiration, sizeof(expiration));
                if (ret > 0) {
                    gettimeofday(&tv, 0);
                    printf("interval timer %ld.%ld, expirations %ju\n",
tv.tv_sec, tv.tv_usec, expiration);
                }
            }
        }
    }

    return 0;
}
```

The expiration indicated by an event from wait mechanism i.e. either `select` or `epoll` and the value is read from the `read`system call. The 8 byte value shall be read describing the number of expirations that have occured.

Message queues

Message queues are another form of IPC.

Linux implements a new way to program the message queues. The interface is called as `mq_overview`.

Header file to include `<mqueue.h>`.

The manual page of `mq_overview` defines a set of API.

API Name	description
mq_open	create / open a message queue
mq_send	write a message to a queue
mq_receive	read a message from a queue
mq_close	close a message queue
mq_unlink	removes a message queue name and marks it for deletion
mq_getattr	get attributes of message queue
mq_setattr	set attributes of message queue

To use the message queues, link with `-lrt`.

mq_open

message queues are created and opened using `mq_open`. This function returns a message queue descriptor of type `mqd_t` which is a file descriptor. This file descriptor can be used as an input to the `select` system call to selectively wait.

Each message queue is identified by a name of the form /name. The maximum length of the name field is 255 bytes.

The prototype of the `mq_open` is as follows:

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

As we can see that the above are two different API declarations. This means that the `mq_open` must be a variable argument function and the prototypes are given to simply the usage.

Here is the example program that uses the `mq_open` API.

```
mqd_t server_fd;
struct mq_attr mq_attr;

server_fd = mq_open("/mq1", O_CREAT | O_RDWR, 0644, &mq_attr);
if (server_fd < 0) {
    printf("failed to open server fd\n");
    perror("mq_open");
    return -1;
}

printf("server fd %d\n", server_fd);
```

The name is the name of the message queue that we are going to use. If the message queue is already not created, it has to be created. To create it, we must provide the following to the oflag field: The O_RDWR | O_CREAT. Meaning, create a message queue (O_CREAT) for sending and receiving (O_RDWR).

If the queue is already created, we just have to open it. The process can simply open it for receive only or cna be opened in sending and receiving modes.

In the create mode of operation, the mode argument must contain the permission bits. The permissions bits are usually 0644. Along with the permission bits, one must also specify the attributes of the message queue. The attribute structure struct mq_attr is to be filled. The structure looks as the following:

```
struct mq_attr {
      long mq_flags;    // 0 or O_NONBLOCK
      long mq_maxmsg;   // maximum number of messages that go into the queue
      long mq_msgsize;  // maximum message size in bytes
      long mq_curmsg;   // number of messages currently queued
};
```

The server program calls the mq_open with the following args. Usually the server program creates a message queue and the client program opens the message queue.

The mq_getattr is used to get attributes of the message queue into the above structure and mq_setattr is used to set attributes of the message queue.

mq_send

mq_send sends a message to the message queue. Here is the prototype.

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int
msg_prio);
```

mq_receive

mq_close

mq_getattr

mq_setattr

Below is one example of the message queues. The server side creates a message queue with mq_open and sets up the queues. The server then waits for the message in mq_receive.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <errno.h>

mqd_t server_fd;
int len;
```

```c
struct mq_attr attr;

int main()
{
    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 64;
    attr.mq_curmsgs = 0;

    server_fd = mq_open("/mq1", O_CREAT | O_RDWR, 0644, &attr);
    if (server_fd < 0) {
        printf("failed to open server fd\n");
        perror("mq_open");
        return -1;
    }

    struct mq_attr new_attr;

    mq_getattr(server_fd, &new_attr);

    printf("flags %d maxmsg %d msgsize %d curmsg %d\n",
                    new_attr.mq_flags,
                    new_attr.mq_maxmsg,
                    new_attr.mq_msgsize,
                    new_attr.mq_curmsgs);
    printf("server fd %d\n", server_fd);

    while (1) {
        char buf[8192 * 2];

        len = mq_receive(server_fd, buf, sizeof(buf), NULL);
        if (len < 0) {
            printf("failed to receive from the mqueue\n");
            perror("mq_receive");
            return -1;
        }

        printf("received msg with len %u message %s\n", len, buf);
    }

    mq_close(server_fd);

    return 0;
}
```

Example: message queue server program

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <errno.h>

mqd_t client_fd;
int len;
struct mq_attr attr;

int main()
{
    client_fd = mq_open("/mq1", O_RDWR);
    if (client_fd < 0) {
        printf("failed to open server fd\n");
        perror("mq_open");
        return -1;
    }

    char buf[8192 * 2];

    while (1) {
        strcpy(buf, "Hello");

        len = mq_send(client_fd, buf, strlen(buf) + 1, 1);
        if (len < 0) {
            printf("failed to send to the mqueue\n");
            perror("mq_send");
            return -1;
        }
        sleep(1);
    }

    mq_close(client_fd);

    return 0;
}
```

Example: message queue client program

Compile the server program as `gcc mq_server.c -lrt -o mq_server` and client program as `gcc mq_client.c -lrt -o mq_client`. Run the server program first and then the client to receive the messages.

Shared memory

system V shared memory

The shared memory is one of the quickest forms of IPC that can be used between the processes. Since the memory is common between the two programs (or more than two) it is a must to protect it from being accessed parallely at the same time causing the corruption. Thus, we need to use some form of locking (such as the semaphores or events). The method of creating and communicating via the shared memory is as follows.

- A process creates a shared memory segment with a unique key value
- The process then attaches to it.
- Another process, knowing the unique key value, attaches to the shared memory segment.
- Now the two processes can communicate (transfer the data between each other) using the shared memory.

To create a shared memory, the Linux OS provides shared memory API as the following.

shm API	description
shmget	allocate shared memory segment
shmat	attach to the shared memory with the given shared memory identifier
shmctl	perform control operations on the shared memory segment
shmdt	detaches from the shared memory

To use the above API we must include `<sys/ipc.h>` and `<sys/shm.h>` header files.

`shmget` is used to create shared memory segments.

The `shmget` prototype is as follows.

```
int shmget(key_t key, size_t size, int shmflg);
```

`shmget` returns the shared memory ID on success.

- The first argument key must be unique. This key can be generated using the `ftok()` call.
- The `size` argument is the size of the shared memory segment (it is rounded to the multiples of PAGE_SIZE. Usually PAGE_SIZE is 4k).
- The `shmflg` is usually set with the IPC_CREAT flag.
- If the key already exist, the `errno` is set to EEXIST and returns -1.

The below is an example to create the shared memory segment. The key is taken to be static number for the example.

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid;
    key_t key = 0x01020304;

    // create 4096 bytes of shared memory
    shmid = shmget(key, 4096, IPC_CREAT);
    if (shmid < 0) {
        fprintf(stderr, "failed to create shm segment\n");
        perror("shmget");
        return -1;
    }

    printf("created %d\n", shmid);
    return 0;
}
```

We compile and execute the program and on success it prints the last print statement i.e. created 993131 (some number that is the shm id).

an ipcs -m command on the created shared memory shows me this.

```
dev@hanzo:~$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 65536      lightdm    600        524288     2          dest
0x00000000 163841     lightdm    600        524288     2          dest
0x00000000 196610     lightdm    600        33554432   2          dest
0x01020304 229379     dev        0          4096       0
```

The API `shmat` performs the attachement to the shared memory segment. Its prototype is as following.

```c
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- the first argument `shmid` is the id returned from shmget.
- the second argument is the attach address, and is usually kept to NULL.
- the shmflg is also kept to 0 when doing read and write operations on the shared memory.

On success `shmat` returns the address of the segment and on failure it returns a value -1 and the value to be type casted to an integer to check for the failures.

Let us write two programs, one is the program that creates the shared memory, attaches to it and writes "Hello" to the memory. The another program attaches to the memory based on the key and reads from the memory and prints the contents on to the console.

Server code

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid;
    key_t key = 0x01020304;
    void *addr;

    shmid = shmget(key, 4096, IPC_CREAT);
    if (shmid < 0) {
        fprintf(stderr, "failed to create shm segment\n");
        perror("shmget");
        return -1;
    }

    printf("created %d\n", shmid);

    addr = shmat(shmid, NULL, 0);
    if ((int)addr == -1) {
        fprintf(stderr, "failed to attach\n");
        perror("shmat");
        return -1;
    }

    printf("got %p\n", addr);

    char *data = addr;

    strcpy(data, "Hello");

    while(1);
    return 0;
}
```

Client code

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid;
    key_t key = 0x01020304;
    void *addr;

    shmid = shmget(key, 4096, 0);
    if (shmid < 0) {
        fprintf(stderr, "failed to create shm segment\n");
        perror("shmget");
        return -1;
    }

    printf("found %d\n", shmid);

    addr = shmat(shmid, NULL, 0);
    if ((int)addr == -1) {
        fprintf(stderr, "failed to attach\n");
        perror("shmat");
        return -1;
    }

    printf("got %p\n", addr);

    char *data = addr;

    printf("Data %s\n", data);

    return 0;
}
```

We compile the two programs and create the binaries as `shmsrv` and `shmcli`. We run the `shmsrv` first and then `shmcli` next. The `shmsrv` program performs the write to the shared memory segment and runs into infinite loop while the `shmcli` program performs a read on the shared memory segment and prints the data `Hello` on to the screen.

Let us run the `ipcs -m -i 229379` (where the 229379 is my shm id).

```
dev@hanzo:~$ ipcs -m -i 229379

Shared memory Segment shmid=229379
uid=1000    gid=1000    cuid=1000    cgid=1000
mode=0    access_perms=0
bytes=4096    lpid=3617    cpid=3430    nattch=0
att_time=Sat Mar 26 17:21:50 2016
det_time=Sat Mar 26 17:21:50 2016
change_time=Sat Mar 26 17:08:59 2016
```

statistics about the shared memory be found using the `shmctl` API.

Let us add the following code to the shmcli.c file.

```
    struct shmid_ds buf;

    ret = shmctl(shmid, IPC_STAT, &buf);
    if (ret < 0) {
        fprintf(stderr, "failed to shmctl\n");
        perror("shmctl");
        return -1;
    }

    printf("size %d\n", buf.shm_segsz);
    printf("attach time %d\n", buf.shm_atime);
    printf("detach time %d\n", buf.shm_dtime);
    printf("change time %d\n", buf.shm_ctime);
    printf("creator pid %d\n", buf.shm_cpid);
    printf("n attach %d\n", buf.shm_nattch);
```

mmap

`mmap` maps the files or device into memory, so that operations can be directly done on the memory. The memory afterwards, can be synced in or out based on its validity. `mmap` creates a new mapping in the virtual memory of the process.

The prototype is as follows.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

If `addr` is `NULL`, the kernel initialises and chooses a memory and returns as the `mmap` return value.

file size is specified in the `length` argument.

the `prot` is is the protection bits for the memory. It is defined as

prot	description
PROT_EXEC	pages may be executable
PROT_READ	pages may be read
PROT_WRITE	pages may be written

PROT_NONE pages may not be accessible

usual `prot` arguments for a file descriptor are `PROT_READ` and `PROT_WRITE`.

The `flags` argument determines whether the updates to the memory are visible to the other processes mapping to the same region. One of the most commonly used flags are `MAP_SHARED` and `MAP_PRIVATE`.

`MAP_SHARED` makes the other processes get the updates on the pages.

`MAP_PRIVATE` creates a private copy on write mapping and the updates are not visible to other processes that are mapping to the same file.

To unmap the memory that is maped by `mmap` the `munmap` is used. The `munmap` unmaps the mapped memory.

The prototype is as follows.

```
int munmap(void *addr, size_t length);
```

include <sys/mman.h> for the `mmap` API. Download [here (https://github.com/DevNaga/gists/blob/master/mmap.c)](https://github.com/DevNaga/gists/blob/master/mmap.c)

sample code:

```c
#include <stdio.h>
#include <errno.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char **argv)
{
    int ret;
    int fd;
    void *addr;
    struct stat s;

    if (argc != 2) {
        printf("%s [filename]\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        printf("failed to open %s\n", argv[1]);
        return -1;
    }

    ret = stat(argv[1], &s);
    if (ret < 0) {
        printf("failed to stat %s\n", argv[1]);
        return -1;
    }

    addr = mmap(NULL, s.st_size, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
    if (!addr) {
        printf("Failed to mmap %s\n", strerror(errno));
        return -1;
    }

    printf("data at the address %p is %s\n", addr, addr);
    munmap(addr, s.st_size);
    close(fd);
    return 0;
}
```

before running the program, perform the following.

```
echo "mmap test" > test
gcc -Wall mmap.c
./a.out test
```

if there is no file that is available or the file is not a text file, the visualisation of the data is not possible.

There is also a way to write the data stored at the memory back to the file using the `msync` API. `msync` allows the memory written at the address to be flushed down to the file either synchronously or asynchronously.

The `msync` API prototype is as follows.

```
int msync(void *addr, size_t length, int flags);
```

The `msync` will write the contents stored at the address `addr` of `length` bytes into the file that the `addr` points to. The `addr` is the return value of the `mmap` where in which the file descriptor is given to map the contents.

The `flags` argument has two values.

`MS_ASYNC`: schedule an update on this address to the file on the disk. The call returns immediately after setting the bit in the kernel for the update.

`MS_SYNC`: request an update and wait till the update finishes.

Here is an extension of the above example that performs the `msync` API. Download [here (https://github.com/DevNaga/gists/blob/master/mmap_sync.c)](https://github.com/DevNaga/gists/blob/master/mmap_sync.c)

```
#include <stdio.h>
#include <errno.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char **argv)
{
    int ret;
    int fd;
    void *addr;
    struct stat s;
    int file_size = 0;

    if (argc != 3) {
        printf("%s [filename] [filesize in MB]\n", argv[0]);
        return -1;
    }

    file_size = atoi(argv[2]);

    fd = open(argv[1], O_RDWR | O_CREAT, S_IRWXU);
    if (fd < 0) {
        printf("failed to open %s\n", argv[1]);
        return -1;
```

```
    }

    ret = ftruncate(fd, file_size * 1024 * 1024);
    if (ret < 0) {
        printf("failed to trucate file %s to %d MB\n", argv[1], file_size);
        return -1;
    }

    ret = stat(argv[1], &s);
    if (ret < 0) {
        printf("failed to stat %s\n", argv[1]);
        return -1;
    }

    addr = mmap(NULL, s.st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (!addr) {
        printf("Failed to mmap %s\n", strerror(errno));
        return -1;
    }

    printf("data at the address %p is %s\n", addr, addr);

    memset(addr, 0, s.st_size);
    strcpy(addr, "Hello Mmap");

    // sync to the disk sychrnously
    msync(addr, s.st_size, MS_SYNC);
    perror("msync");

    munmap(addr, s.st_size);

    close(fd);
    return 0;
}
```

The `ftruncate` is used to first truncate the file before calling mmap with the size of the file. if the file is created newly, its size is default 0 bytes. Thus `mmap` fails on mapping the file to the memory. Instead truncate the file with a specific size and then performing a `stat` on it gives the size of the new truncated value. Thus the call on `mmap` will succeed and the mapping is performed. Subsequent writes on the files are basically copying the values to the address by `strcpy` if string is supposed to be written to the file or a `memcpy` if the data is other than string format.

The `mmap` is mostly used in optimising the file writes, such as in case of data bases. They map the file into the RAM and only write (perform the `msync`) optimally. This reduces the use of write system calls in the kernel and the kernel's paging daemon flushing the pages to the disk and using this saved CPU usage to the other tasks.

Semaphores

utilities

ipcs command

ipcs command can be used to obtain the sysV IPC information.

simply typing `ipcs` command would show us the list of message queues, semaphores and shared memory segments.

an example of this is below:

```
dev@hanzo:~/mbedtls$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 65536      lightdm    600        524288     2          dest
0x00000000 163841     lightdm    600        524288     2          dest
0x00000000 196610     lightdm    600        33554432   2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

- ipcs -m lists the number of shared memory segments in use.
- ipcs -s lists the semaphores.
- ipcs -q lists the message queues.

The help on the ipcs command shows

```
dev@hanzo:~/mbedtls$ ipcs -h

Usage:
 ipcs [resource ...] [output-format]
 ipcs [resource] -i <id>

Options:
 -i, --id <id>  print details on resource identified by <id>
 -h, --help     display this help and exit
 -V, --version  output version information and exit

Resource options:
 -m, --shmems      shared memory segments
 -q, --queues      message queues
 -s, --semaphores  semaphores
 -a, --all         all (default)

Output format:
 -t, --time        show attach, detach and change times
 -p, --pid         show PIDs of creator and last operator
 -c, --creator     show creator and owner
 -l, --limits      show resource limits
 -u, --summary     show status summary
     --human       show sizes in human-readable format
 -b, --bytes       show sizes in bytes
```

The suboption `-i` provides the details of the resource that can be identified by using the shmid, semid or msgqueue id etc.

The command
`ipcs -m -i 65536` gives me the following on my machine.

```
dev@hanzo:~/mbedtls$ ipcs -m -i 65536

Shared memory Segment shmid=65536
uid=112 gid=112 cuid=112    cgid=119
mode=01600  access_perms=0600
bytes=524288    lpid=1030   cpid=983    nattch=2
att_time=Sat Mar 26 12:02:45 2016
det_time=Sat Mar 26 12:02:45 2016
change_time=Sat Mar 26 12:02:42 2016
```

dup and dup2

The dup and dup2 system calls duplicates the file descriptors.

The dup system call creates a copy of the file descriptor.

the dup system call prototype is as follows:

```
int dup(int oldfd);
```

The dup2 system call on the other hand makes a new copy of the old file descriptor. If the new file descriptor is open it closes and performs a dup system call.

The dup2 system call prototype is as follows:

```
int dup2(int olfd, int newfd);
```

dup2 closes the file descriptor `newfd` and dups the `oldfd` into `newfd`. The `newfd` is then returned.

If the `oldfd` and `newfd` are same, then nothing is closed and the `newfd` is returned.

The following example gives a brief description of the dup system call.

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char *msg = "printing \n";
    int fd1, fd2;

    fd1 = dup(1);
    fd2 = dup(fd1);

    printf("%d %d\n", fd1, fd2);

    write(fd1, msg, strlen(msg) + 1);
    return 0;
}
```

fcntl system call

`fcntl` system calls allow to control file descriptor options.

