Contents

Linux Systems Programming with C and C++

This book explains details about the systems programming language in C and C++.

This is a newer version of the book. For C and C++ programming refer here.

Author: Devendra Naga

Revision History

Update : 28/7/2023

- Consolidate all md into one md file.
- Remove the C programming language documentation.
- Generate book.pdf

Introduction

Systems programming deals about writing systems software that utilizes the software and hardware services offered by the OS and Hardware.

This book mainly explains about the Lnux userspace. This book does not explain or does not go much detail into the kernel space. Although, some relative concepts are explained.

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)

```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

```
#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

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

Will simply execute the `ls` command.

the `system` system call prototype is as follows,

```
int system(const char *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 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[]);
```

the `execl` function executes the program by its `path` and given `arg` set. The `execl` must be terminated with `NULL`.

Below is an example of `execl` system call. Download here

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

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

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

    pid = fork();
    if (pid != 0) {
        ret = execl(argv[1], argv[1], argv[2], NULL);
        if (ret < 0) {
            fprintf(stderr, "failed to execl %s\n", strerror(errno));
            return -1;
        }
    }
}
```

compile the above program as `gcc -Wall execl.c` and run it as follows.

```
./a.out /bin/ls -l
```

the above command would produce a list of contents with in the directory that the `a.out` has run. Note the `ls` command is given its path. without giving the path, the `execl` fails. Try it like the below command,

```
./a.out ls -l
```

this means that the `execl` always require a full path of the program to execute.

similar to the `execl` the `execlp` accepts the filename that is the program name. The path is not required as long as it stays with in the standard directory (as in `PATH` variable current executable directory).

Below is an example of the `execlp`. Download here

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

int main(int argc, char **argv)
{
```

```
    int ret;
    pid_t pid;

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

    pid = fork();
    if (pid != 0) {
        ret = execlp(argv[1], argv[1], argv[2], NULL);
        if (ret < 0) {
            fprintf(stderr, "failed to execl %s\n", strerror(errno));
            return -1;
        }
    }
}
```

compile the above program as `gcc -Wall execlp.c` and run it as below,

`./a.out ls -l`

it does execute the `ls -l` command and prints the contents on the screen.

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

```
#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

```
#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

```
#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

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

int main()
```

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

    return 0;
}
```

Below program describe getting the `sysconf`data from all the parameters. Download here

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

struct sysconf_params {
    char *name;
    int param;
    int val;
};

int main()
{
    struct sysconf_params sysconf_args[] = {
        {"_SC_ARG_MAX", _SC_ARG_MAX, 0},
        {"_SC_OPEN_MAX", _SC_OPEN_MAX, 0},
        {"_SC_STREAM_MAX", _SC_STREAM_MAX, 0},
        {"_SC_TZNAME_MAX", _SC_TZNAME_MAX, 0},
        {"_SC_NGROUPS_MAX", _SC_NGROUPS_MAX, 0},
        {"_SC_JOB_CONTROL", _SC_JOB_CONTROL, 0},
        {"_SC_SAVED_IDS", _SC_SAVED_IDS, 0},
        {"_SC_VERSION", _SC_VERSION, 0},
        {"_SC_CLK_TCK", _SC_CLK_TCK, 0},
        {"_SC_CHARCLASS_NAME_MAX", _SC_CHARCLASS_NAME_MAX, 0},
        {"_SC_REALTIME_SIGNALS", _SC_REALTIME_SIGNALS, 0},
        {"_SC_PRIORITY_SCHEDULING", _SC_PRIORITY_SCHEDULING, 0},
        {"_SC_TIMERS", _SC_TIMERS, 0},
        {"_SC_ASYNCHRONOUS_IO", _SC_ASYNCHRONOUS_IO, 0},
        {"_SC_PRIORITIZED_IO", _SC_PRIORITIZED_IO, 0},
        {"_SC_MEMLOCK", _SC_MEMLOCK, 0},
        {"_SC_MEMLOCK_RANGE", _SC_MEMLOCK_RANGE, 0},
        {"_SC_SEMAPHORES", _SC_SEMAPHORES, 0},
        {"_SC_MQ_OPEN_MAX", _SC_MQ_OPEN_MAX, 0},
        {"_SC_MQ_PRIO_MAX", _SC_MQ_PRIO_MAX, 0},
        {"_SC_SEM_NSEMS_MAX", _SC_SEM_NSEMS_MAX, 0},
        {"_SC_SEM_VALUE_MAX", _SC_SEM_VALUE_MAX, 0},
        {"_SC_TIMER_MAX", _SC_TIMER_MAX, 0},
        {"_SC_LOGIN_NAME_MAX", _SC_LOGIN_NAME_MAX, 0},
        {"_SC_THREAD_STACK_MIN", _SC_THREAD_STACK_MIN, 0},
        {"_SC_THREAD_THREADS_MAX", _SC_THREAD_THREADS_MAX, 0},
```

```
        {"_SC_LINE_MAX", _SC_LINE_MAX, 0},
        {"_SC_PAGESIZE", _SC_PAGESIZE, 0},
    };
    int i;

    for (i = 0; i < sizeof(sysconf_args) / sizeof(sysconf_args[0]); i ++) {
        sysconf_args[i].val = sysconf(sysconf_args[i].param);
    }

    for (i = 0; i < sizeof(sysconf_args) / sizeof(sysconf_args[0]); i ++) {
        printf("sysconf(%s) \t : %d\n", sysconf_args[i].name, sysconf_args[i].val);
    }

}
```