Here are some of the options.

option	description
F_DUPFD F_DUPFD_CLOEXEC	duplicate file descriptor
F_GETFD F_SETFD	file descriptor flags
F_GETFL F_SETFL	file status flags
F_SETLK F_SETLKW	advisory locking

Example in C:

```
    int fd;
    int flags = 0;
    int ret;

    flags = fcntl(fd, F_GETFL, 0);
    if (flags != 0) {
        return -1;
    }

    flags |= O_NONBLOCK;

    ret = fcntl(fd, F_SETFL, flags);
    if (ret != 0) {
        return -1;
    }
```

The F_DUPFD option is similar to the dup system call. This can be done alternatively with `fcntl`.

below example demonstrates the F_DUPFD feature of `fcntl` system call. Download [here](https://github.com/DevNaga/gists/blob/master/fcntl.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

int main(int argc, char **argv)
{
    int stdout_fd;

    if (argc != 2) {
        fprintf(stderr, "<%s> text\n", argv[0]);
        return -1;
    }

    stdout_fd = fcntl(1, F_DUPFD, 3);
    if (stdout_fd < 0) {
        fprintf(stderr, "failed to fcntl dupfd %s\n", strerror(errno));
        return -1;
    }

    write(stdout_fd, argv[1], strlen(argv[1]) + 1);

    return 0;
}
```

lockf system call

`lockf` system call applies or removes a lock on a specific portion of the file. Also called record locking.

The prototype is below,

```
int lockf(int fd, int cmd, off_t len);
```

the `fd` is a file descriptor of the file. The `cmd` has one of the following options.

`F_LOCK`: exclusive lock on the specified section of the file. Other user of the same lock on the file may hang indefnitely. Lock is released when the file descriptor is closed.

`F_TLOCK` : try for lock and fails if lock does not acquire.

`F_ULOCK` : unlock a particular section of the file.

the `len` argument can give the portion of the file the lock needs to be held for. If the `len` is set to 0, then the lock extends from beginning to the end of the file.

Below is an example of the `lockf` system call. Download [here](https://github.com/DevNaga/gists/blob/master/lockf.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

off_t filesize(char *file)
{
    struct stat s;
    int ret;

    ret = stat(file, &s);
    if (ret < 0) {
        return -1;
    }

    return s.st_size;
}

int main(int argc, char **argv)
{
    pid_t pid;
    int fd;
    off_t size;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }
```

```
    size = filesize(argv[1]);

    fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        return -1;
    }

    pid = fork();
    if (pid == 0) {
        ret = lockf(fd, F_LOCK, size);
        if (ret < 0) {
            printf("cannot lock.. %s\n", strerror(errno));
            return -1;
        }

        printf("lock acquired.. now set to sleep\n");
        sleep(4);
        printf("unlock..\n");

        ret = lockf(fd, F_ULOCK, size);
    } else {
        sleep(1);
        ret = lockf(fd, F_LOCK, size);
        if (ret < 0) {
            printf("cannto loock.. %s\n", strerror(errno));
            return -1;
        }

        printf("lock acquired by parent..\n");
        lockf(fd, F_ULOCK, size);
    }

    return 0;
}
```

sometimes, the usual case of the `lockf` system call is to use it as a lock for creating the pid file. Pid files are usually created by the deamons to advertise that they have started.

Below is one example of such class. Download [here](https://github.com/DevNaga/gists/blob/master/pidfile.cpp)

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

class pidFile {
    public:
        pidFile() { }
```

```
        ~pidFile() { }
        int setPidFile(std::string fileName)
        {
            int ret;

            fd_ = open(fileName.c_str(), O_RDWR);
            if (fd_ < 0) {
                return -1;
            }

            ret = lockf(fd_, F_TLOCK, 0);
            if (ret< 0) {
                return -1;
            }

            return 0;
        }

        int removePidFile()
        {
            lockf(fd_, F_ULOCK, 0);

            close(fd_);

            return 0;
        }

    private:
        int fd_;
};

int main(int argc, char **argv)
{
    pidFile p;
    int ret;

    if (argc != 2) {
        std::cerr << argv[0] << " filename" << std::endl;
        return -1;
    }


    ret = p.setPidFile(std::string(argv[1]));
    if (ret < 0) {
        std::cerr << "failed to acquire lock " << std::endl;
        return -1;
    }

    std::cout << "lock acquired successfully" << std::endl;

    // run another process and test locking..
    while (1);
```

```
    return 0;
}
```

chroot

The chroot operation on a system changes the current root directory of the process. It effectively hides the root directory to the process.

There are two manual pages about the chroot... one talks about the chroot command and another talks about the chroot system call. We are here with talking about the chroot system call.

The manual page of the chroot system call has the following prototype.

```
int chroot(const char *path);
```

please be sure to include the header file `<unistd.h>`.

only root users or root privileged programs can call the chroot. Otherwise, EACCESS will be returned due to invalid / non-privileged permissions of the caller.

usually the chroot system call is used by the daemons or remote logging programs when performing logging or opening log files (ex: ftp) . This is usually the case because the remote attacker can open a file under / or /etc/ and overwrite the file contents.

The chroot can be used as a jail by the process to limit its scope of the visibility to the file system and the files around it.

So when chroot has been called and sucessful, the current working directory for the process becomes `/` although it is not running under `/`.

The chroot needs to be combined with the chdir system call. It is done as follows. First execute chdir system call on and then chroot system call.

The following example demonstrates this:

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

#define CHROOT_DIR "/home/dev/"

int main(void)
{
    FILE *fp;
    int ret;
    char filename[100];

    strcpy(filename, "./test");

    fp = fopen(filename, "w");
    if (!fp) {
        fprintf(stderr, "failed to open %s\n", filename);
        return -1;
    }

    fprintf(stderr, "opened %s success\n", filename);

    fclose(fp);

    ret = chdir(CHROOT_DIR);
    if (ret != 0) {
        fprintf(stderr, "failed to chdir to " CHROOT_DIR);
        return -1;
    }

    fprintf(stderr, "chdir success %d\n", ret);

    ret = chroot(CHROOT_DIR);
    if (ret != 0) {
        fprintf(stderr, "failed to chroot into " CHROOT_DIR);
        return -1;
    }

    fprintf(stderr, "chroot success %d\n", ret);

    fp = fopen(filename, "r");
    printf("fp %p\n", fp);
    if (!fp) {
        fprintf(stderr, "failed to open %s\n", filename);
        return -1;
    }

    return 0;
}
```

So before running this program, we change the directory to some new directory. I was running this program in `/mnt/linux_drive/gists/`. My home directory is `/home/dev/`.

The program creates a new file called `test` under `/mnt/linux_drive/gists/` and then changes the directory to `/home/dev/` and chroots into the `/home/dev/` directory. It then tests if the `chroot` is successful by opening the program in the same directory.

useful links on the chroot:

1. https://lwn.net/Articles/252794/

readlink

readlink resolves the symbolic links. The readlink API prototype is as follows.

```
size_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

Also, include the `<unistd.h>` header file.

the `pathname` is the symbolic link and the readlink API dereferences the symbolic link and places the real name into the `buf` argument. The `readlink` API does not terminate the `buf` pointer with a `\0`. Thus we should be doing the `buf[readlink_ret + 1] = '\0'` to make sure the buffer is null terminated.

The below example demonstrates the `readlink`

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buf[100];
    size_t len;

    if (argc != 2) {
        fprintf(stderr, "%s <linkfile name>\n", argv[0]);
        return -1;
    }

    memset(buf, 0, sizeof(buf));

    len = readlink(argv[1], buf, sizeof(buf));
    buf[len + 1] = '\0';

    fprintf(stderr, "resolved %s\n", buf);

    return 0;
}
```

we compile and run the program on to one of the files under /proc. For ex: when run with the `/proc/1/fd/1` the program gives us:

```
resolved /dev/null
```

The real name of the `/proc/1/fd/1` is actually `/dev/null`.

Here is another example of `readdir` that reads a directory and describes the links with in the directory.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    int ret;
    DIR *dirp;
    struct dirent *entry;

    if (argc != 2) {
        fprintf(stderr, "%s <directory>\n", argv[0]);
        return -1;
    }

    dirp = opendir(argv[1]);
    if (!dirp) {
        fprintf(stderr, "failed to open %s\n", argv[1]);
        return -1;
    }

    while (entry = readdir(dirp)) {
        char path[400];
        size_t len;
        char realname[400];
        struct stat s;

        memset(path, 0, sizeof(path));
        strcpy(path, argv[1]);
        strcat(path, "/");
        strcat(path, entry->d_name);

        ret = stat(path, &s);
        if (ret < 0) {
            fprintf(stderr, "failed to stat %s\n", path);
            continue;
        }

        if (S_ISDIR(s.st_mode)) {
            continue;
        }

        printf("filename %s\t", path);

        memset(realname, 0, sizeof(realname));
```

```
        len = readlink(path, realname, sizeof(realname));
        if (len < 0) {
            fprintf(stderr, "failed to readlink\n");
            return -1;
        }

        if (len > sizeof(realname)) {
            fprintf(stderr, "too large realname\n");
            continue;
        }

        realname[len + 1] = '\0';

        printf("realname %s\n", realname);
    }

    closedir(dirp);

    return 0;
}
```

The program opens a directory with the `opendir` system call and reads it using `readdir` till the end of the file is reached. At each entry read, we check if the file is a directory and drop dereferencing it. Otherwise, we reference the link using the `readlink` and print the realname of the link.

The program is careful at avoiding the buffer overflow when the length of the name exceeds the length of the buffer. In such cases we continue to the next name.

symlink

symlink is another system call used to create symlinks of a real file. symlinks are many ways useful to shorthand represent a long path, to represent a generic name for paths with random names etc...

The symlink prototype is as follows.

```
int symlink(const char *target, const char *linkpath);
```

the `target` is the original file and `linkpath` is the link file. It is advised that both of the arguments should be represented with the absolute paths thus avoiding the broken links although the real directory or the file is present.

If the file that the link is pointing to is deleted or moved somewhere else, the link becomes invalid. This link is called as **dangling symlink**.

Another important note is that when a link gets deleted with the `unlink` command, only the link will be removed not the original file that the link is pointing to.

The following example provides the `symlink` API in use:

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

int main(int argc, char **argv)
{
    int ret;

    if (argc != 3) {
        fprintf(stderr, "%s <original file> <symlink>\n", argv[0]);
        return -1;
    }

    ret = symlink(argv[1], argv[2]);
    if (ret != 0) {
        perror("symlink");
        fprintf(stderr, "failed to create symlink %s for file %s\n", argv[2],
argv[1]);
        return -1;
    }

    printf("created symlink %s for %s\n", argv[2], argv[1]);

    return 0;
}
```

the <original file> and <symlink> must be represented with the absolute paths via the command line. such as the below example:

```
./a.out /home/dev/test.c /home/dev/work/test.c

created symlink /home/dev/work/test.c for /home/dev/test.c
```

backtracing

At some situations such as crash, it is at most useful to trace the function calls. One possible way to find this is to perform gdb on the program and typing `bt` when the crash occured. The other possible way to simply print the dump using some of the C APIs such as `backtrace` and `backtrace_symbols`.

The function prototypes are as follows.

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

The `backtrace` function returns a backtrace of the calling program, into the array pointed to by buffer. provide `size` large enough to accomodate all the addresses.

`backtrace` API collects all the fubnction call addresses into the buffer.

The `backtrace_symbols` function translates the addresses into the strings.

The header file <execinfo.h> contains the prototypes for these API.

The example of such is follows. Download [here (https://github.com/DevNaga/gists/blob/master/backtrace.c)](https://github.com/DevNaga/gists/blob/master/backtrace.c)

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

void function2()
{
    void *buf[300];
    char **strings;
    int i, len;

    len = backtrace(buf, 300);
    printf("returns %d\n", len);

    strings = backtrace_symbols(buf, len);
    if (strings) {
        for (i = 0; i < len; i ++) {
            printf("%s\n", strings[i]);
        }
    }
}

void function1()
{
    function2();
}

int main(void)
{
    function1();
    return 0;
}
```

compile the program as follows.

```
[root@localhost manuscript]# gcc -rdynamic backtrace.c  -g
```

Without the `-g` or `-rdynamic` the backtrace that is producted may not contain the needed symbols, and some of the symbols might be lost.

run the program as follows thus producing the following output.

```
[root@localhost manuscript]# ./a.out
returns 5
./a.out(function2+0x1f) [0x4008f5]
./a.out(function1+0xe) [0x40096f]
./a.out(main+0xe) [0x40097f]
/lib64/libc.so.6(__libc_start_main+0xf0) [0x7f04ca774fe0]
./a.out() [0x400809]
[root@localhost manuscript]#
```

Some examples of the `backtrace` use it to produce a crashtrace when the crash occur. For this, the program registers the `SIGSEGV` (segfault signal) via `signal` or `sigprocmask` call. The handler gets called at the event of the crash. The below program provides a case on such scenario.

The program registers the segfault handler with the `signal` and dereferences a null character pointer, thus resulting in a crash. The handler immediately gets called and provides us the trace of the calls made to come to the crash path. The first function in the calls will be the signal handler.

NOTE: When registering a signal handler for the segfault (i.e, `SIGSEGV`) please make sure to abort the program, otherwise the signal handler will be restarted continuously. To test that out, remove the abort function call in the signal handler below.

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

/**
 * @brief - signal handler for SIGSEGV
 */
void segfault_handler(int si)
{
        void *crash_trace[100];
        char **stack_trace;
        size_t size;
        int i;

        size = backtrace(crash_trace, 100);

        if (size <= 0) {
                fprintf(stderr, "No symbols found \n");
                goto end;
        }

        stack_trace = backtrace_symbols(crash_trace, size);
        if (!stack_trace) {
                fprintf(stderr, "No symbols found \n");
                goto end;
        }

        fprintf(stderr, "Trace\n");
        fprintf(stderr, "----------------XXXXXX-------------\n");
        for (i = 0; i < size; i ++) {
                printf("[%s]\n", stack_trace[i]);
        }
        fprintf(stderr, "----------------XXXXXX-------------\n");
end:
        abort();
}
```

```
void function3()
{
        int *data = NULL;

        printf("Data %s\n", *data);
}

void function2()
{
        function3();
}

void function1()
{
        function2();
}

int main()
{
        signal(SIGSEGV, segfault_handler);
        function1();
}
```

Sometimes, it is necessary to dump the trace to a file by the program. This occurs when the program is running as a daemon or running in the background. The glibc provides us another function called `backtrace_symbols_fd`. This can also be useful when sending the trace over to a network socket or to a local pipe to monitor the crash and perform necessary action such as recording.

The `backtrace_symbols_fd` prints the trace into a file. Here is an example:

```
#include <stdio.h>
#include <execinfo.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int fd;

void segfault_handler(int si)
{
        void *crash_trace[100];
        size_t size;

        size = backtrace(crash_trace, 100);

        if (size <= 0) {
```

```
                fprintf(stderr, "No symbols found \n");
                goto end;
        }

        backtrace_symbols_fd(crash_trace, size, fd);
end:
        abort();
}

void function3()
{
        int *data = NULL;

        printf("Data %s\n", *data);
}

void function2()
{
        function3();
}

void function1()
{
        function2();
}

int main(int argc, char **argv)
{
        if (argc != 2) {
                fprintf(stderr, "%s <trace file>\n", argv[0]);
                return -1;
        }

        fd = open(argv[1], O_CREAT | O_RDWR, S_IRWXU);
        if (fd < 0) {
                fprintf(stderr, "failed to open %s\n", argv[1]);
                return -1;
        }

        signal(SIGSEGV, segfault_handler);
        function1();
}
```

Core dump

Sometimes, the programs crash. At some points of time, it is hard to debug what's wrong with the program. The Linux OS supports a coredump feature to overcome this situation. When the program terminates in linux, the current state of the program along with every other opened

file, state information and registers etc will be dumped into the file called `core`. The core file contains an image of the process's memory at the time of termination. The coredump file can be used in gdb to debug the problem.

Let's see below how we can enable the coredump settings one by one.

1. The **CONFIG_COREDUMP** kernel configuration parameter need to be setup while compiling the kernel.
2. once kernel is built with **CONFIG_COREDUMP**, the system is restarted into the built kernel.
3. The coredump file name is to be configured in **/proc/sys/kernel/core_pattern**.

The below table describe a format of the coredump that could be configured.

format	description
%%	a single % character
%c	core file size soft resource limit of crashing process
%d	dump mode
%e	executable filename
%E	pathname of the executable
%g	real GID of the dumped process
%h	hostname
%i	TID of the thread that triggered the core-dump
%p	PID of the dumped process
%s	number of signal causing dump
%t	time of dump expressed since the epoch

In the above table, if you see, the most important ones are `hostname`, `executable filename`, `number of signals` and the `time of dump`.

1. The `hostname` would tell us on which computer the crash was on, this is very useful if the crash is on a large network of systems running virtualised machines etc.. (such as docker).
2. The `executable filename` will pin-point to the process or the software that we should debug
3. the `number of signals` will tell us that if this is a cause of asnychrnous signals / unknown signal triggers
4. the `time of dump` would give us a chance in time that the crash occured at this point in time and that we could compare this time with the log or usually a syslog to see what log messages are spewed in it at the time of this crash.

The coredump can be configured with `sysctl` as well.. as root

```
sysctl -w kernel.core_pattern=/tmp/core_%h_%e_%s_%t
```

this will create a corefile under /tmp/ with name `core_${host}_${exename}_${number_of_signal}_${timeofdump}`.

Corefiles are usually huge, and they should be in a mount point where there is sufficient memory (they are big of order of 40 MB on an embedded system). If the mount point, does not contain sufficient memory, the directory of the corefiles should be managed and used or older files must be deleted in order to capture new coredump.

Corefiles are usually analysed by using the GCC. The following is the list of steps that are used to analyze the core file to a program that links to the shared libraries.

```
gdb <binary_name>

# in gdb
set solib-searchpath path/to/shared_libraries

# in gdb
core-file <core_file_name>

# in gdb
bt

# in gdb
bt full # dumps the full contents of the stack and the history of the program
at this point
```

A brief description on the `set solib-searchpath path/to/shared_libraries` command in the gdb:

1. The **path/to/shared_libraries** are the target shared libraries.
2. in case if your compiler is native and running on a native system, then you can simply point that to /usr/lib/ or /lib/.
3. in case if your compiler is a cross compiler such as for target ARM or MIPS .. you should point the path to the toolchain of those cross compiled libs that the program is linked against.

The above program dumps the trace to the crash and points to the line number of a c program. The program must be compiled with **-g** option while using the `gcc` to get a proper trace value.

usually `bt` is prefered over `bt full` because `bt` generally gives an idea without going in detail about any problem.
Cases, where a full debug and history are required, `bt full` is generally used.

popen and pclose

`popen` and `pclose` are the useful API that are another form of executing a shell or a command with in the process.

The prototypes are as follows.

```
FILE *popen(const char *command, const char *mode);
```

```
int pclose(FILE *stream);
```

The popen call executes the shell command as a sub process. However, instead of waiting for the completion of the command, it returns a file pointer.

The file pointer can then be used to read or write to the stream. Meaning, a read gets the output of the command and a write sends the data to the sub process.

In case of failure, the file pointer returned will be `NULL`.

The `pclose` will then close the file pointer.

The following code sample shows the popen and `pclose`. You can download it in here (https://github.com/DevNaga/gists/blob/master/popen_pclose.c)

```
#include <stdio.h>

int main(void)
{
    char data[64];
    FILE *fp;

    fp = popen("ls -l", "r");
    if (!fp) {
        fprintf(stderr, "failed to open ls -l \n");
        return -1;
    }

    while (fgets(data, sizeof(data), fp)) {
        fprintf(stderr, "%s", data);
    }

    pclose(fp);

    return 0;
}
```

prctl

posix threads programming

Thread is light weight process. Here are some of the features of threads.

- Has the same address as the main thread (the thread that has created it).
- Do not have to assign any stack. It will be allocated automatically.
- Main thread can either wait or can create threads that will be wait by the OS. The OS will directly perform the cleanup
- Thread immediately starts executing after it has been created. The execution depends on the scheduler. The thread will exit as soon as the main thread exits.
- Threads can be created and stopped at run time.
- Creation might involve some overhead. So the threads can be created at the initial stage, and can be woken up in-between or when needed. These are also called worker threads.
- The `mutex`, `conditional variable` are some of the synchronisation functions to avoid corruption and parallel access to the same address space.
- Standard GNU library provides POSIX thread implementation library. In linux there is no big difference between threads and processes as the overhead at creation and maintenance is mostly the same.
- The threads will not appear in the `ps` command output. but `ps -eLf` would give all threads and processes running in the system.

In the below sections, the topic of discussion is on posix threads, specially on pthreads programming and handling.

For compilation , include `pthread.h` and pass `-pthread` to the linker flags.

creating threads

similar to processes, threads have IDs. Thread have the id of the form `pthread_t`. Threads are created using the `pthread_create` API... Its prototype is desscribed below,

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*thread_func)
(void *), void *data);
```

the above API accepts a thread id of the form `pthread_t` and attribute of the form `pthread_attr_t` and a thread function that is executed when the thread is created as well the data that thread function going to receive.

usually, most implementations call the `pthread_create` the following way,

```
int ret;
pthread_t tid;
int data;

ret = pthread_create(&tid, NULL, thread_func, &data);
if (ret < 0) {
    fprintf(stderr, "failed to create thread\n");
    return -1;
}
```

Thread creation might fail, because maximum number of processes on the system is exceeded the limit set. (see `sysconf` for max processes allowed)

the thread created will be automatically started up and starts running the thread function. So care must be taken to protect the common memory areas the thread and the main thread or others are accessing. This means the use of locks. pthread library provides mutexes and conditional variables for locking and job scheduling or waking up of threads. More about the locks and condition variables in below sections.

Below code explains a bit more about the pthread creation and running.

Download [here (https://github.com/DevNaga/gists/blob/master/pthread_create.c)](https://github.com/DevNaga/gists/blob/master/pthread_create.c)

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *thread_func(void *thread_ctx)
{
    while (1) {
        sleep(1);
        printf("thread func..\n");
    }
}

int main()
{
    int ret;
    pthread_t tid;

    ret = pthread_create(&tid, NULL, thread_func, NULL);
    if (ret < 0) {
        fprintf(stderr, "failed to create thread\n");
        return -1;
    }

    pthread_join(tid, NULL);

    return 0;
}
```

above example creates a thread called `thread_func` and starts executing it. The `pthread_join` is called to wait for the thread to complete its execution. More about the `pthread_join` in below sections. The created thread starts running the function and the function has an infinite loop and at every second it wakes up and prints a text "thread func..." on the screen.

Most threads that are created are not meant to die for short periods, but are used for executing a set of work items that main thread cannot handle. So main thread provides the work to the threads and they execute the work for the main thread and notify the main thread of the work completion, so that the main thread can push another job for the thread to execute.

Linux threads have the same overhead in creation as that of the linux processes.

joining threads and thread attributes

Threads are created by default in attached mode, meaning they are joinable and must be joined by using `pthread_join`.

`pthread_join` prototype is as below.

```c
int pthread_join(pthread_t tid, void **retval)
```

the `pthread_join` accepts the thread id and then the `retval` is the value output when the thread returns and stops execution. This is catched in the `retval` argument of the `pthread_join` call.

Threads must be joined when they are not created in detach state. The main thread has to wait for the threads to complete their execution. The default thread created is joinable. This can be found via the `pthread_attr_getdetachstate` API.

The prototype is as follows,

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detach_state);
```

The thread can be created in detach state with the use of `pthread_attr_setdetachstate`.

The prototype is as follows.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detach_state)
```

where the `detach_state` is one of `PTHREAD_CREATE_DETACHED`, `PTHREAD_CREATE_JOINABLE`.

Before calling the APIs, one must initialise the thread attributes before creating a thread by using `pthread_create`.

This can be done using the `pthread_attr_init` API.

The prototype is as follows.

```
int pthread_attr_init(pthread_attr_t *attr);
```

Below is one of the example of the `pthread_attr_setdetachstate` and `pthread_attr_getdetachstate` functions.. Download here (https://github.com/DevNaga/gists/blob/master/pthread_det.cpp)

```
#include <iostream>
#include <chrono>
#include <unistd.h>
#include <pthread.h>

void *thread_func(void *thread_contx)
{
    while (1) {
        sleep(1);
        std::cout <<" thread " << pthread_self() << std::endl;
    }
}

int main()
{
    pthread_attr_t attr;
    pthread_t thread_id;
    int ret;

    ret = pthread_attr_init(&attr);
    if (ret < 0) {
        return -1;
    }

    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

```
    if (ret < 0) {
        return -1;
    }

    ret = pthread_create(&thread_id, &attr, thread_func, &thread_id);
    if (ret < 0) {
        return -1;
    }

    int detach_state = 0;

    ret = pthread_attr_getdetachstate(&attr, &detach_state);
    if (ret < 0) {
        return -1;
    }

    if (detach_state == PTHREAD_CREATE_DETACHED) {
        std::cout  << "thread created as detached..\n" ;
    } else if (detach_state == PTHREAD_CREATE_JOINABLE) {
        std::cout << "thread created as joniable\n";
    }

    if (detach_state == PTHREAD_CREATE_JOINABLE) {
        pthread_join(thread_id, NULL);
    } else {
        while (1) {
            sleep(1);
        }
    }

    return 0;
}
```

The `pthread_attr_init` is called before creating any thread with the `pthread_create`. This is further called with the `pthread_attr_setdetachstate` with thread in detachmode, aka using the `PTHREAD_CREATE_DETACHED`. since the thread is in detached state, it cannot be joined with `pthread_join`. Once the thread is created, the attribute is again checked using the `pthread_attr_getdetachstate` and the detach state is checked if its in `JOINABLE` condition, a `pthread_join` is then called upon, otherwise, the program spins in a sleep loop forever.

Try replacing the `PTHREAD_CREATE_DETACHED` with `PTHREAD_CREATE_JOINABLE` in the call to `pthread_attr_setdetachstate` and see what happens to the joinable section of the code below.

locking

mutexes

mutexes are varibles that is used for locking more than one thread of execution. Pthreads provide the mutexes for locking purposes.

A mutex needs to be initialised before it is being used in the program. pthread library defines the mutex as `pthread_mutex_t`.

To declare a mutex variable,

```
pthread_mutex_t mutex;
```

mutex needs to be initialised before it is used to lock any particular section of the data. The `pthread_mutex_init` is used to initialise the mutex. Its prototype is as follows,

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr_t *attr);
```

usually, the following code is generally used to initilaise a mutex.

```
pthread_mutex_t mutex;
int ret;

ret = pthread_mutex_init(&mutex, NULL);
if (ret < 0) {
    return -1;
}
```

Once the mutex is initialised, it can then be used to lock and unlock portion of the program. The `pthread_mutex_lock` and `pthread_mutex_unlock` are used to lock and unlock particular section of the program respectively.

The `pthread_mutex_lock` prototype is as follows.

```
int pthread_mutex_lock(pthread_mutex_t *lock);
```

The `pthread_mutex_unlock` prototype is as follows.

```
int pthread_mutex_unlock(pthread_mutex_t *lock);
```

Below example provide a locking demo. Download [here](https://github.com/DevNaga/gists/blob/master/pthread_mutex.c) (https://github.com/DevNaga/gists/blob/master/pthread_mutex.c)

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t lock;

void *thread_f(void *d)
{
    int *ptr = d;

    while (1) {
        sleep(1);

        pthread_mutex_lock(&lock);

        (*ptr) ++;
```

```
        pthread_mutex_unlock(&lock);
    }
}


int main()
{
    int t = 4;
    pthread_t tid;
    pthread_attr_t attr;
    int ret;

    ret = pthread_attr_init(&attr);
    if (ret < 0) {
        return -1;
    }

    ret = pthread_mutex_init(&lock, NULL);
    if (ret < 0) {
        return -1;
    }

    ret = pthread_create(&tid, &attr, thread_f, &t);
    if (ret < 0) {
        return -1;
    }

    while (1) {

        pthread_mutex_lock(&lock);

        printf("t value %d\n", t);

        pthread_mutex_unlock(&lock);
        sleep(1);
    }

    return 0;
};
```

condition variables

condition variables are synchronisation primitives that wait till a particular condition occurs. condition variables are mostly used in threads to wait for a particular event and sleep. No load occurs when the condition variable sleeps. So that the other thread can signal the condition variable to wake up the thread that is waiting on this condition variable. The pthread provides some of the below condition varilables.

1. `pthread_cond_init`.
2. `pthread_cond_wait`.
3. `pthread_cond_signal`.

4. `pthread_cond_broadcast`.
5. `pthread_cond_destroy`.

the `pthread_cond_t` is used to declare a condition variable. It is declared as the following,

```
pthread_cond_t cond;
```

the `pthread_cond_init` initialises the condition variable.

thread pools

Creating threads at runtime is a bit costly job and a create-delete sequence is an overhead if done frequent.

In general, the threads are created as a pool of workers waiting for the work to be executed.

Below is an example of the basic thread pooling. Download here (https://github.com/DevNaga/gists/blob/master/pools.c)

```c
#include <stdio.h>
#include <pthread.h>

void *thread_func(void *data)
{
    int *i = data;

    while (1) {
        printf("----------------------------\n");
        sleep(1);
        printf("tid-> [%lu] i %d\n", pthread_self(), *i);

        (*i) ++;
        printf("----------------------------\n");
    }

}

int main()
{
    int array[8];
    int i;
    pthread_t tid[8];
    int ret;

    for (i = 0; i < 8; i ++) {
        array[i] = 0;
        ret = pthread_create(&tid[i], NULL, thread_func, &array[i]);
        if (ret < 0) {
            printf("failed to create thread\n");
            return -1;
        }
    }

    for (i = 0; i < 8; i ++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}
```

more detailed thread pool mechanism is shown in the example below. Download [here (https://github.com/DevNaga/gists/blob/master/pool_lib.c)](https://github.com/DevNaga/gists/blob/master/pool_lib.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

struct pthread_object {
    pthread_t tid;
```

```
    int work_count;
    void *thread_data;
    void (*work)(void *thread_data);
    pthread_mutex_t lock;
    pthread_cond_t cond;
};

struct pthread_pool_ctx {
    int n_threads;
    struct pthread_object *n_obj;
};

void *thread_worker(void *thread_data)
{
    struct pthread_object *ctx = thread_data;

    while (1) {
        pthread_mutex_lock(&ctx->lock);

        pthread_cond_wait(&ctx->cond, &ctx->lock);

        ctx->work(ctx->thread_data);

        //ctx->work_count --;

        pthread_mutex_unlock(&ctx->lock);
    }
}

void * pthread_pool_create(int n_workers)
{
    struct pthread_pool_ctx *ctx;
    int i;
    int ret;

    ctx = calloc(1, sizeof(struct pthread_pool_ctx));
    if (!ctx) {
        return NULL;
    }

    ctx->n_threads = n_workers;

    ctx->n_obj = calloc(n_workers, sizeof(struct pthread_object));
    if (!ctx->n_obj) {
        return NULL;
    }


    for (i = 0; i < n_workers; i ++) {
        ctx->n_obj[i].work_count = 0;
        ret = pthread_create(&ctx->n_obj[i].tid, NULL, thread_worker, &ctx-
>n_obj[i]);
```

```
        if (ret < 0) {
            return NULL;
        }
    }

    return ctx;
}

void pthread_schedule_work(void *priv, void (*work)(void *thread_data), void
*work_data)
{
    struct pthread_pool_ctx *ctx = priv;
    static int min = -1;
    int idx = 0;
    int i;

    min = ctx->n_obj[0].work_count;
    for (i = 1; i < ctx->n_threads; i ++) {
        if (ctx->n_obj[i].work_count < min) {
            min = ctx->n_obj[i].work_count;
            idx = i;
        }
    }

    if (idx == -1) {
        return;
    }

    pthread_mutex_lock(&ctx->n_obj[idx].lock);

    ctx->n_obj[idx].work = work;
    ctx->n_obj[idx].thread_data = work_data;
    ctx->n_obj[idx].work_count ++;

    printf("out worker loads  ==== ");

    for (i = 0; i < ctx->n_threads; i ++) {
        printf("| %d ", ctx->n_obj[i].work_count);
    }
    printf(" |\n");

    pthread_cond_signal(&ctx->n_obj[idx].cond);

    pthread_mutex_unlock(&ctx->n_obj[idx].lock);
}

void work_func(void *priv)
{
    int *i = priv;

    printf("value at i %d\n", *i);
```

```
    (*i) ++;

}

int main()
{
    void *priv;

    priv = pthread_pool_create(8);
    if (!priv) {
        return -1;
    }

    int work_data = 0;

    while (1) {
        usleep(200 * 1000);

        pthread_schedule_work(priv, work_func, &work_data);
    }


    return 0;
}
```

The above example defines 2 API for the thread pool.

```
1. pthread_pool_create
2. pthread_schedule_work
```

`pthread_pool_create` creates a pool of threads and assigns the work objects for each of the threads and the corresponding locking. The objects are of type `pthread_object`.

Each thread that has started executes the same worker function, the worker function must be async safe and re-entrant to be able to execute concurrently more than one thread. Each thread waits on the condition variable to see if there is any work pending for the thread.

Main thread signals the threads upon a work is available, and the threads wake up running the job.

The wake up is done via `pthread_schedule_work`. This function checks weights for each thread, considering what amount of work the thread been doing and if its complete, using the `work_count` variable. The function then runs a round robin scheduler, selecting the next thread that has less work than all the threads. The `work_count` is incremented on this thread and the job is given.

Worker loads are continuously printed on the screen when the `pthread_schedule_work` is called.

the main program, initialises the pool by calling `pthread_pool_create` and then schedules the work with `pthread_schedule_work` as and when needed. In the demonstration above, the main thread simply calls the `pthread_schedule_work` at every 200 msec.

The tcp server and client programs under socket section can use threads to exploit the multithreaded client server communication. Below is an example that describe the multi thread server. Download here ()

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <errno.h>

struct thread_data {
    int sock;
};

void *thread_handle(void *data)
{
    struct thread_data *thread_data = data;
    int ret;

    while (1) {
        char rxbuf[1024];

        ret = recv(thread_data->sock, rxbuf, sizeof(rxbuf), 0);
        if (ret <= 0) {
            fprintf(stderr, "failed to recv %s\n", strerror(errno));
            break;
        }

        printf("data %s from client %d\n", (char *)rxbuf, thread_data->sock);
    }
}

int main(int argc, char **argv)
{
    struct sockaddr_in serv;
    int sock;
    int csock;
    struct thread_data data;
    pthread_t tid;
    int ret;

    if (argc != 3) {
        fprintf(stderr, "<%s> <ip> <port>\n", argv[0]);
        return -1;
    }
```

```
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        fprintf(stderr, "failed to socket open %s\n", strerror(errno));
        return -1;
    }

    memset(&serv, 0, sizeof(serv));

    serv.sin_addr.s_addr = inet_addr(argv[1]);
    serv.sin_port = htons(atoi(argv[2]));
    serv.sin_family = AF_INET;

    ret = bind(sock, (struct sockaddr *)&serv, sizeof(serv));
    if (ret < 0) {
        fprintf(stderr, "failed to bind %s\n", strerror(errno));
        return -1;
    }

    ret = listen(sock, 4);
    if (ret < 0) {
        fprintf(stderr, "failed to listen %s\n", strerror(errno));
        return -1;
    }

    while (1) {
        csock = accept(sock, NULL, NULL);
        if (csock < 0) {
        fprintf(stderr, "failed to accept %s\n", strerror(errno));
            return -1;
        }

        struct thread_data *thr;

        thr = calloc(1, sizeof(struct thread_data));
        if (!thr) {
            fprintf(stderr, "failed to allocate %s\n", strerror(errno));
            return -1;
        }

        thr->sock = csock;

        ret = pthread_create(&tid, NULL, thread_handle, thr);
        if (ret < 0) {
            fprintf(stderr, "failed to pthrad_create %s\n", strerror(errno));
            return -1;
        }
    }

    return 0;
}
```

In the above program a server socket is created and waiting for the connections, while it waits for the connections in the infinite while loop, the connection may arrive if the client connects to the server at the given port and ip address.

As and when the connection is arrived, the server creates a thread context structure and sets the socket address to the client socket.

This is then provided as a data pointer to the `pthread_create` call. Once the thread function `thread_handle` is called, it is then waits on client descriptor forever and reads from the clients.

In the above program, the main thread acts as a controller and waiting for the client connections and initiates a thread as soon as it sees a new client. The started thread then serves the connection by waiting on the `recv` system call.

Another example below is the client. Download here ()

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int cli_sock;
    int ret;

    if (argc != 3) {
        fprintf(stderr, "<%s> <ip> <port>\n", argv[0]);
        return -1;
    }

    struct sockaddr_in serv_addr = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = inet_addr(argv[1]),
        .sin_port = htons(atoi(argv[2])),
    };

    cli_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (cli_sock < 0) {
        return -1;
    }

    ret = connect(cli_sock, (struct sockaddr *)&serv_addr, sizeof(struct
sockaddr_in));
    if (ret < 0) {
```

```
        return -1;
    }

    while (1) {
        char msg[] = "sending data to the server";

        send(cli_sock, msg, strlen(msg) + 1, 0);

        sleep(1);
    }

    close(cli_sock);

    return 0;
}
```

Notes:

1. **C++ language does provide some abstraction in the STL via the posix or based on the native thread API employed by the operating system used. It is `std::thread`.**

/proc file system

getting process name from the pid is achieved by using the `/proc/<process-id>/status` file. This file is present for each process.