to summarise, the following `sysconf` variables 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

Resource name	description
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.

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

static void _get_rlimit(char *data, int flag)
{
    struct rlimit rlim;
    int ret;

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

    printf("[item %s]: soft: %lu hard: %lu\n", data, rlim.rlim_cur, rlim.rlim_max);
}
```

```c
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;
    int flag;
    void (*get_callback)(char *, int);
    void (*set_callback)(int n, char **rem_args);
} rlimit_list[] = {
    {"max_addr_space", RLIMIT_AS, _get_rlimit, NULL},
    {"max_file_size", RLIMIT_FSIZE, _get_rlimit, NULL},
    {"max_stack_size", RLIMIT_STACK, _get_rlimit, set_max_stack_size},
    {"max_cpu_time", RLIMIT_CPU, _get_rlimit, NULL},
    {"max_data_size", RLIMIT_DATA, _get_rlimit, NULL},
    {"max_core_size", RLIMIT_CORE, _get_rlimit, NULL},
    {"max_process", RLIMIT_NPROC, _get_rlimit, NULL},
    {"max_files", RLIMIT_NOFILE, _get_rlimit, NULL},
    {"max_memlock", RLIMIT_MEMLOCK, _get_rlimit, NULL},
    {"max_locks", RLIMIT_LOCKS, _get_rlimit, NULL},
    {"max_sigpending", RLIMIT_SIGPENDING, _get_rlimit, NULL},
    {"max_msgqueue", RLIMIT_MSGQUEUE, _get_rlimit, NULL},
    {"max_nice", RLIMIT_NICE, _get_rlimit, NULL},
    {"max_rtprio", RLIMIT_RTPRIO, _get_rlimit, NULL},
    {"max_rt_timeout", RLIMIT_RTTIME, _get_rlimit, 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 ++) {
            rlimit_list[i].get_callback(rlimit_list[i].string, rlimit_list[i].flag);
        }
    } 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). 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)

```
#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
```

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.

error code	description
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)

```
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

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

```c
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

```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$
```

3. Below is an example that demos the `EADDRINUSE`.

Download here

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int fd;
    int ret;

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    struct sockaddr_in addr;

    addr.sin_addr.s_addr = INADDR_ANY;
```

```
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4444);

    ret = bind(fd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret < 0) {
        fprintf(stderr, "failed to bind : %s\n", strerror(errno));
        close(fd);
        return -1;
    }

    while (1) {
        char buf[100];

        ret = recvfrom(fd, buf, sizeof(buf), 0, NULL, NULL);
        if (ret < 0) {
            break;
        }
    }

    close(fd);

}
```

compile and run the above program on one console. Run it again on the other console, observe the output of the failure in `bind` system call.

Third Chapter

Contents

Memory layout of a program

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. Kernel calls the `/sbin/init` soon after the kernel initialization is successful. If the init process dies, the system brings down to shutdown / halt. Init manages the system and starts the rest of the processes within the system.

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

The `setsid` system call creates a new session if the calling process is not the process group leader. When called, the calling process becomes the process group leader. This means that the session id will be same as that of the process id.

Generally `setsid` is called right after the call to `fork` system call in the child process.

`setsid` is mainly used when daemonizing a process.

Daemonize

Daemons or system daemons detach from the controlling terminal and have the parent as the init process. Daemons always run in the background.

Below are few steps to make a process, a daemon.

1. `fork` and exit parent.
2. call `setsid` in child process.
3. move `stdin`, `stdout` and `stderr` to `/dev/null`.
4. change the working directory to `/`.

Below is an example of such in C.

```
int daemonize()
{
    pid_t pid;
```

```
    int ret;

    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "failed to fork\n");
        return -1;
    } else if (pid > 0) {
        exit(0); // parent process exits
    }

    ret = setsid();
    if (ret < 0) {
        fprintf(stderr, "failed to setsid\n");
        return -1;
    }

    int fd;

    fd = open("/dev/null", O_RDWR);
    if (fd >= 0) {
        dup2(fd, 0); // 0 is stdin
        dup2(fd, 1); // 1 is stdout
        dup2(fd, 2); // 2 is stderr

        close(fd);
    }

    ret = chdir("/");
    if (ret < 0) {
        fprintf(stderr, "failed to chdir\n");
        return -1;
    }

    return 0;
}
```

getpriority and setpriority system calls

scheduling system calls

Linux defines the following Scheduling system calls.

Scheduling system call	description

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

```
#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..

```
#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

The prototype of `putenv` is as follow,

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

Below is an example of `putenv`.

```
#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.

```
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

```
#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;
}
```

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

```
#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

```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:

```
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

```
#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 t
        //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

```
#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:

1. 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.
2. Parent then sets the read end of the file descriptor to the `fd_set` for each children.
3. Parent waits in `select` system call to monitor the `fd_set`.
4. 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

```
#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;
}
```

the above code, creates 10 child processes. Each by calling `fork` system call. Each child closes the read end and parent closes the write end. The read end of the parent process is set to the `fd_set`. The system call `select` is then used to wait for any data on the read end of the pipes. If a child process die, the pipe close triggers a read of 0 bytes, causing `select` to return. The `waitpid` is then called to reap the child process. Each child that is dead, is then cleared from the `fd_set`.

[self pipe trick]

signals

Introduction

Signals are used to notify a process about some event. Signals are asynchornous events. 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 - SIG_IGN (ignore the signal by specifying this)
2. perform the handler function execution - create a callback handler that gets called when signal occurs
3. default action - is default for each process when created - kernel creates a default handler for each process

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);
```

The first argument is the signal number. The second argument is the signal handler callback.

An example of system call is shown below.

```
void signal_callback(int sig)
{
    printf("signal handler\n);
}

int main()
{
    signal(SIGINT, signal_callback);

    sleep(2);
}
```

above program registers `SIGINT` that's `ctrl + c` combination. A `ctrl + c` key combination is pressed when the program is running, it invokes the `signal_callback`.

to set the signal to default we must use `SIG_DFL` in the callback argument. Such as,

```
signal(SIGINT, SIG_DFL);
```

Sigaction The `sigaction` system call is more sophisticated system call for signal handling. The prototype of `sigaction` system call is as follows,

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

The structure `sigaction` is shown as below,

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

The first argument is the signal number. The second argument is the structure `sigaction`.

The `sa_flags` contain the following information. It provides a various set of flags but only the following list is useful.

1. `SA_SIGINFO` - the callback `sa_sigaction` is used when flags are set to this.
2. 0 - a default `sa_handler` should be set

Key differences A very good stackoverflow question 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

The system call `sigwait` is used to wait for signal until one of the signals specified in the signal mask are pending.