Below is an example of getting the process name from the process id. Download [here (https://github.com/DevNaga/gists/blob/master/pname_from_pid.c)](https://github.com/DevNaga/gists/blob/master/pname_from_pid.c)

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char name[200];
    char buf[2048];
    char val[100];
    FILE *fp;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> pid\n", argv[0]);
        return -1;
    }

    snprintf(name, sizeof(name), "/proc/%s/status", argv[1]);

    fp = fopen(name, "r");
    if (!fp) {
        return -1;
    }

    while (fgets(buf, sizeof(buf), fp)) {
        int i = 0;
        int j = 0;

        if (strstr(buf, "Name")) {
            i = strlen("Name:");

            while ((buf[i] != '\0') && (buf[i] == ' ')) {
                i ++;
            }

            while (buf[i] != '\n') {
                val[j] = buf[i];
                j ++;
                i ++;
            };

            val[j] = '\0';

        }
    }

    printf("name: %s\n", val);

    return 0;
}
```

valgrind

1. valgrind is a memory checker. It detects various memory errors and problems such as access errors, leaks, un-freeable memory etc.
2. valgrind can be installed in the following way:

   on Ubuntu:

   ```
   sudo apt-get install valgrind
   ```

   on Fedora:

   ```
   #  dnf install valgrind
   ```

3. valgrind simple example:

   ```
   valgrind -v --leak-check=full --leak-resolution=high ./leak_program
   ```

   always keep --leak-resolution to "high" when doing the leak check on the program.

4. describe the possible kinds of leaks:

   ```
   valgrind -v --leak-check=full --leak-resolution=high --show-leak-
   kinds=all ./leak-program
   ```

   set always to `all` for the `--show-leak-kinds` knob to display all possible kinds of leaks.

5. stack trace of an undefined value error:

   ```
   valgrind -v --leak-check=full --leak-resolution=high --show-leak-
   kinds=all --track-origins=yes ./leak_program
   ```

The below sample program describe a simple memory leak.

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

int main(void)
{
    int *var;

    var = malloc(sizeof(int));
    *var = 2;

    printf("%d\n", *var);
    return 0;
}
```

compile the program with `gcc -g` option.

running just `valgrind -v ./a.out` produces the following output.

```
devnaga@devnaga-VirtualBox:~/personal$ valgrind -v ./a.out
```

```
==4259== Memcheck, a memory error detector
==4259== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4259== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4259== Command: ./a.out
==4259==
--4259-- Valgrind options:
--4259--    -v
--4259-- Contents of /proc/version:
--4259--   Linux version 4.4.0-31-generic (buildd@lgw01-16) (gcc version 5.3.1
20160413 (Ubuntu 5.3.1-14ubuntu2.1) ) #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC
2016
--4259--
--4259-- Arch and hwcaps: AMD64, LittleEndian, amd64-cx16-lzcnt-rdtscp-sse3-avx
--4259-- Page sizes: currently 4096, max supported 4096
--4259-- Valgrind library directory: /usr/lib/valgrind
--4259-- Reading syms from /home/devnaga/personal/a.out
--4259-- Reading syms from /lib/x86_64-linux-gnu/ld-2.23.so
--4259--   Considering /lib/x86_64-linux-gnu/ld-2.23.so ..
--4259--   .. CRC mismatch (computed 30b9eb7c wanted d576ac3f)
--4259--   Considering /usr/lib/debug/lib/x86_64-linux-gnu/ld-2.23.so ..
--4259--   .. CRC is valid
--4259-- Reading syms from /usr/lib/valgrind/memcheck-amd64-linux
--4259--   Considering /usr/lib/valgrind/memcheck-amd64-linux ..
--4259--   .. CRC mismatch (computed 5529a2c7 wanted 5bd23904)
--4259--    object doesn't have a symbol table
--4259--    object doesn't have a dynamic symbol table
--4259-- Scheduler: using generic scheduler lock implementation.
--4259-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4259== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4259-by-
devnaga-on-???
==4259== embedded gdbserver: writing to   /tmp/vgdb-pipe-to-vgdb-from-4259-by-
devnaga-on-???
==4259== embedded gdbserver: shared mem   /tmp/vgdb-pipe-shared-mem-vgdb-4259-
by-devnaga-on-???
==4259==
==4259== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4259== don't want to do, unless you know exactly what you're doing,
==4259== or are doing some strange experiment):
==4259==   /usr/lib/valgrind/../../bin/vgdb --pid=4259 ...command...
==4259==
==4259== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4259==   /path/to/gdb ./a.out
==4259== and then give GDB the following command
==4259==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4259
==4259== --pid is optional if only one valgrind process is running
==4259==
--4259-- REDIR: 0x401cdc0 (ld-linux-x86-64.so.2:strlen) redirected to
0x3809e181 (???)
--4259-- Reading syms from /usr/lib/valgrind/vgpreload_core-amd64-linux.so
--4259--   Considering /usr/lib/valgrind/vgpreload_core-amd64-linux.so ..
--4259--   .. CRC mismatch (computed a30c8eaa wanted 7ae2fed4)
--4259--    object doesn't have a symbol table
```

```
--4259-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
--4259--   Considering /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so ..
--4259--   .. CRC mismatch (computed 402c2ab5 wanted 745f25ae)
--4259--    object doesn't have a symbol table
==4259== WARNING: new redirection conflicts with existing -- ignoring it
--4259--     old: 0x0401cdc0 (strlen              ) R-> (0000.0) 0x3809e181 ???
--4259--     new: 0x0401cdc0 (strlen              ) R-> (2007.0) 0x04c31020
strlen
--4259-- REDIR: 0x401b710 (ld-linux-x86-64.so.2:index) redirected to 0x4c30bc0
(index)
--4259-- REDIR: 0x401b930 (ld-linux-x86-64.so.2:strcmp) redirected to 0x4c320d0
(strcmp)
--4259-- REDIR: 0x401db20 (ld-linux-x86-64.so.2:mempcpy) redirected to
0x4c35270 (mempcpy)
--4259-- Reading syms from /lib/x86_64-linux-gnu/libc-2.23.so
--4259--   Considering /lib/x86_64-linux-gnu/libc-2.23.so ..
--4259--   .. CRC mismatch (computed 4e01d81e wanted 7d461875)
--4259--   Considering /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.23.so ..
--4259--   .. CRC is valid
--4259-- REDIR: 0x4ec8e50 (libc.so.6:strcasecmp) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4259-- REDIR: 0x4ec46d0 (libc.so.6:strcspn) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4259-- REDIR: 0x4ecb140 (libc.so.6:strncasecmp) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4259-- REDIR: 0x4ec6b40 (libc.so.6:strpbrk) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4259-- REDIR: 0x4ec6ed0 (libc.so.6:strspn) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4259-- REDIR: 0x4ec859b (libc.so.6:memcpy@GLIBC_2.2.5) redirected to
0x4a286f0 (_vgnU_ifunc_wrapper)
--4259-- REDIR: 0x4ec6850 (libc.so.6:rindex) redirected to 0x4c308a0 (rindex)
--4259-- REDIR: 0x4ebd580 (libc.so.6:malloc) redirected to 0x4c2db20 (malloc)
--4259-- REDIR: 0x4ecfbb0 (libc.so.6:strchrnul) redirected to 0x4c34da0
(strchrnul)
--4259-- REDIR: 0x4ec8800 (libc.so.6:__GI_mempcpy) redirected to 0x4c34fa0
(__GI_mempcpy)
2
--4259-- REDIR: 0x4ebd940 (libc.so.6:free) redirected to 0x4c2ed80 (free)
==4259==
==4259== HEAP SUMMARY:
==4259==     in use at exit: 4 bytes in 1 blocks
==4259==   total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==4259==
==4259== Searching for pointers to 1 not-freed blocks
==4259== Checked 64,544 bytes
==4259==
==4259== LEAK SUMMARY:
==4259==    definitely lost: 4 bytes in 1 blocks
==4259==    indirectly lost: 0 bytes in 0 blocks
==4259==      possibly lost: 0 bytes in 0 blocks
==4259==    still reachable: 0 bytes in 0 blocks
```

```
==4259==         suppressed: 0 bytes in 0 blocks
==4259== Rerun with --leak-check=full to see details of leaked memory
==4259==
==4259== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4259== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

running the program again with `valgrind -v --leak-check=full --leak-resolution=high --track-origins=yes ./a.out` produces the following output.

```
devnaga@devnaga-VirtualBox:~/personal$ valgrind -v --leak-check=full --leak-
resolution=high --track-origins=yes ./a.out

==4274== Memcheck, a memory error detector
==4274== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4274== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4274== Command: ./a.out
==4274==
--4274-- Valgrind options:
--4274--    -v
--4274--    --leak-check=full
--4274--    --leak-resolution=high
--4274--    --track-origins=yes
--4274-- Contents of /proc/version:
--4274--   Linux version 4.4.0-31-generic (buildd@lgw01-16) (gcc version 5.3.1
20160413 (Ubuntu 5.3.1-14ubuntu2.1) ) #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC
2016
--4274--
--4274-- Arch and hwcaps: AMD64, LittleEndian, amd64-cx16-lzcnt-rdtscp-sse3-avx
--4274-- Page sizes: currently 4096, max supported 4096
--4274-- Valgrind library directory: /usr/lib/valgrind
--4274-- Reading syms from /home/devnaga/personal/a.out
--4274-- Reading syms from /lib/x86_64-linux-gnu/ld-2.23.so
--4274--   Considering /lib/x86_64-linux-gnu/ld-2.23.so ..
--4274--   .. CRC mismatch (computed 30b9eb7c wanted d576ac3f)
--4274--   Considering /usr/lib/debug/lib/x86_64-linux-gnu/ld-2.23.so ..
--4274--   .. CRC is valid
--4274-- Reading syms from /usr/lib/valgrind/memcheck-amd64-linux
--4274--   Considering /usr/lib/valgrind/memcheck-amd64-linux ..
--4274--   .. CRC mismatch (computed 5529a2c7 wanted 5bd23904)
--4274--    object doesn't have a symbol table
--4274--    object doesn't have a dynamic symbol table
--4274-- Scheduler: using generic scheduler lock implementation.
--4274-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4274== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4274-by-
devnaga-on-???
==4274== embedded gdbserver: writing to   /tmp/vgdb-pipe-to-vgdb-from-4274-by-
devnaga-on-???
==4274== embedded gdbserver: shared mem   /tmp/vgdb-pipe-shared-mem-vgdb-4274-
by-devnaga-on-???
==4274==
==4274== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4274== don't want to do, unless you know exactly what you're doing,
```

```
==4274== or are doing some strange experiment):
==4274==   /usr/lib/valgrind/../../bin/vgdb --pid=4274 ...command...
==4274==
==4274== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4274==   /path/to/gdb ./a.out
==4274== and then give GDB the following command
==4274==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4274
==4274== --pid is optional if only one valgrind process is running
==4274==
--4274-- REDIR: 0x401cdc0 (ld-linux-x86-64.so.2:strlen) redirected to
0x3809e181 (???)
--4274-- Reading syms from /usr/lib/valgrind/vgpreload_core-amd64-linux.so
--4274--   Considering /usr/lib/valgrind/vgpreload_core-amd64-linux.so ..
--4274--   .. CRC mismatch (computed a30c8eaa wanted 7ae2fed4)
--4274--    object doesn't have a symbol table
--4274-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
--4274--   Considering /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so ..
--4274--   .. CRC mismatch (computed 402c2ab5 wanted 745f25ae)
--4274--    object doesn't have a symbol table
==4274== WARNING: new redirection conflicts with existing -- ignoring it
--4274--     old: 0x0401cdc0 (strlen              ) R-> (0000.0) 0x3809e181 ???
--4274--     new: 0x0401cdc0 (strlen              ) R-> (2007.0) 0x04c31020
strlen
--4274-- REDIR: 0x401b710 (ld-linux-x86-64.so.2:index) redirected to 0x4c30bc0
(index)
--4274-- REDIR: 0x401b930 (ld-linux-x86-64.so.2:strcmp) redirected to 0x4c320d0
(strcmp)
--4274-- REDIR: 0x401db20 (ld-linux-x86-64.so.2:mempcpy) redirected to
0x4c35270 (mempcpy)
--4274-- Reading syms from /lib/x86_64-linux-gnu/libc-2.23.so
--4274--   Considering /lib/x86_64-linux-gnu/libc-2.23.so ..
--4274--   .. CRC mismatch (computed 4e01d81e wanted 7d461875)
--4274--   Considering /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.23.so ..
--4274--   .. CRC is valid
--4274-- REDIR: 0x4ec8e50 (libc.so.6:strcasecmp) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4274-- REDIR: 0x4ec46d0 (libc.so.6:strcspn) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4274-- REDIR: 0x4ecb140 (libc.so.6:strncasecmp) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4274-- REDIR: 0x4ec6b40 (libc.so.6:strpbrk) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4274-- REDIR: 0x4ec6ed0 (libc.so.6:strspn) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4274-- REDIR: 0x4ec859b (libc.so.6:memcpy@GLIBC_2.2.5) redirected to
0x4a286f0 (_vgnU_ifunc_wrapper)
--4274-- REDIR: 0x4ec6850 (libc.so.6:rindex) redirected to 0x4c308a0 (rindex)
--4274-- REDIR: 0x4ebd580 (libc.so.6:malloc) redirected to 0x4c2db20 (malloc)
--4274-- REDIR: 0x4ecfbb0 (libc.so.6:strchrnul) redirected to 0x4c34da0
(strchrnul)
--4274-- REDIR: 0x4ec8800 (libc.so.6:__GI_mempcpy) redirected to 0x4c34fa0
(__GI_mempcpy)
```

```
2
--4274-- REDIR: 0x4ebd940 (libc.so.6:free) redirected to 0x4c2ed80 (free)
==4274==
==4274== HEAP SUMMARY:
==4274==     in use at exit: 4 bytes in 1 blocks
==4274==   total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==4274==
==4274== Searching for pointers to 1 not-freed blocks
==4274== Checked 64,544 bytes
==4274==
==4274== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4274==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4274==    by 0x400577: main (leak.c:8)
==4274==
==4274== LEAK SUMMARY:
==4274==    definitely lost: 4 bytes in 1 blocks
==4274==    indirectly lost: 0 bytes in 0 blocks
==4274==      possibly lost: 0 bytes in 0 blocks
==4274==    still reachable: 0 bytes in 0 blocks
==4274==         suppressed: 0 bytes in 0 blocks
==4274==
==4274== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==4274== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Observe that the `valgrind` shows the line number at the leak summary.

Lets observe the invalid memory access detection using the valgrind. Lets modify the above program as below.

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

int main(void)
{
    int *var;

    var = malloc(sizeof(int));

    *var = 2;

    // increment by integer bytes .. this makes var pointing to an invalid
location
    var ++;

    *var = 4;

    printf("%d\n", *var);
}
```

Recompile the program with `gcc -g` option.

Running the `valgrind` on the above program results in the following output.

```
devnaga@devnaga-VirtualBox:~/personal$ valgrind -v --leak-check=full --leak-
resolution=high --track-origins=yes ./a.out
==4366== Memcheck, a memory error detector
==4366== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4366== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4366== Command: ./a.out
==4366==
--4366-- Valgrind options:
--4366--    -v
--4366--    --leak-check=full
--4366--    --leak-resolution=high
--4366--    --track-origins=yes
--4366-- Contents of /proc/version:
--4366--   Linux version 4.4.0-31-generic (buildd@lgw01-16) (gcc version 5.3.1
20160413 (Ubuntu 5.3.1-14ubuntu2.1) ) #50-Ubuntu SMP Wed Jul 13 00:07:12 UTC
2016
--4366--
--4366-- Arch and hwcaps: AMD64, LittleEndian, amd64-cx16-lzcnt-rdtscp-sse3-avx
--4366-- Page sizes: currently 4096, max supported 4096
--4366-- Valgrind library directory: /usr/lib/valgrind
--4366-- Reading syms from /home/devnaga/personal/a.out
--4366-- Reading syms from /lib/x86_64-linux-gnu/ld-2.23.so
--4366--   Considering /lib/x86_64-linux-gnu/ld-2.23.so ..
--4366--   .. CRC mismatch (computed 30b9eb7c wanted d576ac3f)
--4366--   Considering /usr/lib/debug/lib/x86_64-linux-gnu/ld-2.23.so ..
--4366--   .. CRC is valid
--4366-- Reading syms from /usr/lib/valgrind/memcheck-amd64-linux
--4366--   Considering /usr/lib/valgrind/memcheck-amd64-linux ..
--4366--   .. CRC mismatch (computed 5529a2c7 wanted 5bd23904)
--4366--    object doesn't have a symbol table
--4366--    object doesn't have a dynamic symbol table
--4366-- Scheduler: using generic scheduler lock implementation.
--4366-- Reading suppressions file: /usr/lib/valgrind/default.supp
==4366== embedded gdbserver: reading from /tmp/vgdb-pipe-from-vgdb-to-4366-by-
devnaga-on-???
==4366== embedded gdbserver: writing to   /tmp/vgdb-pipe-to-vgdb-from-4366-by-
devnaga-on-???
==4366== embedded gdbserver: shared mem   /tmp/vgdb-pipe-shared-mem-vgdb-4366-
by-devnaga-on-???
==4366==
==4366== TO CONTROL THIS PROCESS USING vgdb (which you probably
==4366== don't want to do, unless you know exactly what you're doing,
==4366== or are doing some strange experiment):
==4366==   /usr/lib/valgrind/../../bin/vgdb --pid=4366 ...command...
==4366==
==4366== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==4366==   /path/to/gdb ./a.out
==4366== and then give GDB the following command
==4366==   target remote | /usr/lib/valgrind/../../bin/vgdb --pid=4366
==4366== --pid is optional if only one valgrind process is running
==4366==
```

```
--4366-- REDIR: 0x401cdc0 (ld-linux-x86-64.so.2:strlen) redirected to
0x3809e181 (???)
--4366-- Reading syms from /usr/lib/valgrind/vgpreload_core-amd64-linux.so
--4366--   Considering /usr/lib/valgrind/vgpreload_core-amd64-linux.so ..
--4366--   .. CRC mismatch (computed a30c8eaa wanted 7ae2fed4)
--4366--    object doesn't have a symbol table
--4366-- Reading syms from /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
--4366--   Considering /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so ..
--4366--   .. CRC mismatch (computed 402c2ab5 wanted 745f25ae)
--4366--    object doesn't have a symbol table
==4366== WARNING: new redirection conflicts with existing -- ignoring it
--4366--     old: 0x0401cdc0 (strlen              ) R-> (0000.0) 0x3809e181 ???
--4366--     new: 0x0401cdc0 (strlen              ) R-> (2007.0) 0x04c31020
strlen
--4366-- REDIR: 0x401b710 (ld-linux-x86-64.so.2:index) redirected to 0x4c30bc0
(index)
--4366-- REDIR: 0x401b930 (ld-linux-x86-64.so.2:strcmp) redirected to 0x4c320d0
(strcmp)
--4366-- REDIR: 0x401db20 (ld-linux-x86-64.so.2:mempcpy) redirected to
0x4c35270 (mempcpy)
--4366-- Reading syms from /lib/x86_64-linux-gnu/libc-2.23.so
--4366--   Considering /lib/x86_64-linux-gnu/libc-2.23.so ..
--4366--   .. CRC mismatch (computed 4e01d81e wanted 7d461875)
--4366--   Considering /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.23.so ..
--4366--   .. CRC is valid
--4366-- REDIR: 0x4ec8e50 (libc.so.6:strcasecmp) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4366-- REDIR: 0x4ec46d0 (libc.so.6:strcspn) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4366-- REDIR: 0x4ecb140 (libc.so.6:strncasecmp) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4366-- REDIR: 0x4ec6b40 (libc.so.6:strpbrk) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4366-- REDIR: 0x4ec6ed0 (libc.so.6:strspn) redirected to 0x4a286f0
(_vgnU_ifunc_wrapper)
--4366-- REDIR: 0x4ec859b (libc.so.6:memcpy@GLIBC_2.2.5) redirected to
0x4a286f0 (_vgnU_ifunc_wrapper)
--4366-- REDIR: 0x4ec6850 (libc.so.6:rindex) redirected to 0x4c308a0 (rindex)
--4366-- REDIR: 0x4ebd580 (libc.so.6:malloc) redirected to 0x4c2db20 (malloc)
==4366== Invalid write of size 4
==4366==    at 0x40058F: main (leak.c:14)
==4366==  Address 0x5203044 is 0 bytes after a block of size 4 alloc'd
==4366==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4366==    by 0x400577: main (leak.c:8)
==4366==
==4366== Invalid read of size 4
==4366==    at 0x400599: main (leak.c:16)
==4366==  Address 0x5203044 is 0 bytes after a block of size 4 alloc'd
==4366==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4366==    by 0x400577: main (leak.c:8)
```