```
int sigwait(const sigset_t *set, int *sig);
```

the second argument `sig` contain the returned signal number.

The `sigwait` returns 0 on success and returns a positive error on failure.

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

```
#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.

usually, the `sigaddset` is called this way:

```
sigset_t set;

sigaddset(&set, SIGINT); // ignore SIGINT
```

ignores the signal `SIGINT`.

To setup a group of signals, the `sigaddset` can be allowed to call in a loop. For example,

```
int i = 0;
int signal_list[] = {SIGINT, SIGQUIT, SIGALRM};
sigset_t set;

// setup signal mask for the signals in signal_list
while (i < sizeof(signal_list) / sizeof(signal_list[0])) {
    sigaddset(&set, signal_list[i]);
}
```

sigfillset

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

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

sigemptyset

`sigemptyset` clears the signal mask. The initialization of the set of type `sigset_t` is usually done with the `sigemptyset`. Before any calls to `sigaddset` the set of type `sigset_t` must be cleared with `sigemptyset`.

The above examples calls of `sigaddset` must use `sigemptyset`. So the fixed code examples look like the below,

```
sigset_t set;
```

```
sigemptyset(&set); // clear the signal set

sigaddset(&set, SIGINT); // ignore SIGINT


int i = 0;
int signal_list[] = {SIGINT, SIGQUIT, SIGALRM};
sigset_t set;

sigemptyset(&set); // clear the signal set

// setup signal mask for the signals in signal_list
while (i < sizeof(signal_list) / sizeof(signal_list[0])) {
    sigaddset(&set, signal_list[i]);
}
```

sigismember

`sigismember` validates if the given signal is with in the set. Prototype is as follows,

```
int sigismember(sigset_t *set, int no);
```

usually, the calling example looks like the following way,

```
sigismember(&set, SIGALRM);
```

below is one of the examples of using both `sigfillset` and `sigismember`.

```
/**
 * sigismember and sigfillset example
 *
 * Author: Devendra Naga (devendra.aaru@gmail.com)
 *
 * LICENSE MIT
 */

#include <stdio.h>
#include <signal.h>

int main()
{
    sigset_t set;

    sigfillset(&set);
```

```
    if (sigismember(&set, SIGALRM)) {
        printf("sigalrm is a member of the set\n");
    } else {
        printf("sigalrm is not a member of the set\n");
    }
}
```

sigdelset

`sigdelset` deletes the signal from the given set. The prototype is as follows,

```
int sigdelset(sigset_t *set, int no);
```

sigprocmask

The system call `sigprocmask`, used to block certain signals.

The `sigprocmask` prototype is as follows (from the man pages),

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

`how` is defined by one of the following, `SIG_BLOCK` and `SIG_UNBLOCK`.

Below is an example use of `sigprocmask`.

```
/**
 * example sigprocmask
 *
 * Author: Devendra Naga (devendra.aaru@gmail.com)
 *
 * LICENSE MIT
 */
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main()
{
    sigset_t set;
    int ret;

    sigemptyset(&set);

    // add SIGINT
    sigaddset(&set, SIGINT);
```

```
    ret = sigprocmask(SIG_BLOCK, &set, NULL);
    if (ret != 0) {
        perror("sigprocmask");
        return -1;
    }

    int count = 0;

    while (1) {
        printf("hello .. %d\n", count ++);
        sleep(1);
        if (count == 5) {
            break;
        }
    }

    ret = sigprocmask(SIG_UNBLOCK, &set, NULL);
    if (ret != 0) {
        perror("sigprocmask");
        return -1;
    }
}
```

In the above example, the signal set of type `sigset_t` is created with `sigemptyset` followed by the `sigaddset`. The signal `SIGINT` being setup in the signal set.

The `sigprocmask` system call is made with `SIG_BLOCK` that effectively blocks the signal `SIGINT` till the loop below executes. The loop is created only for testing purposes to see if the `SIGINT` is actually blocked by holding down `ctrl +c` combination on the keyboard. Till the count of 5, the signal is blocked and a call to the `sigprocmask` with `SIG_UNBLOCK` is made to unblock the signal.