```
==4366==
--4366-- REDIR: 0x4ecfbb0 (libc.so.6:strchrnul) redirected to 0x4c34da0
(strchrnul)
--4366-- REDIR: 0x4ec8800 (libc.so.6:__GI_mempcpy) redirected to 0x4c34fa0
(__GI_mempcpy)
4
--4366-- REDIR: 0x4ebd940 (libc.so.6:free) redirected to 0x4c2ed80 (free)
==4366==
==4366== HEAP SUMMARY:
==4366==     in use at exit: 4 bytes in 1 blocks
==4366==   total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==4366==
==4366== Searching for pointers to 1 not-freed blocks
==4366== Checked 64,544 bytes
==4366==
==4366== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4366==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4366==    by 0x400577: main (leak.c:8)
==4366==
==4366== LEAK SUMMARY:
==4366==    definitely lost: 4 bytes in 1 blocks
==4366==    indirectly lost: 0 bytes in 0 blocks
==4366==      possibly lost: 0 bytes in 0 blocks
==4366==    still reachable: 0 bytes in 0 blocks
==4366==         suppressed: 0 bytes in 0 blocks
==4366==
==4366== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
==4366==
==4366== 1 errors in context 1 of 3:
==4366== Invalid read of size 4
==4366==    at 0x400599: main (leak.c:16)
==4366==  Address 0x5203044 is 0 bytes after a block of size 4 alloc'd
==4366==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4366==    by 0x400577: main (leak.c:8)
==4366==
==4366==
==4366== 1 errors in context 2 of 3:
==4366== Invalid write of size 4
==4366==    at 0x40058F: main (leak.c:14)
==4366==  Address 0x5203044 is 0 bytes after a block of size 4 alloc'd
==4366==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==4366==    by 0x400577: main (leak.c:8)
==4366==
==4366== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Observe the 'invalid read' and 'invalid write' errors that the `valgrind` has detected. The fix is to allocate a sufficient memory to `var` variable and use indexes than incrementing the var variable.

Advanced concepts

In this section of the book, i am going to describe some of the hidden and very nice features of the linux OS as a whole.

AF_ALG

AF_ALG is an interface provided by the kernel to perform the crypto operations.

The base example is provided in the LKML list here (https://lwn.net/Articles/410833/).

Here is the slightly modified program from the same list. This program does the sha1.

You can download this program here (https://github.com/DevNaga/gists/blob/master/crypto_sha1.c).

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <linux/if_alg.h>

int main(int argc, char **argv)
{
    int cli_fd;
    int ser_fd;

    struct sockaddr_alg sa = {
        .salg_family = AF_ALG,
        .salg_type = "hash",
        .salg_name = "sha1",
    };
    int i;
    char buf[1000];

    if (argc != 2) {
        fprintf(stderr, "%s [input]\n", argv[0]);
        return -1;
    }

    ser_fd = socket(AF_ALG, SOCK_SEQPACKET, 0);
    if (ser_fd < 0)
        return -1;

    int ret;

    ret = bind(ser_fd, (struct sockaddr *)&sa, sizeof(sa));
```

```
    if (ret < 0)
        return -1;

    cli_fd = accept(ser_fd, NULL, NULL);
    if (cli_fd < 0)
        return -1;

    write(cli_fd, argv[1], strlen(argv[1]));
    ret = read(cli_fd, buf, sizeof(buf));
    if (ret < 0)
        return -1;

    for (i = 0; i < ret; i ++) {
        printf("%02x", buf[i] & 0xff);
    }
    printf("\n");

    close(cli_fd);
    close(ser_fd);

    return 0;
}
```

Here is another that does the md5. You can download the program [here (https://github.com/DevNaga/gists/blob/master/crypto_md5.c)](https://github.com/DevNaga/gists/blob/master/crypto_md5.c).

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <linux/if_alg.h>

int main(int argc, char **argv)
{
    int cli_fd;
    int ser_fd;

    struct sockaddr_alg sa = {
        .salg_family = AF_ALG,
        .salg_type = "hash",
        .salg_name = "md5",
    };
    int i;
    char buf[1000];

    if (argc != 2) {
        fprintf(stderr, "%s [input]\n", argv[0]);
        return -1;
```

```
    }

    ser_fd = socket(AF_ALG, SOCK_SEQPACKET, 0);
    if (ser_fd < 0)
        return -1;

    int ret;

    ret = bind(ser_fd, (struct sockaddr *)&sa, sizeof(sa));
    if (ret < 0)
        return -1;

    cli_fd = accept(ser_fd, NULL, NULL);
    if (cli_fd < 0)
        return -1;

    write(cli_fd, argv[1], strlen(argv[1]));
    ret = read(cli_fd, buf, sizeof(buf));
    if (ret < 0)
        return -1;

    for (i = 0; i < ret; i ++) {
        printf("%02x", buf[i] & 0xff);
    }
    printf("\n");

    close(cli_fd);
    close(ser_fd);

    return 0;
}
```

Another sample program to describe the available hash functions is here.

You can also download the program in [here](https://github.com/DevNaga/gists/blob/master/crypto_hashes.c)

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <linux/if_alg.h>

int main(int argc, char **argv)
{
    int cli_fd;
    int ser_fd;
```

```c
    struct sockaddr_alg sa = {
        .salg_family = AF_ALG,
        .salg_type = "hash",
        .salg_name = "sha1",
    };
    int i;
    char buf[1000];

    if (argc != 3) {
        fprintf(stderr, "%s [hash_function] [input]\n"
                        "Where hash_function is one of the below\n"
                        "\t 1. crct10dif\n"
                        "\t 2. sha224\n"
                        "\t 3. sha256\n"
                        "\t 4. sha1\n"
                        "\t 5. md5\n"
                        "\t 6. md4\n",
                        argv[0]);
        return -1;
    }

    strcpy(sa.salg_name, argv[1]);

    ser_fd = socket(AF_ALG, SOCK_SEQPACKET, 0);
    if (ser_fd < 0)
        return -1;

    int ret;

    ret = bind(ser_fd, (struct sockaddr *)&sa, sizeof(sa));
    if (ret < 0)
        return -1;

    cli_fd = accept(ser_fd, NULL, NULL);
    if (cli_fd < 0)
        return -1;

    write(cli_fd, argv[2], strlen(argv[2]));
    ret = read(cli_fd, buf, sizeof(buf));
    if (ret < 0)
        return -1;

    for (i = 0; i < ret; i ++) {
        printf("%02x", buf[i] & 0xff);
    }
    printf("\n");

    close(cli_fd);
    close(ser_fd);

    return 0;
}
```

scatter gather i/o

The system call layer supports the two system calls `readv` and `writev` which can be used to perform scatter gather i/o respectively.

The `readv` prototype is described below.

```
int readv(int fd, struct iovec *iov, size_t count);
```

the `writev` prototype is described below.

```
int writev(int fd, struct iovec *iov, size_t count);
```

Both of the above system calls accepts the structure of the form `struct iovec`. it is defined as below,

```
struct iovec {
    void *iov_base;
    int iov_len;
}
```

The `iov_base` contain the pointer to the bytes that are 1) either to be written to file or 2) read from file and to be copied to. The `iov_len` is the length of the bytes that are available in the `iov_base`.

include `sys/uio.h` when using `readv` and `writev` system calls.

there can be as many as the `struct iovec` objects, and the number of such structures is defined in the `count` argument for the `readv` and `writev` system calls.

the `sysconf` API gets the maximum vectors of type `struct iovec` to be 1024.

Below is an example of such. Download [here](https://github.com/DevNaga/gists/blob/master/sysconf.c)

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

int main()
{
    long iov = sysconf(_SC_IOV_MAX);

    printf("max sysconf iov %ld\n", iov);

    return 0;
}
```

example of the `readv` is below. Download [here](https://github.com/DevNaga/gists/blob/master/readev.c)

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

```c
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int fd;
    struct iovec iov[4];

    if (argc != 2) {
        fprintf(stderr, "<%s> filename\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "failed to open file %s\n", argv[1]);
        return -1;
    }

    int i;

    for (i = 0; i < 4; i ++) {
        iov[i].iov_base = calloc(1, 1024);
        if (!iov[i].iov_base) {
            return -1;
        }

        iov[i].iov_len = 1024;
    }

    int ret;

    ret = readv(fd, iov, 4);
    if (ret <= 0) {
        fprintf(stderr, "failed to read from file\n");
    }

    int bytes = 0;

    if (ret < 4 * 1024) {
        fprintf(stderr, "read [%d] expected 4096 \n", ret);
    }

    for (i = 0; i < 4; i ++) {
        if (bytes < ret) {
            char *content = iov[i].iov_base;

            fprintf(stderr, "---- iov[%d]: size [%ld]-------\n", i,
iov[i].iov_len);
```

```
            fprintf(stderr, "%s", content);

            free(content);

            bytes += iov[i].iov_len;
        }
    }

    close(fd);

    return 0;
}
```

Example of the `writev` is as below. Download it from [here (https://github.com/DevNaga/gists/blob/master/sgio.c)](https://github.com/DevNaga/gists/blob/master/sgio.c)

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>

int main(int argc, char **argv)
{
    int fd;

    if (argc < 2) {
        fprintf(stderr, "%s [filename] [content]\n", argv[0]);
        return -1;
    }

    fd = open(argv[1], O_CREAT | O_RDWR, S_IRWXU);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s\n", argv[1]);
        return -1;
    }

    int i;
    int count = 0;
    struct iovec iov[argc - 1];

    for (i = 2; i < argc; i ++) {
        iov[i - 2].iov_base = argv[i];
        iov[i - 2].iov_len = strlen(argv[i]);
        count ++;
    }

    char newline_str[] =  "\n";

    iov[count].iov_base = newline_str;
    iov[count].iov_len = strlen(newline_str);
    count ++;

    writev(fd, iov, count);

    close(fd);
}
```

the `/proc/crypto` contain the list of hash algorithms, ciphers supported by the kernel.

sendmsg / recvmsg

Raw sockets

The Raw sockets, are useful to perform very low level operations from the L2 to the upper layer bypassing the linux kernel networking stack. They are very useful to craft a packet and test the network stack ability of the device under test.

They are also useful to offload the networking stack to the user space to improve the performance or throughput.

The family AF_RAW is used for this purpose. The packet type AF_PACKET is used.

Below is an example of raw sockets that use ethernet header encapsulation to send a packet with "hello world" every 1 sec.

```c
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <netinet/ether.h>
#include <linux/if_packet.h>
#include <net/if.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int sock;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> interface\n", argv[0]);
        return -1;
    }

    sock = socket(AF_PACKET, SOCK_RAW, 0);
    if (sock < 0) {
        return -1;
    }

    uint8_t sendbuf[2048];
    uint8_t *data;
    int datalen = 0;
    int ifindex;
    struct ether_header *eh;

    eh = (struct ether_header *)sendbuf;

    data = sendbuf + sizeof(*eh);
    datalen += sizeof(*eh);

    struct ifreq ifr;

    memset(&ifr, 0, sizeof(ifr));
```

```
    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(sock, SIOCGIFINDEX, &ifr);
    if (ret < 0) {
        return -1;
    }

    ifindex = ifr.ifr_ifindex;

    uint8_t dest[] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
    uint8_t srcmac[6];

    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(sock, SIOCGIFHWADDR, &ifr);
    if (ret < 0) {
        return -1;
    }

    memcpy(srcmac, (uint8_t *)(ifr.ifr_hwaddr.sa_data), 6);

    char *msg = "hello world";
    memcpy(data, msg, strlen(msg) + 1);
    datalen += strlen(msg) + 1;

    memcpy(eh->ether_shost, srcmac, 6);
    memcpy(eh->ether_dhost, dest, 6);
    eh->ether_type = htons(0x0800);

    struct sockaddr_ll lladdr;

    lladdr.sll_ifindex = ifindex;

    lladdr.sll_halen = ETH_ALEN;

    lladdr.sll_addr[0] = dest[0];
    lladdr.sll_addr[1] = dest[1];
    lladdr.sll_addr[2] = dest[2];
    lladdr.sll_addr[3] = dest[3];
    lladdr.sll_addr[4] = dest[4];
    lladdr.sll_addr[5] = dest[5];

    while (1) {
        sleep(1);

        ret = sendto(sock, sendbuf, datalen, 0, (struct sockaddr *)&lladdr,
sizeof(lladdr));
        if (ret < 0) {
            break;
        }
    }
```

```
    return 0;
}
```

another program to listen on the raw sockets is below, it also has some of the well known mac address listing.

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <netinet/ether.h>
#include <linux/if_packet.h>
#include <net/if.h>
#include <unistd.h>

struct mac_vendor_list {
    uint8_t mac[6];
    char *vendor;
} lookup[] = {
    { {0x04, 0xd3, 0xb0, 0x00, 0x00, 0x00}, "Intel"},
    { {0xb4, 0x6b, 0xfc, 0x00, 0x00, 0x00}, "Intel Corp"},
    { {0x70, 0x10, 0x6f, 0x00, 0x00, 0x00}, "HP Enterprise"},
    { {0x8c, 0x85, 0x90, 0x00, 0x00, 0x00}, "Apple Inc"},
    { {0x74, 0x40, 0xbb, 0x00, 0x00, 0x00}, "Honhai preci"},
    { {0xf0, 0x18, 0x98, 0x00, 0x00, 0x00}, "Apple Inc"},
    { {0x68, 0xfe, 0xf7, 0x00, 0x00, 0x00}, "Apple Inc"},
    { {0x54, 0x72, 0x4f, 0x00, 0x00, 0x00}, "Apple Inc"},
    { {0x30, 0x35, 0xad, 0x00, 0x00, 0x00}, "Apple Inc"},
    { {0x28, 0xc6, 0x3f, 0x00, 0x00, 0x00}, "Intel Corp"},
};

int main(int argc, char **argv)
{
    int sock;
    int ret;

    if (argc != 2) {
        fprintf(stderr, "<%s> interface\n", argv[0]);
        return -1;
    }

    sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    if (sock < 0) {
        perror("socket");
        return -1;
    }

    struct ifreq ifr;
```

```
    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(sock, SIOCGIFFLAGS, &ifr);
    if (ret < 0) {
        perror("ioctl");
        return -1;
    }

    ifr.ifr_flags |= IFF_PROMISC;
    ret = ioctl(sock, SIOCSIFFLAGS, &ifr);
    if (ret < 0) {
        perror("ioctl");
        return -1;
    }

#if 0
    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(sock, SIOCGIFINDEX, &ifr);
    if (ret < 0) {
        return -1;
    }

    ifindex = ifr.ifr_ifindex;

    uint8_t dest[] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
    uint8_t srcmac[6];

    memset(&ifr, 0, sizeof(ifr));

    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(sock, SIOCGIFHWADDR, &ifr);
    if (ret < 0) {
        return -1;
    }

    memcpy(srcmac, (uint8_t *)(ifr.ifr_hwaddr.sa_data), 6);

    char *msg = "hello world";
    memcpy(data, msg, strlen(msg) + 1);
    datalen += strlen(msg) + 1;

    memcpy(eh->ether_shost, srcmac, 6);
    memcpy(eh->ether_dhost, dest, 6);
    eh->ether_type = htons(0x0800);

    struct sockaddr_ll lladdr;

    lladdr.sll_ifindex = ifindex;

    lladdr.sll_halen = ETH_ALEN;
```

```
    lladdr.sll_addr[0] = dest[0];
    lladdr.sll_addr[1] = dest[1];
    lladdr.sll_addr[2] = dest[2];
    lladdr.sll_addr[3] = dest[3];
    lladdr.sll_addr[4] = dest[4];
    lladdr.sll_addr[5] = dest[5];
#endif

    while (1) {
        int i;
        uint8_t rxbuf[2048];
        struct ether_header *eh;
        char *vendor_name = NULL;

        eh = (struct ether_header *)rxbuf;

        ret = recvfrom(sock, rxbuf, sizeof(rxbuf), 0, NULL, NULL);
        if (ret < 0) {
            break;
        }

        for (i = 0; i < sizeof(lookup) / sizeof(lookup[0]); i ++) {
            if ((lookup[i].mac[0] == eh->ether_shost[0]) &&
                (lookup[i].mac[1] == eh->ether_shost[1]) &&
                (lookup[i].mac[2] == eh->ether_shost[2])) {
                vendor_name = lookup[i].vendor;
                break;
            }
        }

        printf("ether src: %02x:%02x:%02x:%02x:%02x:%02x (%s)\n",
                            eh->ether_shost[0],
                            eh->ether_shost[1],
                            eh->ether_shost[2],
                            eh->ether_shost[3],
                            eh->ether_shost[4],
                            eh->ether_shost[5], vendor_name);
    }

    return 0;
}
```

PPS

Pulse Per Second1 is used for precise time measurement. The PPS1 is a signal comes from a GPS chip that is used to adjust the time reference.

The linux kernel supports the 1PPS or PPS1 pulse signal and exposes a set of usersapce functions to deal with it and use it in the userspace. Usually, the linux kernel expose the PPS device as **/dev/pps1** or **/dev/pps0**.

There is a tool called **ppstest** written especially for this purpose. The **ppstest** is used to monitor the 1PPS signal coming from the device.

More on PPS:

- Linux PPS Wiki [http://linuxpps.org/mediawiki/index.php/Main_Page]## syslogd

Syslogd is a daemon that logs every single event to a file stored in the standard directory. The event can come from different sources such as the kernel, user programs etc.

The header file <`syslog.h`> contain the prototypes of syslog. There is also a variadic function of syslog called `vsyslog`. Just like `vprintf` or `vfprintf` the `vsyslog` could be used for the similar purposes.

Syslogd exposes the `syslog` API to the userspace programs to interact with the syslogd daemon.

The `syslog` API packs the message into the format that is understandable by the syslogd daemon and sends it to the daemon over the unix socket. The daemon then unpacks and logs the message into the file.

The syslogs are also called system logs and are stored under `/var/log/` with a common name of either `syslog` or `messages`.

With the systemd in the latest linux operating systems, the functionality of syslogd has faded. However, the syslogd is still an important gem in the embedded environment.

With different operating systems providing the same syslog API behavior and support the syslog API may become generic and portable.

The `syslog` API prototype:

```
void syslog(int priority, const char *format, ...);
```

The priority is an OR combination of the facility and level.

the facility has the following values from the manual page (`man 3 syslog`). But here we only describe those that are most commonly used and easy to get on with in the coding.

facility	description
LOG_AUTH	security / authorization messages
LOG_DAEMON	system daemon messages
LOG_KERN	kernel messages
LOG_USER	user level generated messages

the level has the following values from the manual page (`man 3 syslog`) and here also only the most used ones.

level	description
LOG_EMERG	emergency messages
LOG_ALERT	very important messages
LOG_CRIT	critical messages
LOG_ERR	error conditions
LOG_WARNING	warning conditions

LOG_NOTICE	normal messages
LOG_INFO	informative messages
LOG_DEBUG	debugging messages

however, the most common example of the `syslog` API is the following:

Default logtype is `LOG_KERN` and thus in the system logs one could see `kern.err` for `syslog(LOG_ERR, ..)` messages. To make it through user, one should use `LOG_USER`.

```
syslog(LOG_USER | LOG_ERR, "invalid length of data on the socket!\n");
```

or another form can be that

```
syslog(LOG_USER | LOG_AUTH | LOG_ALERT, "unprivilied access from
192.168.1.1:111\n");
```

The linux kernel prints all its messages to the ring buffer. The ring buffer can be accessed via the `/dev/kmsg`.

Some of syslog implementations read the `/dev/kmsg` and parse and then print them into the log file under `/var/log/`.

There is another API called `openlog` that helps much more descriptive messages be logged into system logs.

```
int openlog(const char *ident, int option, int facility);
```

The `openlog` API creates a connection to the syslogd. This is used to prepend the `ident` to every string the program writes the message to the system logs.
The option is usually `LOG_PID` so that the messages that could be logged to the system logs will have the PID of the process that is being logged.
The facility is being set to 0.

One example below describe about using `openlog` in the program.

```
openlog(argv[0], LOG_PID, 0);
```

where `argv[0]` is the program name, if the program accepts the command line arguments.

/dev/kmsg

`/dev/kmsg` is a device file that stores the kernel logging information.

Appendix. A docker

docker is an alternative but very different to the virtualisation platforms such as VirtualBox and VMWare.

starting docker daemon:

`docker -d` as a root to start the deamon

docker images to list available docker images.

```
dev@hanzo:~$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
ubuntu                        latest              ab035c88d533        2 weeks
ago         188 MB
xdrum/openwrt                 latest              2ff262d9c211        5 months
ago         5.2 MB
telsaleh/iot-discovery        latest              9403bdcb4a51        6 months
ago         899.5 MB
openwrt-x86-generic-rootfs    latest              6558146e8176        10 months
ago         5.283 MB
ubuntu                        14.04               07f8e8c5e660        11 months
ago         188.3 MB
hello-world                   latest              91c95931e552        11 months
ago         910 B
solarkennedy/openwrt          latest              6c74e6ce9d45        23 months
ago         14.44 MB
```

docker ps to list containers.

```
dev@hanzo:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
496911dda486        ubuntu:latest       "/bin/bash"         21 seconds ago
Up 20 seconds                           sharp_lumiere
```

docker logs <containerid> will gives us the latest screenshots of the commands that we performed.

```
dev@hanzo:~$ docker logs 496911dda486
root@496911dda486:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
root@496911dda486:/# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   21 ?        00:00:00 ps
root@496911dda486:/# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 bash
   22 ?        00:00:00 ps
root@496911dda486:/# clear
```

docker run command makes a container run.

```
dev@hanzo:~$ docker run -a stdin -a stdout -i -t ubuntu /bin/bash
root@496911dda486:/#
```

when `docker run` is performed, it gives a random name to the container. This can be avoided by providing a name at run time like below.

```
docker run --name my-ubuntu ubuntu:14.04
```

in the above command the name my-ubuntu will be the name of the running container.

docker info lists system wide information

```
dev@hanzo:~$ docker info
Containers: 31
Images: 49
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 111
 Dirperm1 Supported: true
Execution Driver: native-0.2
Kernel Version: 3.19.0-15-generic
Operating System: Ubuntu 15.04
CPUs: 1
Total Memory: 707.1 MiB
Name: hanzo
ID: T3XK:VG72:W6OE:W7E6:BZFE:YPZE:ZJLC:WPZL:WBGU:WRTY:RSU2:OWVU
Username: devnaga4
Registry: [https://index.docker.io/v1/]
```

docker stats display a live stream of one or more containers.

```
CONTAINER           CPU %               MEM USAGE/LIMIT       MEM %
NET I/O
496911dda486        0.00%               34.39 MiB/707.1 MiB   4.86%
21.58 MiB/492 KiB
```

stopping a container is done with the following command.

```
docker stop my-ubuntu
```

Appendix. B Build systems

###1. OpenWRT (https://openwrt.org) build system

- The most popular router based OS platform. Runs linux kernel as the main OS.
- Used by many router vendors in their production systems.
- Very simple root file system and provides easy access to the development and test.
- Some missing components that are needed for the production systems (such as CLI, Clean web-interface, one click setup, customization etc). Each vendor implements their own production features to release in the market.
- Has a very versatile and modular build system. Sometimes it is hard to program the build Makefile, but most of the time a copy and paste of a sample Makefile would do the trick.
- The packaging system allows to configure the final software image on to the embedded device allowing it to selectively add or delete software components in the image.
- Allows generation of a firmware image to many target variants: X86, ARM, MIPS, PPC etc. Supports many hardware platforms: Gateworks, i.MX, Broadcom, Marvell etc.
- Allows creation of an SDK for other developers to use and program the system very simply without downloading everything that OpenWRT offers.

- Contains `.config` the single configuration file that contains all the build system information including the kernel, root file system and any other packages. Can be changed by hand or via the `make menuconfig` command.
- A single `make` command or a `make V=99` command would perform the build towards generating the target firmware. The `V=99` is to verbosely build the firmware image.
- To compile a single package `make V=99 package/my-package/compile` would simply build one package.
- To clean a single package `make V=99 package/my-package/clean` would simply clean the given package.
- To clean only the build, perform `make clean`.
- The `opkg` system is the OpenWRT packaging system. Allows the binary and configs etc.. into one single compressed package.
- Has a very good community to support any issues with the OpenWRT.

####1.1 OpenWRT packages

- The OpenWRT's most powerful feature is the packages. Packages add or remove a functionality to the final image.
- They are useful to build a tiny system to a very large miniature computer system. The packages can selectively be chosen to fit the needs of the software project / product.
- Sample package Makefile for the bridge (taken from [here (https://wiki.openwrt.org/doc/devel/packages)](https://wiki.openwrt.org/doc/devel/packages)).

```
include $(TOPDIR)/rules.mk

PKG_NAME:=bridge
PKG_VERSION:=1.0.6
PKG_RELEASE:=1

PKG_BUILD_DIR:=$(BUILD_DIR)/bridge-utils-$(PKG_VERSION)
PKG_SOURCE:=bridge-utils-$(PKG_VERSION).tar.gz
PKG_SOURCE_URL:=@SF/bridge
PKG_MD5SUM:=9b7dc52656f5cbec846a7ba3299f73bd
PKG_CAT:=zcat

include $(INCLUDE_DIR)/package.mk

define Package/bridge
  SECTION:=base
  CATEGORY:=Network
  TITLE:=Ethernet bridging configuration utility
  #DESCRIPTION:=This variable is obsolete. use the Package/name/description
define instead!
  URL:=http://bridge.sourceforge.net/
endef

define Package/bridge/description
 Ethernet bridging configuration utility
 Manage ethernet bridging; a way to connect networks together to
 form a larger network.
endef

define Build/Configure
  $(call Build/Configure/Default,--with-linux-headers=$(LINUX_DIR))
endef

define Package/bridge/install
        $(INSTALL_DIR) $(1)/usr/sbin
        $(INSTALL_BIN) $(PKG_BUILD_DIR)/brctl/brctl $(1)/usr/sbin/
endef

$(eval $(call BuildPackage,bridge))
```

###2. LTIB (Linux Target Image builder)

requirements:

Ltib is called linux target image builder. From the website here (http://ltib.org/home-intro), the ltib is defined as a project tool that can be used to develop and deploy BSPs for a number of embedded platforms including powerpc, arm and coldfire.

This build system is used in most of the freescale and nxp platforms.

To download the ltib you have to run the following:

```
cvs -z3 -d:pserver:anonymous@cvs.savannah.nongnu.org:/sources/ltib co -P ltib
```

go inside the ltib directory.

```
cd ltib
```

and then perform the following command.

```
./ltib
```

The build may fail due to following reasons.

1. need to have rpm, rpmbuild and bison.
2. you might also need to install the perl modules.

on Ubuntu follow the steps:

1. `sudo apt-get install rpm`
2. `sudo apt-get install libwww-perl`
3. `sudo apt-get install`

on Fedora follow the steps:

1. `dnf install perl-libwww-perl`

and then add the following line to /etc/sudoers.

```
root ALL = NOPASSWD: /usr/bin/rpm, /opt/ltib/usr/bin/rpm
```

Appendix . C USE CASES and Frameworks

This appendix is focused on the realworld middleware software and frameworks that come in handy at the job or doing any software development. As the current world is getting more connected and communicated, In most realworld applications, at some point it is mandatory to have the following requirements.

1. communication means - network knowledge
2. positioning / GPS - self identification / IOT
3. wireless packet analysis - for cybersecurity / testing / validation
4. serialisation - for communication between machines
5. security defensive practises and mechanisms employed with in the linux kernel and exploting them in user programs

Before we go into the details, consider some of the most important points in software development :

1. Always check your warnings
2. always fix warnings and errors (enable `-Werror`)
3. in a debug environment, always have `-g and -ggdbg` and remove `-s` and `-0`. Check for **Appendix. D GCC flags** for more information
4. Always build test and run the software atleast once on a hardware before you commit changes in a multi-personal project

process manageer

seccomp

Below is one example of seccomp. Download here (https://github.com/DevNaga/gists/blob/master/prctl.c)

Example:

```
#include <stdio.h>
#include <sys/prctl.h>
#include <linux/seccomp.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);

    int fd;

    fd = open("./test", O_RDWR | O_CREAT, S_IRWXU);
    if (fd < 0) {
        return -1;
    }

    printf("opened file although restricted in seccomp\n");

    return 0;
}
```

The seccomp system call is a superset of the `prctl` method and has more detailed interface via the BPF (Berkeley Packet Filter)

A more detailed and easier implementation of such is the userspace seccomp library.

install it by running the following command:

```
sudo apt install libseccomp-dev
```

Below is one example of the seccomp API. Download here (https://github.com/DevNaga/gists/blob/master/seccomp_init.c)

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <seccomp.h>

int main()
{
    scmp_filter_ctx ctx;

    ctx = seccomp_init(SCMP_ACT_KILL);

    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);

    seccomp_load(ctx);

    int fd;

    fd = open("./test", O_RDWR | O_CREAT, S_IRWXU);
    if (fd < 0) {
        printf("failed to create file \n");
        return -1;
    }

    printf("file is created\n");

    return 0;
}
```

socket library

We are going to write a socket library as an exercise to what we have learnt in the socket programming.

The socket library has the following API. Header file is defined [here (https://github.com/DevNaga/gists/blob/master/socketlib.h)](https://github.com/DevNaga/gists/blob/master/socketlib.h)

```c
// create new tcp socket
int new_tcp_socket();

// create new udp socket
int new_udp_socket();

// create new unix tcp socket
int new_tcp_unix_socket();

// create new unix udp socket
int new_udp_unix_socket();
```

```c
// delete socket
int destroy_socket();

// create server based on `ip` and `port` and listen for `n_conn`
int new_tcp_server(const char *ip, int port, int n_conns);

// accept conn on the socket, and send out dest_addr and port back
int accept_conn(int server_fd, char *dest_addr, uint32_t *dest_port);

// create new udp server based off of `ip` and `port`
int new_udp_server(const char *ip, int port);

// create a new tcp unix server with given path
int new_tcp_unix_server(const char *path);

// create a new udp unix server with given path
int new_udp_unix_server(const char *path);

// create a new tcp unix client with given path
int new_tcp_unix_client(const char *path);

// create a new udp unix client with given path
int new_udp_unix_client(const char *path);

// send message on the fd
// dest_addr is valid for udp fd and unix fd
int send_message(int fd, void *msg, int msglen, char *dest_addr, int
dest_port);

// recieve message on the fd
// dest is valid for udp fd and unix fd
int recv_message(int fd, void *msg, int msglen, char *dest, uint32_t
*dest_port);

// callback list
struct socket_loop_callbacks {
    int fd;
    void (*callback)(int fd, void *ptr);
    struct socket_loop_callbacks *next;
};

// privatepointer to socket_loop class
struct socket_loop {
    fd_set allfd;
    struct socket_loop_callbacks *callbacks;
};

// create a pointer to socket_loop and return
void *new_socket_loop();

// use the sock_loop returned and add fd to the fd_set
int add_fd_to_socket_loop(int fd, void *sock_loop, void *ptr, void (*callback)
```

```
(int fd, void *ptr));

// remove fd from socket_loop
int del_fd_from_socket_loop(int fd, void *sock_loop, void *ptr, void
(*callback)(int fd, void *ptr));

// run the socket loop and call callbacks
int socket_loop_run(void *sock_loop);

// set multicast option on the socket for tx
int socket_set_multicast(int fd);

// subscribe for multicast data
int socket_add_to_mcast_group(int fd, char *mcast_ip);

// reuse socket address
int socket_reuse_addr(int fd);
```

Event Library

In this chapter we are going to write an event library that allows us to do multiple jobs in a single process.

GPSd

Introduction

The GPSd (http://www.catb.org/gpsd/) is a service that communicates with the GPS hardware and gets us the GPS fix and position. GPSd is an opensource software that is widely used in a lot of embedded systems including phones.

Many GPS devices provide the data over a serial connection. The GPS device reports the data over the serial interface once per second. The once per second can also be configured once per some millseconds for say 100 milliseconds. The configuration may involve setting the serial port baud rate and using special firmware commands asking the device to perform a speed change of the data.

The GPSd software then reads and parses the information coming over the serial connection and finds out if the satellites gives us a fix with current time, latitude, longitude, elevation, speed and heading. There are some more parameters that GPSd gives us such as Dilution of precision and noise statistics. However, most of the GPSes won't give us everything. The GPSes may provide the information to the userspace via their own protocol or via the NMEA protocol. NMEA protocol is a standard protocol implemented by almost all GPS receivers and understood by almost all software modules. The most prominent messages ni the NMEA protocol are GPGGA, GPGSA, GPGST, GPRMC and GPGSV. These are also called as NMEA strings. More about the NMEA is here (http://aprs.gids.nl/nmea/)

Most of the GPSes lose the fix intermittently when the device comes across the trees and building complexes. At this point of time a special algorithm such as position estimation may be employed (in rare GPSes it is implemented) and gives out the estimated GPS value.

GPSd interfacing

The GPSd is a service daemon that listens on the TCP socket for clients and opens a serial connection with (almost all) GPS devices.

The GPSd provides the data to the clients in a JSON formatted strings. The strings are then decoded at the client apps using the GPSd client library API.

API list:

The following are the most important API that are used to interface with the GPSd and get the GPS information

Here is the [link (http://www.catb.org/gpsd/libgps.html)](http://www.catb.org/gpsd/libgps.html) to the GPSd API. And part of the API description is taken from the link.

The include file for the client library is `gps.h` and the linker flags are `lgps` and `lm`.

API	description
gps_open	create a socket to the GPS daemon
gps_read	read from the GPS daemon
gps_stream	stream the report to the user
gps_unpack	parse JSON from a specified buffer into a session structure
gps_close	close the socket connection with the GPS daemon

The `gps_open` prototype is as follows.

```
int gps_open(char *server, int port, struct gps_data_t *gps_data);
```

For most of the time, the `server` and `port` arguments are set to 0. The `gps_data` structure is initialised and returned 0 on success. -1 is returned on failure.

The GPSd client library provides an API to automatically stream the fix back to the application. This is done using the `gps_stream` API.

The `gps_stream` prototype is as follows.

```
int gps_stream(struct gps_data *gps_data, unsigned int flags, void *data);
```

the `flags` arguments allows the method or / and type of streaming.

One of the most commonly used `flags` are `WATCH_ENABLE`.

The `gps_read` prototype is as follows.

```
int gps_read(struct gps_data_t *gps_data);
```

The `gps_read` API polls the GPS daemon for the data. Thus the polling frequency also does matter on your GPS fix quality or difference between two GPS fixes. The good quality of the GPS fix is when your poll rate becomes same as the update rate on the serial device (GPS hardware communication channel).

The GPSd provides another API to selectively wait for the application to wait for GPS data.

```
bool gps_waiting(struct gps_data_t *gps_data, int timeout);
```

The gps_waiting waits for the GPS data from the GPS socket for timeout milliseconds. If the data comes with in the timeout the function returns true and otherwise on a timeout, it returns false. Upon the gps_waiting returning true the gps_read API is called to get the GPS data.

The gps_close prototype is as follows.

```
void gps_close(struct gps_data_t *gps_data);
```

The gps_close closes or ends the session.

One example of the GPS API is follows.