The use case of `sigprocmask` is useful when we have multiple threads and we need to handle signals specifically in one of the threads / main thread.

since the signals are inherited by the threads / child processes, the signal might be delivered to any thread.

To do this, block signals in all the threads except the thread that signal needs to be handled.

for example, a below code can be used.

```
static void block_term_signals()
{
    sigset_t set;
```

```
    int ret;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);

    ret = sigprocmask(SIG_BLOCK, &set, NULL);
    if (ret < 0) {
        return;
    }
}
```

Simple demonstration of this is as follows:

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

static void block_term_signals()
{
    sigset_t set;
    int ret;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);

    ret = sigprocmask(SIG_BLOCK, &set, NULL);
    if (ret < 0) {
        return;
    }
}

void *thread_callback(void *data)
{
    block_term_signals();

    while (1) {
        sleep(1);
        printf("in %d %s\n", __LINE__, __func__);
    }
}

void *thread_callback_(void *data)
{
    block_term_signals();
```

```
    while (1) {
        sleep(1);
        printf("in %d %s\n", __LINE__, __func__);
    }
}

void signal_handler(int sig)
{
    printf("in signal handler\n");
    signal(sig, SIG_DFL);
    raise(sig);
}

int main()
{
    pthread_t tid1, tid2;
    sigset_t set;
    int ret;

    signal(SIGINT, signal_handler);

    ret = pthread_create(&tid1, NULL, thread_callback, NULL);
    if (ret < 0) {
        return -1;
    }

    ret = pthread_create(&tid2, NULL, thread_callback_, NULL);
    if (ret < 0) {
        return -1;
    }

    while (1) {
        printf("in main thread\n");
        sleep(1);
    }
}
```

in the program, the main program creates two threads and in each thread the signals are blocked except in main thread. main thread registers the signal handler for the particular signal.

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

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

sigchld handling

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;
}
```

Pipes and Fifos

Pipe

A `pipe` is a method to connect an output of one process to the input of another process. The `pipe` provides a one way communication between the processes.

Example:

```
ls | sort
```

The above `|` is called a pipe that pipes the output of ls to the input of sort.

The `pipe` system call is used to create a pipe in C.

The following is the prototype of the function.

```
int pipe(int pipefd[2]);
```

The `pipefd[0]` is the read end of the pipe and `pipefd[1]` is the write end of the pipe.

On success 0 is returned from the function and on failure -1.

include header file `unistd.h` for the `pipe` system call.

Here is the below example describing the usage of `pipe` system call. You can also download the program here

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

int main()
{
    int pipefd[2];
    int ret;

    ret = pipe(pipefd);
    if (ret < 0) {
        fprintf(stderr, "failed to create a pipe\n");
        return -1;
    }

    pid_t child_pid;
```

```
    child_pid = fork();
    if (child_pid == 0) {
        char msg[100];

        close(pipefd[1]);
        read(pipefd[0], msg, sizeof(msg));
        printf("read %s\n", msg);
        exit(1);
    } else {
        char msg[] = "Hello ";

        close(pipefd[0]);
        write(pipefd[1], msg, sizeof(msg));
        exit(1);
    }

    return 0;
}
```

Fifo

A fifo is a named pipe and works same as the `pipe`.

One way to create a fifo is the following

```
mknod fifo_file p
mfifo a=rw fifo_file
```

`mknod` creates a block or special character files. The `p` option is used to create a fifo.

`mkfifo` creates a fifo with the given name. The `a` option is used to open the file in `rw` mode.

There are `mknod` and `mkfifo` system calls and are defined below.