```
int main(int argc, char **argv)
{
    struct gps_data_t gps_data;
    int ret;

    ret = gps_open(NULL, 0, &gps_data);
    if (ret < 0) {
        fprintf(stderr, "failed to open connection to GPSd\n");
        return -1;
    }

    gps_stream(&gps_data, WATCH_ENABLE, NULL);

    while (1) {
        if (gps_waiting(&gps_data, 500)) {
            if (gps_read(&gps_data) == -1) {
                fprintf(stderr, "failed to read GPS data\n");
                return -1;
            }
            if ((gps_data.status == STATUS_FIX) && (gps_data.mode >= 2)) {
                fprintf(stderr, "GPS mode : %d\n", gps_data.mode);
                if (!isnan(gps_data.latitude)) {
                    fprintf(stderr, "GPS latitude : %f\n", gps_data.latitude);
                }

                if (!isnan(gps_data.longitude)) {
                   fprintf(stderr, "GPS longitude : %f\n", gps_data.longitude);
                }
            }
        }
    }

    gps_stream(&gps_data, WATCH_DISABLE, NULL);
    gps_close(&gps_data);
}
```

(Remember to use our select interface's timeout option is also an alternative to the gps_waiting in here that when the timer expire periodically and calling the gps_read)

mbedtls

Mbedtls is an encryption library that is similar to the **openssl**.

Mbedtls provide the hash, `encryption`, `random number generator`, `ssl` and `random` number generator.

The Mbedtls is an easy to use cryptographic library.

The Mbedtls API change continuously.. so i have kept the flags makefile that looks something like below:

```
$(MBEDTLS_DIR) = $(HOME)/GitHubProjects/mbedtls/

CFLAGS += -Wall
INCLUDES += -I$(MBEDTLS_DIR)/include/
LIBRARIES += $(MBEDTLS_DIR)/library/libmbedtls.a
$(MBEDTLS_DIR)/library/libmbedcrypto.a $(MBEDTLS_DIR)/library/libmbedx509.a
```

hash functions:

md5 API:

md5 is an algorithm that produces a 16 byte hash out of an input string. The hashing algorithms always follow the avalanche effect (A small variation in the input provides a large variation in the hash). (You can download the example here (https://github.com/DevNaga/gists/blob/master/hash_Test.c))

MD5 API description:

```
md5(input, len, hash);
```

Example:

```c
#include "mbedtls/config.h"
#include "mbedtls/platform.h"
#include "mbedtls/md5.h"
#include <stdio.h>
#include <stdint.h>

int digest_print(uint8_t *str, int len)
{
    int i;

    for (i = 0; i < len; i ++) {
        printf("%02x", str[i] & 0xff);
    }

    printf("\n");

    return 0;
}

int main(int argc, char **argv)
{
    int i;
    uint8_t digest[16];

    if (argc != 2) {
        printf("%s <string>\n", argv[0]);
        return -1;
    }

    mbedtls_md5(argv[1], strlen(argv[1]), digest);

    digest_print(digest, sizeof(digest));

    return 0;
}
```

We use the above makefile that we have created and generate our binary.
Run the binary with the arguments to see the produced 16 byte hash value. This is the hash of the input that we have given.

Cipher information

cipher list that is supported by the library can be get using the `mbedtls_cipher_list` API. (Downloadable here (https://github.com/DevNaga/gists/blob/master/cipher_list.c))

Here is the simple example,

Example:

```c
#include "mbedtls/config.h"
#include "mbedtls/platform.h"
#include "mbedtls/cipher.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    int *cipher_list;
    mbedtls_cipher_info_t *cipher_info;

    cipher_list = mbedtls_cipher_list();
    if (!cipher_list) {
        fprintf(stderr, "failed to get cipher list\n");
        return -1;
    }

    while (*cipher_list) {
        cipher_info = mbedtls_cipher_info_from_type(*cipher_list);
        fprintf(stderr, "cipher name \t%s\n", cipher_info->name);
        fprintf(stderr, "cipher keylen \t%d\n", cipher_info->key_bitlen);
        cipher_list ++;
    }

    return 0;
}
```

MD information

The list of available MD algorithms can be get using the `mbedtls_md_list` API. The below example shows how one can do. (Downloadable [here (https://github.com/DevNaga/gists/blob/master/md_list.c)](https://github.com/DevNaga/gists/blob/master/md_list.c))

```c
#include "mbedtls/config.h"
#include "mbedtls/platform.h"
#include "mbedtls/md_internal.h"
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main(int argc, char **argv)
{
    int *md_list;

    md_list = mbedtls_md_list();
    if (!md_list) {
        fprintf(stderr, "failed to get mdlist\n");
        return -1;
    }

    while (*md_list) {
        mbedtls_md_info_t *md_info;

        md_info = mbedtls_md_info_from_type(*md_list);
        fprintf(stderr, "MD name: \t%s\n", md_info->name);
        fprintf(stderr, "MD size: \t%d\n", md_info->size);

        md_list ++;
    }

    return 0;
}
```

More versatile program is below which finds the hash for a given hashing algorithm and the string.

```c
#include "mbedtls/config.h"
#include "mbedtls/platform.h"
#include "mbedtls/md_internal.h"
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main(int argc, char **argv)
{
    int *md_list;

    if (argc != 3) {
        printf("%s <algorithm> <string-to-hash>\n", argv[0]);
        return -1;
    }
```

```
    md_list = mbedtls_md_list();
    if (!md_list) {
        fprintf(stderr, "failed to get mdlist\n");
        return -1;
    }

    while (*md_list) {
        mbedtls_md_info_t *md_info;

        md_info = mbedtls_md_info_from_type(*md_list);

        if (!strcasecmp(md_info->name, argv[1])) {
            int i;
            int hash_len;
            uint8_t hash[200];

            memset(hash, 0, sizeof(hash));

            md_info->digest_func(argv[2], strlen(argv[2]), hash);

            fprintf(stderr, "MD name: \t%s\n", md_info->name);
            fprintf(stderr, "MD size: \t%d\n", md_info->size);

            printf("hash: ");
            for (i = 0; i < md_info->size; i ++) {
                printf("%02x", hash[i]);
            }

            printf("\n");
            break;
        }
        md_list ++;
    }

    return 0;
}
```

Version control systems

###Git

1. Git is very powerful version control system.
2. Created by Linus Torvalds, also the creator of Linux Operating system.
3. Git allows versioning of a file or set of files into a group called repository stored somewhere in the server.
4. Git can also be thought of as a time travel system to go back in time to find the versions of the file or set of files that are good / bad and find what changed and who changed it. Also allows going back to the older or last good set of files.

5. In most cases Git is used to maintain the code. In many other cases even the documents.
6. This very book is written on a platform that uses the Git as one of its backend.
7. installation of the git:

   on Ubuntu:

   ```
   sudo apt-get install git-all
   ```

   on Fedora:

   ```
   # dnf install git-all
   ```

8. Git commands

command	description
add	Add file contents to the index
bisect	Find by binary search the change that introduced a bug
branch	List, create, or delete branches
checkout	Checkout a branch or paths to the working tree
clone	Clone a repository into a new directory
commit	Record changes to the repository
diff	Show changes between commits, commit and working tree, etc
fetch	Download objects and refs from another repository
grep	Print lines matching a pattern
init	Create an empty Git repository or reinitialize an existing one
log	Show commit logs
merge	Join two or more development histories together
mv	Move or rename a file, a directory, or a symlink
pull	Fetch from and integrate with another repository or a local branch
push	Update remote refs along with associated objects
rebase	Forward-port local commits to the updated upstream head
reset	Reset current HEAD to the specified state
rm	Remove files from the working tree and from the index
show	Show various types of objects
status	Show the working tree status
tag	Create, list, delete or verify a tag object signed with GPG

git examples (command line)

clone a repository:

```
git clone https://github.com/DevNaga/gists.git
```

check the status of the branch

```
git status
```

configure the git username

```
git config --add user.name "devnaga"
```

confiure the git user email address

```
git config --add user.email "devendra.aaru@gmail.com"
```

configure the color screen for branch and status

```
git config --add color.diff auto
git config --add color.branch auto
git config --add color.status auto
```

configure the difftool

```
git config --add diff.tool meld
```

configure the favourite editor for the commit

```
git config --add core.editor subl
```

pull the code in the repository:

```
git pull
```

launch a difftool

```
git difftool
```

create a branch:

```
git branch new
```

delete a branch:

```
git branch -D new
```

stash local changes

```
git stash
```

list the stashed items

```
git stash list
```

show the stashed change

```
git stash show -p stash@{0}
```

where `0` is the number of the stash.

diff the local changes

```
git diff
```

counters of the diff

```
git diff --stat
```

view the changes in the staging

```
git diff --cached
```

view the counters of the changes in the staging

```
git diff --stat --cached
```

commit the changes

```
git commit
```

push the changes

```
git push origin <branch-name>
```

merge a branch

```
git merge origin/<branch-name>
```

rebase the branch

```
git rebase -i
```

where i is for the interactive rebase.

Git clients

1. https://www.gitkraken.com/
2. https://desktop.github.com/

More on the git is at the below links:

Links:

https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

Openssl

Openssl is a widely used crypto library and SSL communications library. Below are some of the examples of the openssl library

IV - Initial Vector

IV is used only at the beginning. The IV is used as the initial value for the rounds. IVs are given to the receivers to aid in decrypt stage.

symmetric key

Symmetric keys are kept secretly somewhere.

nonce - Number used only once

useful context privs

1. EVP_CIPHER_CTX

Useful API

1. EVP_BytesToKey
2. EVP_aes_256_cbc()
3. EVP_sha1
4. EVP_CIPHER_CTX_init
5. EVP_EncryptInit_ex
6. EVP_DecryptInit_ex
7. EVP_EncryptUpdate
8. EVP_EncryptFinal_ex
9. EVP_DecryptUpdate
10. EVP_DecryptFinal_ex
11. EVP_CIPER_CTX_cleanup

useful include

1. openssl/evp.h
2. openssl/sha.h

Ubuntu does not have the OpenSSL libraries. Use the following command line to install the OpenSSL libraries.

```
sudo apt install libssl-dev
```

Linker flags: **-lssl -lcrypto**

Example: SHA1 generation

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdint.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <openssl/sha.h>

int main(int argc, char **argv)
{
        int fd;
        uint8_t sha1sum[40];

        if (argc != 2) {
                return -1;
        }

        fd = open(argv[1], O_RDONLY);
        if (fd < 0) {
                return -1;
        }

        SHA_CTX c;

        SHA1_Init(&c);

        for (;;) {
                char data[2000];
                int ret;

                memset(data, 0, sizeof(data));
                ret = read(fd, data, sizeof(data));
                if (ret <= 0) {
                        break;
                }
                SHA1_Update(&c, data, ret);
        }
        SHA1_Final(sha1sum, &c);

        int i;

        printf("sha1sum: ");
        for (i = 0; i < SHA_DIGEST_LENGTH; i ++) {
                printf("%02x", sha1sum[i]);
        }
        printf("\n");

        return 0;
}
```

Example: SHA256 generation

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <openssl/sha.h>

int main(int argc, char **argv)
{
    if (argc != 2) {
        return -1;
    }

    int fd;

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        return -1;
    }

    SHA256_CTX ctx;

    SHA256_Init(&ctx);
    int i;

    int ret;

    for (;;) {
        char data[2000];

        memset(data, 0, sizeof(data));
        ret = read(fd, data, sizeof(data));
        if (ret <= 0) {
            break;
        }
        SHA256_Update(&ctx, data, ret);
    }

    uint8_t md[64];

    SHA256_Final(md, &ctx);

    printf("[%s] sha256sum: ", argv[1]);
    for (i = 0; i < SHA256_DIGEST_LENGTH; i ++) {
        printf("%02x", md[i]);
    }
    printf("\n");
}
```

Example: SHA512 generation

```c
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <openssl/sha.h>

int main(int argc, char **argv)
{
    int fd;
    int i;
    int ret;

    if (argc != 2) {
        return -1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0) {
        return -1;
    }

    SHA512_CTX ctx;

    SHA512_Init(&ctx);
    for (;;) {
        char data[2000];

        memset(data, 0, sizeof(data));
        ret = read(fd, data, sizeof(data));
        if (ret <= 0) {
            break;
        }

        SHA512_Update(&ctx, data, ret);
    }

    uint8_t md[64];

    SHA512_Final(md, &ctx);

    printf("[%s] sha512sum:  ", argv[1]);
    for (i = 0; i < SHA512_DIGEST_LENGTH; i ++) {
        printf("%02x", md[i]);
    }
    printf("\n");

    return 0;
}
```

Example: Random number generator

```c
/**
 * Program to generate rand num @ linux OS
 *
 * Dev Naga (devendra.aaru@gmail.com)
 */
#include <stdio.h>
#include <stdint.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv)
{
    int fd;
    uint8_t rand[64];
    int ret;
    int no_bytes;
    int i;

    if (argc != 2) {
        return -1;
    }

    fd = open("/dev/urandom", O_RDONLY);
    if (fd < 0) {
        return -1;
    }

    no_bytes = atoi(argv[1]);

    if (no_bytes > sizeof(rand)) {
        return -1;
    }

    memset(rand, 0, sizeof(rand));
    ret = read(fd, rand, no_bytes);
    if (ret != no_bytes) {
        return -1;
    }

    for (i = 0; i < no_bytes; i ++) {
        printf("%02x", rand[i]);
    }
    printf("\n");

    return 0;
}
```

protocol buffers

Intro

- usually the message communication involve exchanging the data between one or more machines. The data may contain variables of different types and are to be kept in the same fashion at the sender that the receiver understands. Meaning a common understanding to be known between the two or more parties.
- The information is then represented in the structured format that defines the placement and types of data that the structure can contain. The structure is then transmitted over the network.
- However, serialisation comes into the picture when we are transmitting data between two different machines of different architectures (such as big and little endian). Thus we convert everything that we transmit into the network endian (usually big endian) and transmit and the receiver converts it back to host endian (its machine endian format) and reads the data after. Then, it comes to the encoding of types and data, for ex, the `long` type is of 4 bytes on a 32 bit machine, however in the most common present world we are using almost all 64 bit machines. However, the exchange of communication is also possible with the embedded devices such as phones, GPS receivers, routers, switches etc. The exact representation of the data is to be defined in this situation. This is where the serialisation comes into the picture.
- Serialsing the data involves telling the other party the type of the each element in the structure that we are sending, the size each element can hold to etc.
- One of the serialisation formats is the protocol buffer format created by Google.
- The protocol buffers asks to define the structure that we wanted to send in a `.proto` format.
- For example:

```
package position_info;

enum ModeOfFix {
    Mode_No_fix = 0;
    Mode_2D_fix = 1;
    Mode_3D_fix = 2;
}

message gpsdata {
    required ModeOfFix fix_mode = 1 [default = Mode_No_fix];
    required double latitude = 2;
    required double longitude = 3;
    required double altitude = 4;
}
```

The above example defines a fix mode, latitude, longitude and altitude in a structure position_information. The position_information is what we wanted to send over the network. The position_information may be filled from the GPS data. See that the protobuf allows us to

send the double types over the network. The .proto file starts with a package definition to avoid naming conflicts between different packages or different structure defintions in different files.

Here we are going to concentrate mostly on the protobuf-c.

Data types of the protobuf format

data type	Description
double	double data type in C
float	float data type in C
int32	Variable length encoding and only encodes positive 32 bit numbers
int64	Variable length encoding and only encodes positive 64 bit numbers
uint32	only positive 32 bit
uint64	only positive 64 bit
sint32	Variable length encoding and only encodes negative 32 bit numbers
sint64	Variable length encoding and only encode negative 64 bit numbers
fixed32	fixed 32 bit unsigned
fixed64	fixed 64 bit unsigned
sfixed32	always 32 bit signed
sfixed64	always 64 bit signed
bool	boolean data type
string	strings
bytes	byte array

For C, there is another latest C bindings implementation of protobuf is here (https://github.com/protobuf-c/protobuf-c.git). In order to use this first we need protobuf compiler which can be found here (https://github.com/google/protobuf.git).

To compile the protobuf, here are the following instructions.

```
./autogen.sh
./configure
make
sudo make install
```

You might encounter an issue in locating the library `libprotobuf.so.17`, simply do the following at `protobuf`.

```
cp -r ./src/.libs/libprotobuf.so* /usr/lib/
sudo ldconfig
```

The same steps are followed for the protobuf-c bindings.

Lets run the compiler on the above proto definition as follows. Copy the contents to a file and name it `test.proto`.

```
protoc-c --c_out=. test.proto
```

The above command generates one .C file and one .h file.

Here is a brief explanation on the protobuf file above.

The message field is a C structure. The enum is a C enum. each element of the message or the enum must contain a number that defines the order of the elements that appear in the output buffer. The message is used to define the content of the data that is to be serialized.

Each element described, can be an optional element (only rarely used thus no need to encode it always) and mandatory elements. The optional elements are used to reduce the overall encoded payload. The mandatory elements are tagged with required attribute before the declaration of the variable. The optional elements are tagged with optional attribute.

The sample test program that utilizes the generated code is as follows. The below code encodes / serializes the structured data.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include "test.pb-c.h"

int main(int argc, char **argv)
{
    PositionInfo__Gpsdata gps = POSITION_INFO__GPSDATA__INIT;
    int i;
    uint8_t buf[300];
    int len;

    gps.fix = POSITION_INFO__MODE_OF_FIX__Mode_No_fix;
    gps.latitude = 1.1;
    gps.longitude = 12.2;
    gps.altitude = 3.3;

    len = position_info__gpsdata__get_packed_size(&gps);

    position_info__gpsdata__pack(&gps, buf);

    fprintf(stderr, "Writing %d serialized bytes\n", len);

    for (i = 0; i < len; i ++) {
        printf("%02x", buf[i]);
    }

    printf("\n");

    return 0;
}
```

The main important functions are getting the length of a payload and packing the bytes into the buf object. Here are the two APIs that are auto generated by the compiler.

```
position_info__gpsdata__get_packed_size
position_info__gpsdata__pack
```

The generated bytes are then copied over to the network using the networking sockets described in the before chapters in this book.

The sample code above with the generated C file is compiled as follows. Copy the sample code into a file call posinfo.c.

```
gcc posinfo.c test.pb-c.c -lprotobuf-c -I.
```

Running the program gives the following.

```
Writing 29 serialized bytes
0800119a9999999999f13f19666666666666628402166666666666660a40
```

Lets add code to decode the above messgae. The final program looks below.

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include "test.pb-c.h"

int main(int argc, char **argv)
{
    PositionInfo__Gpsdata gps = POSITION_INFO__GPSDATA__INIT;
    PositionInfo__Gpsdata *gps_dec;
    int i;
    uint8_t buf[300];
    int len;

    gps.fix = POSITION_INFO__MODE_OF_FIX__Mode_No_fix;
    gps.latitude = 1.1;
    gps.longitude = 12.2;
    gps.altitude = 3.3;

    len = position_info__gpsdata__get_packed_size(&gps);

    position_info__gpsdata__pack(&gps, buf);

    fprintf(stderr, "Writing %d serialized bytes\n", len);

    for (i = 0; i < len; i ++) {
        printf("%02x", buf[i]);
    }

    printf("\n");

    gps_dec = position_info__gpsdata__unpack(NULL, len, buf);
    if (!gps_dec) {
        return -1;
    }

    printf("fix %d\n", gps_dec->fix);
    printf("lat %f long %f alt %f\n",
                    gps_dec->latitude,
                    gps_dec->longitude,
                    gps_dec->altitude);

    position_info__gpsdata__free_unpacked(gps_dec, NULL);

    return 0;
}
```

The result of the output when compiled and run is as follows.

```
Writing 29 serialized bytes
0800119a9999999999f13f19666666666666628402166666666666660a40
fix 0
lat 1.100000 long 12.200000 alt 3.300000
```

One of the other features of the protobuf are the use of repeated fields and optional fields. Sometimes, we would like to pass an array of n members whose size is known dynamically at the run time. So protobuf support this concept.

Consider the below proto file for example.