```
int mknod(const char *path, mode_t mode, dev_t dev);
int mkfifo(const char *path, mode_t mode);
```

Example of the calls are shown as below

```
mknod("/home/devnaga/fifo_file", S_IFIFO | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP, 0);
mkfifo("/home/devnaga/fifo_file", S_IFIFO | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
```

sometimes, `S_IFIFO` is omitted.

once a fifo is created an `open` call is performed on the fifo file at either ends. At the creator end, the call shall be made with `O_CREAT` option.

the code example that does it is as follows,

```
int ret;
int fd;

ret = mkfifo ("/fifo_test", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
if (ret < 0) {
    return -1;
}

fd = open("/fifo_test", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
if (fd < 0) {
    return -1;
}
```

Below is an example of a program that use fifo for the communication. Download here

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

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

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

    int fd;

    if (!strcmp(argv[1], "create")) {
        unlink("/fifo_test");

        ret = mkfifo("/fifo_test", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
        if (ret < 0) {
            return -1;
        }

        fd = open("/fifo_test", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
        if (fd < 0) {
            return -1;
        }
```

```
        while (1) {
            char *sender_msg = "Hello from sender";
            write(fd, sender_msg, strlen(sender_msg) + 1);

            sleep(1);
        }
    } else if (!strcmp(argv[1], "open")) {
        fd = open("/fifo_test", O_RDWR);
        if (fd < 0) {
            return -1;
        }

        while (1) {
            char rxmsg[200];

            ret = read(fd, rxmsg, sizeof(rxmsg));
            if (ret <= 0) {
                break;
            }

            printf("msg fromsender: %s\n", rxmsg);
        }
    }

    return 0;
}
```

The above program creates a fifo in `create` portion of the code and opens it in read write, the arguments "create / open" are passed via the command line. In `open` mode, the fifo is simply opened for read and write.

run the above program in two separate terminals and observe the output. ## 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.

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

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

```c
#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

```
#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, sockl
```

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 *c
```

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

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

```c
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>
#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

```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.

```
#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.

```
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.

```
#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.

```
#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 and Client

```
#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>

#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:

```
#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

```
#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;
}
```

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.

The below are the error numbers to return by the APIs that we are about to write.

```
typedef enum {
    TCP_SOCK_OPEN_SUCCESS = 0,
    TCP_SOCK_OPEN_ERROR_INVAL_PROTO,
    TCP_SOCK_BIND_ERROR_SERVER_ALREADY_BOUND,
    TCP_SOCK_BIND_ERROR_INVAL_AF,
    TCP_SOCK_CONN_ERROR_INVAL_CONN,
    TCP_SOCK_ACCEPT_ERROR_INVAL_CLIENT,
    TCP_SOCK_SEND_ERROR_SOCK_CLOSED,
    TCP_SOCK_SEND_ERROR_SOCK_INVAL,
    UDP_SOCK_OPEN_ERROR_INVAL_PROTO,
    UDP_SOCK_BIND_ERROR_SERVER_ALREADY_BOUND,
    UDP_SOCK_BIND_ERROR_INVAL_AF,
```

```
    UDP_SOCK_CONN_ERROR_INVAL_CONN,
    UDP_SOCK_SEND_ERROR_SOCK_CLOSED,
    UDP_SOCK_SEND_ERROR_SOCK_INVAL,
} socketlib_ret_t;

// default connections to the listen function
#define SOCKET_LIB_MAX_CONN 100

1. tcp_socket_create_client(char *ip_port);
2. tcp_socket_create_server(char *ip_port, tcp_socket_accept_conn_cb *);
3. tcp_socket_destroy(int sock);
4. tcp_socket_accept_conn_cb(int sock, struct sockaddr *remote, socklen_t *remote_len);
5. tcp_socket_send(int sock, void *buf, int len, int opts);
6. tcp_socket_recv(int sock, void *buf, int len, int opts);
7. udp_socket_create_client(char *ip_port, char *to_send_ip_port);
8. udp_socket_create_server(char *ip_port, udp_socket_accept_conn_cb *);
9. udp_socket_destroy(int sock);
10. udp_socket_send(int sock, void *buf, int len, int opts);
11. udp_socket_recv(int sock, void *buf, int len, int opts);
12. udp_socket_sendto(int sock, void *buf, int len, int opts, struct sockaddr *addr, sockler
13. udp_socket_recvfrom(int sock, void *buf, int len, int opts, struct sockaddr *addr, sockl
14. socklib_select(int sock, fd_set *rdfds, struct timeval *timeout);
```

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;
    };
};
```

There is a way to create a dummy interface. Here's one way

```
sudo modprobe dummy
sudo ip link add eth10 type dummy
```

This enables an interface `eth10`, by default interface is not up.

below is the example to get the interface flags.

interface flags

```
#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, 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
```

MTU

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
```

Get Mac Address

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.

```
#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
```

Set Mac Address

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>
```

```c
#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
```

Get Interface Index

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

```
#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>
```

```c
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;
}
```

Set Interface Flags

Below example is another one of `SIOCGIFFLAGS`. Download here

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
```

```c
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

```
#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;
}
```

Get IPv4 Address

The ioctl flag `SIOCGIFADDR` is used to get interface ipv4 address. Below is the example program.