```
message repeated_msg {
    repeated int32 n_apples = 1;
};
```

Now, similar to this, the optional elements are tagged before.

```
message repeated_msg_optional {
    repeated int32 n_apples = 1;
    required int32 n_baskets = 2;
    optional int32 n_apple_trees = 3;
}
```

Let's say that the `n_baskets` in the above proto definition is not filled while serializing. Thus it may contain the garbage data and the `default` attribute comes to the rescue.

The below definition fixes that.

```
message repeated_msg_optional {
    repeated int32 n_apples = 1;
    required int32 n_baskets = 2 [default = 10]
    optional int32 n_apple_trees = 3;
}
```

The apples are used to make the apple juice and lets' link the apples structure into the juice structure.

```
// juice structure
message juice {
    required apples quantity = 1;
    required int32 liters = 2;
}

// apple structure
message apples {
    repeated n_apples = 1;
    required n_baskets = 2 [default = 10]
}
```

##More on protocol buffers:

[1]. https://developers.google.com/protocol-buffers/
[2]. https://developers.google.com/protocol-buffers/docs/proto?csw=1

CAN controller (socketCAN interface)

Linux socketCAN layer

linux interface provides a virtual socket CAN layer that is used from the app just like the use of `socket` with `AF_RAW`, linux socketCAN layer provides `AF_CAN` family.

in case there is no CAN hardware, the kernel provide a virtual linux driver.

Here's how to setup the virtual CAN interface,

```
sudo modprobe can_dev
sudo modprobe can
sudo modprobe can_raw
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

cross verify if the link is available,

```
ip link show vcan0
```

this should show up `vcan0` interface.

install `can-tools`. On Ubuntu, one can install using `apt`

```
sudo apt install can-tools
```

use header file `linux/can.h` for any of the CAN data structures.

The CAN protocol uses the family `AF_CAN` (defined in linux/can.h) and the `CAN_RAW` to send raw CAN frames.

A CAN frame is a TLV data frame where the TAG refers to the CAN id, length is the length of the Data field and V is the data in bytes. The data could go till maximum of 8 bytes.

the header `linux/can.h` contains the can frame `struct can_frame` defined:

```
struct can_frame {
    canid_t canid;
    __u8 dlc;
    __u8 pad;
    __u8 res0;
    __u8 res1;
    __u8 data[CAN_MAX_DLEN];
};
```

Below example provides the usage of the linux CAN layer APIs. Download [here (https://github.com/DevNaga/gists/blob/master/can_frame.c)](https://github.com/DevNaga/gists/blob/master/can_frame.c)

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <string.h>
#include <unistd.h>
```

```c
int main(int argc, char **argv)
{
    int fd;
    struct sockaddr_can can;
    struct can_frame canframe;
    struct ifreq ifr;
    int ret;

    fd = socket(AF_CAN, SOCK_RAW, CAN_RAW);
    if (fd < 0) {
        return -1;
    }

    strcpy(ifr.ifr_name, "vcan0");
    ret = ioctl(fd, SIOCGIFINDEX, &ifr);
    if (ret < 0) {
        return -1;
    }

    can.can_family = AF_CAN;
    can.can_ifindex = ifr.ifr_ifindex;

    ret = bind(fd, (struct sockaddr *)&can, sizeof(can));
    if (ret < 0) {
        return -1;
    }

    static int i;

    while (1) {
        sleep(1);

        canframe.can_id = 0x600 + i;
        canframe.can_dlc = 8;
        canframe.data[0] = 0x44 + i;
        canframe.data[1] = 0x44 + i;
        canframe.data[2] = 0x44 + i;
        canframe.data[3] = 0x44 + i;
        canframe.data[4] = 0x44 + i;
        canframe.data[5] = 0x44 + i;
        canframe.data[6] = 0x44 + i;
        canframe.data[7] = 0x44 + i;

        printf("write %02x\n", canframe.can_id);

        ret = write(fd, (void *)&canframe, sizeof(canframe));
        if (ret < 0) {
            return -1;
        }

        i ++;
```

```
        if (i > 4) {
            i = 0;
        }
    }

    return 0;
}
```

in the above example, the structure `struct sockaddr_can` is used to fill in the CAN specific information such as `ifindex`, `family`, so that a `send` operation can be performed on the CAN device.

the `ifindex` is interface index of the device which can be found using `SIOCGIFINDEX` ioctl call. the ioctl call as described in before topics can be used to get the interface index on a CAN device.

to bind the socket to all the interfaces the `can_ifindex` must be set to 0.

If the `bind` is done on all interfaces, then it is recommended to use `recvfrom` than `recv` to know which interface the CAN frames being received. and to send it is recommended to use `sendto` than `send` or `write`.

The above program writes a series of CAN messages with IDs ranging from 0x600 till 0x604 and after that repeats the same every second.

compile and run the above program. The program expects to have a virtual can device `vcan0`. First the interface needs to be created and configured before running the program.

now after running the above program, run the can trace.

```
candump vcan0

  vcan0  600   [8]  44 44 44 44 44 44 44 44
  vcan0  601   [8]  45 45 45 45 45 45 45 45
  vcan0  602   [8]  46 46 46 46 46 46 46 46
  vcan0  603   [8]  47 47 47 47 47 47 47 47
  vcan0  604   [8]  48 48 48 48 48 48 48 48
```

The above displays the CAN message going over to the vcan0.

One of the examples of receiving CAN messages is described below. Download here ()

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <linux/can.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <string.h>
#include <unistd.h>
#include <linux/can/raw.h>
```

```c
int main(int argc, char **argv)
{
    int fd;
    struct sockaddr_can can;
    struct can_frame canframe;
    struct ifreq ifr;
    int ret;

    fd = socket(AF_CAN, SOCK_RAW, CAN_RAW);
    if (fd < 0) {
        return -1;
    }

    strcpy(ifr.ifr_name, "vcan0");
    ret = ioctl(fd, SIOCGIFINDEX, &ifr);
    if (ret < 0) {
        return -1;
    }

    can.can_family = AF_CAN;
    can.can_ifindex = ifr.ifr_ifindex;

    ret = bind(fd, (struct sockaddr *)&can, sizeof(can));
    if (ret < 0) {
        return -1;
    }

#if 0
    canframe.can_id = 0x600;
    canframe.can_dlc = 8;
    canframe.data[0] = 0x44;
    canframe.data[1] = 0x44;
    canframe.data[2] = 0x44;
    canframe.data[3] = 0x44;
    canframe.data[4] = 0x44;
    canframe.data[5] = 0x44;
    canframe.data[6] = 0x44;
    canframe.data[7] = 0x44;
#endif

    struct can_filter f[2];

    f[0].can_id = 0x600;
    f[0].can_mask = CAN_SFF_MASK;

    f[1].can_id = 0x601;
    f[1].can_mask = CAN_SFF_MASK;

    ret = setsockopt(fd, SOL_CAN_RAW, CAN_RAW_FILTER, &f, sizeof(f));
    if (ret < 0) {
        return -1;
```

```
    }

    while (1) {
        sleep(1);

        ret = read(fd, (void *)&canframe, sizeof(canframe));
        if (ret < 0) {
            return -1;
        }

        printf("%02x %02x ", canframe.can_id, canframe.can_dlc);
        int i;

        for (i = 0; i < canframe.can_dlc; i ++) {
            printf("%02x:", canframe.data[i]);
        }
        printf("\n");
    }

    return 0;
}
```

The above code has the setup code similar to the `write` on the CAN, except the filters and the read calls.

to selectively filter out the CAN frames at the network layer, the `SOL_CAN_RAW` socket option is used. Below is one example code snippet.

```
struct can_filter f[2];

f[0].can_id = 0x600;
f[0].can_mask = CAN_SFF_MASK; // standard CAN frame only

f[1].can_id = 0x601;
f[1].can_mask = CAN_SFF_MASK;

ret = setsockopt(fd, SOL_CAN_RAW, CAN_RAW_FILTER, &f, sizeof(f));
if (ret < 0) {
    return -1;
}
```

The above example use a structure `struct can_filter`. It contains the below format,

```
struct can_filter {
    can_id_t can_id;
    can_id_t can_mask;
};
```

the `can_id` is the ID of the frame that the program wants to receive and the mask can be one of `CAN_SFF_MASK` or `CAN_EFF_MASK`.

if the standard CAN messages to be received then `CAN_SFF_MASK` is used, and if the extended CAN messages are to be received then `CAN_EFF_MASK` is used.

the `setsockopt` option `SOL_CAN_RAW` accepts a `CAN_RAW_FILTER` with a filter of type `struct can_filter`. This can be an array and set to the `setsockopt` as described above.

The `read` system call above reads each CAN frame and dumps them on the screen, only the DLC `can_dlc` bytes are set in the `data` portion and the rest must be treated as not set or invalid while processing the CAN data.

The structure `struct can_frame` is filled with `can_id` `can_dlc` and `data` members upon a `recvfrom` call when given as argument to `recvfrom`.

Below is one of the example of such case.

```
struct sockaddr_can can;
struct ifreq ifr;
socklen_t len = sizeof(can);
struct can_frame f;

ret = recvfrom(fd, &f, sizeof(f), 0, (struct sockaddr *)&can, &len);
if (ret < 0) {
    return -1;
}

if (ret != sizeof(f)) {
    return -1;
}

ifr.ifr_ifindex = can.can_ifindex;
ret = ioctl(fd, SIOCGIFNAME, &ifr);
if (ret < 0) {
    return -1;
}

printf("CAN ifname %s\n", ifr.ifr_name);
```

CAN broadcast manager (BCM)

SQL

SQL is a systems query language and is a database thats very popular and widely used.

1. CREATE TABLE celebs (id INTEGER, name TEXT, age INTEGER);

Creates a new Table entry

2. INSERT INTO celebs (id, name, age) VALUES(1, 'Jason Statham', 40);
3. SELECT * FROM celebs

Appendix. D Gcc compilation flags

The following options are useful while using the gcc.

option	brief description
-Wall	enable all warnings
-Werror	enable all errors
-Os	enable optimistaion
-Wformat	enable format of the arguments printf, scanf etc..
-pipe	write to pipe instaed of file for faster compilation
-Wimplicit-function-declaration	warn if function is not declared
-Wshadow	enable detection of shadow variables
`_FORTIFY_SOURCE	enable security options and insert memory fences on final binary
-I	include files path
-g	enable debug symbols
-ggdb	enable gdb specific debug symbols
-L	library files path
-l	link with the library
-fstack-usage	generate stack usage for each function that is in the source file
-Wstack-usage	generate a warning if the stack usage exceeds some given bytes as argument to -Wstack-usage
-fPIC	enable position independent object file compilation
-fPIE	enable position independent exeutable file compilation

some of the above options are detailed below..

1. Wall to get all sorts of warnings related to the C code. This will also perform a code review (basic style) and catches bugs that occur at beginning or intermediate stages of the testing process. the code review is done mostly the static code review. This catches undefined varibles, unused variables and places where variables are used without initialisation or assignment. When compiled with -Werror it forces the programmer to fix all the warnings. However some or more of the options may not be really needed fixing. some things such as unused fuction arguments can be disabled with -Wno-unused-parameter option.

2. Werror forces the programmer to fix the warnings generated.

3. On is an optimisation flag that is useful in the code optimisation at the compilation stage. it goes with 1, 2, 3 or s. where each number specify the optimisation level. if given s, it means that the compiler can choose what optimisation can be applied. This is better than programmer choosing what optimisation better for the program. Sometimes, the optimisation level higher does not mean the generated code be high performance. So leaving this option to the compiler is better.

At each optimisation levels, gcc try to become more smarter. for example, if the loops are present in the program, gcc figures if the value of the final count in a for loop can be predicted. Such as the following.

```
uint8_t mac[] = {0x00, 0x01, 0x01, 0x02, 0x02, 0x02};
int i;

for (i = 0; i < sizeof(mac); i ++) {
    printf("%x:", mac[i]);
}
printf("\n");
```

gcc sees that the loop above can be optimised by simply replacing the `for` loop with 6 printf statements than extra jump statements that may stall pipeline in the CPU or add up more instructions.

4. `pipe` is used to faster the process of compilation. The `pipe` option makes the compiler to write to a pipe instead of writing to temporary files. The `pipe` is then used as a means of communication to the other programs in the subsequent sections of the compilation.
5. `Wimplicit-function-declaration` is used to find places where the function is not declared . This is the very important as in which the `C` language takes the function declaration as `int function(void)` if its not declared. In cases where `double` is returned by a function and the function is not declared anywhere, the returned value becomes `int`. This becomes a serious bug when the function tries to estimate the distance between two co-ordinates.
6. `I` is used to provide the compiler a path to look for the include files that the program uses.
7. `g` is the option that enables the debugging in the code. It adds needed symbols for the `gdb` for the debugging purposes and to properly record coredumps and also for the `valgrind` program to correctly locate the faulty line in the code for memory leaks / invalid memory accesses.

The `-fstack-usage` option generates a stack usage file with `.su` extension. the source file when given `-fstack-usage` as option to `gcc` or `g++`. Knowing the stack usage is really important when programming very small embedded hardware with limited resources.

The `-Wstack-usage` warns the user when the stack usage exceeds a particular given limit as option to `-Wstack-usage`. The `-Wstack-usage=2048` mean that the warning is produced if a function uses the stack over the size 2k bytes.

The `_FORTIFY_SOURCE` is a macro option to be used with `-D` just like for an `#ifdef` to be defined at the compile time.

The `_FORTIFY_SOURCE` has some of the very nice features when it comes to security. It is to be used with the arguments 1 or 2. When given this option, the optimisation flag `-O` with levels 2 or 3 must be enabled otherwise the `_FORTIFY_` will not compile and dumps errors.

This option adds string safety mechanism when used against the libc string library. If the size of the string is deterministic at the compile time, then it can predict if any buffer overrun and underrun that can cause process access undefined memory.

If such a possibility cannot be predicted, it adds the fences at each buffer boundary to the final binary. When a buffer overrun or underrun is being detected, the program simply crashes and dumps the debug on the console. This lets the program not run when there is such issue.

without the FORTIFY option, the program may silently execute (or may possible crash) the code beyond the given memory location allowing for the exploit to execute arbitrary code.

Gcov for code coverage:

to compile with test coverage (to use with gcov), the following options need to be also present with gcc.

`-fprofile-arcs -ftest-coverage -pg -lgcov`

The gcov then adds in the profile code in the object file generated, when executed, the binary generates the profiler output files `.gcno` and `.gcna` for further analysis with gcov.

LDD:

ldd program print any shared object dependencies on the file exe.

example:

```
ldd loggerTest
    linux-vdso.so.1 =>  (0x00007ffe71305000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9774539000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f9774903000)
```

all the above `.so` files are shared objects from standard libc.

nm:

nm command lists the symbols in the output object file or an executable. simply running nm command on the object or executable dumps out the symbol information.

For example: compile the program `pipes.c` in the repo as

```
gcc pipes.c
```

will produce an `a.out` binary. running nm on `a.out` as,

```
nm a.out
```

will produce the following symbol list.

```
0000000000601078 B __bss_start
                 U close@@GLIBC_2.2.5
0000000000601088 b completed.7594
0000000000601068 D __data_start
0000000000601068 W data_start
0000000000400700 t deregister_tm_clones
0000000000400780 t __do_global_dtors_aux
0000000000600e18 t __do_global_dtors_aux_fini_array_entry
0000000000601070 D __dso_handle
0000000000600e28 d _DYNAMIC
0000000000601078 D _edata
0000000000601090 B _end
                 U exit@@GLIBC_2.2.5
0000000000400944 T _fini
                 U fork@@GLIBC_2.2.5
00000000004007a0 t frame_dummy
0000000000600e10 t __frame_dummy_init_array_entry
0000000000400aa0 r __FRAME_END__
                 U fwrite@@GLIBC_2.2.5
0000000000601000 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000400978 r __GNU_EH_FRAME_HDR
00000000004005e8 T _init
0000000000600e18 t __init_array_end
0000000000600e10 t __init_array_start
0000000000400950 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000600e20 d __JCR_END__
0000000000600e20 d __JCR_LIST__
                 w _Jv_RegisterClasses
0000000000400940 T __libc_csu_fini
00000000004008d0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000004007c6 T main
                 U pipe@@GLIBC_2.2.5
                 U printf@@GLIBC_2.2.5
                 U read@@GLIBC_2.2.5
0000000000400740 t register_tm_clones
                 U __stack_chk_fail@@GLIBC_2.4
00000000004006d0 T _start
0000000000601080 B stderr@@GLIBC_2.2.5
0000000000601078 D __TMC_END__
                 U write@@GLIBC_2.2.5
```

alternatively `nm` can be run on the object files.

the `nm` gives some of the symbols with letters such as u U B b T or t etc..

symbol	**meaning**
u	unique symbol

U	undefined symbol
T t	text section
B b	uninitialised data section
D d	initialised data section
R r	read-only section

to generate an object file, use

```
gcc -c pipes.c
```

this generates an object file pipes.o. running nm on it is same as that of the final binary file.

```
nm pipes.o
```

would produce less debug symbols because the object is not linked against the libraries. The nm output is as follows.

```
                 U close
                 U exit
                 U fork
                 U fwrite
0000000000000000 T main
                 U pipe
                 U printf
                 U read
                 U __stack_chk_fail
                 U stderr
                 U write
```

what nm actually does is that it reads the debug symbols to produce the references. To cause no debug symbols the final executable can be compiled with the strip option -s. This option is only valid for the final executable but not with the obj files.

running with the strip option produces no output from nm.

```
gcc -s pipes.c
nm a.out

nm: a.out: no symbols
```

with -g gcc option, nm would produce even more dbeug symbols.

some of the useful arguments of nm are the following.

argument	**meaning**
-a	display all the symbols

ar:

static linking the object files is done with the ar command.

```
ar rcv libabc.a obj.o obj1.o  ..
```

above example is generally used most commonly.

readelf:

displays information about the elf files.

creating shared object:

objdump:

Appendix. E Programming problems

This appendix details about some of the programming problems for practise.

processes / fork

file operations / directory manip

pthreads

message queues

Appendix. F Shell

The `#!` is called **shebang**. the next followed by the shebang is the interpreter used. usually the interpreter used are the shells such as `/bin/sh` or `/bin/bash`.

printing on the shell is as easy as using `echo` command.

The `echo` command prints text on the shell when given as

```
echo "hello"
```

prints "hello" on the shell.

Below is the shellscript. Download [here (https://github.com/DevNaga/gists/blob/master/echo.sh)]

```
#!/bin/bash

echo "hello"
```

conditional statements:

conditional statements such as `if` `elif` and `else` exist in shell.

```
if [ -f fork.c ] ; then
    echo "file present"
fi
```

looping statements:

switch:

some of the useful commands:

Below are some of the useful commands when using the shell.

command	description	
echo	echo some text on the shell	
`	`	pipe the output of one command to another command
grep	search for a keyword or text or pattern	
which	locate the program in standard paths	
chmod	change the file mode permissions	