```
#include <stdio.h>
#include <string.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <netinet/in.h>

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

    if (argc != 2) {
        printf("<%s> <ifname>\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));
    ifr.ifr_addr.sa_family = AF_INET;
    strcpy(ifr.ifr_name, argv[1]);

    int ret;

    ret = ioctl(fd, SIOCGIFADDR, &ifr);
    if (ret < 0) {
        printf("failed to ioctl\n");
        return -1;
    }

    char *ip_addr = inet_ntoa(((struct sockaddr_in *)(&ifr.ifr_addr))->sin_addr);
    if (ip_addr)
        printf("ipaddr %s\n", ip_addr);

    return 0;
}
```

The function `if_nametoindex` is declared in `net/if.h`.

Get Broadcast Address

The ioctl `SIOCGIFBRDADDR` allows to get the broadcast address of network interface.

Below is an example. Download here

```
#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>
```

```c
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;
}
```

Get Network Mask

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

```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;
}
```

Get Network Interface List

Below is an example to get the network interface list (that have the ipv4 address).

```
struct interface_info {
    char ifname[24];
    char ifaddr[80];
};

int get_interfaces(std::vector<interface_info> &ifinfo)
{
    struct ifconf ifc;
    struct ifreq *req;
    char data[4096];
    int ret;
    int fd;

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd < 0) {
        return -1;
    }

    ifc.ifc_len = sizeof(data);
    ifc.ifc_buf = (caddr_t)(data);
    ret = ioctl(fd, SIOCGIFCONF, &ifc);
    if (ret < 0) {
        goto err;
    }

    req = (struct ifreq *)data;
    while ((char *)req < data + ifc.ifc_len) {
        switch (req->ifr_addr.sa_family) {
            case AF_INET:
                interface_info ifi;
                char *intf_addr;

                strcpy(ifi.ifname, req->ifr_name);
                intf_addr = inet_ntoa(((struct sockaddr_in *)(&req->ifr_addr))->sin_addr);
                if (!intf_addr) {
                    return -1;
                }
                strcpy(ifi.ifaddr, intf_addr);
```

```
                ifinfo.push_back(ifi);
            break;
        }
        req = (struct ifreq *)((char *)req + sizeof(*req));
    }

    ret = 0;

    close(fd);

err:
    return ret;
}
```

Set Interface Name

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

```
#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.

VLANs

VLANs are used to logically divide the network into many interfaces although there is only one physical interface available. In linux VLANs can be created on with the `vlan` utility or programmatically.

We will look at programmatically creating VLANs here.

VLANs are generally associated with the physical name and an ID. There can be 0 4095 VLAN Ids. The interface must be a real interface in order to create a VLAN.

Below program is used to add a VLAN interface.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if_vlan.h>
#include <linux/sockios.h>

int main(int argc, char **argv)
{
    struct vlan_ioctl_args ioctl_args;
    int sock;
    int ret;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        return -1;
    }

    memset(&ioctl_args, 0, sizeof(ioctl_args));
    strcpy(ioctl_args.device1, argv[1]);
    ioctl_args.u.VID = atoi(argv[2]);
    ioctl_args.cmd = ADD_VLAN_CMD;

    ret = ioctl(sock, SIOCSIFVLAN, &ioctl_args);
    if (ret < 0) {
        perror("ioctl");
        return -1;
    }

    close(sock);
```

```
    return 0;
}
```

running `./a.out enp2s0 4` creates an interface `enp2s0.4`.

```
enp2s0.4: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether 58:8a:5a:0a:6b:2e  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
```

VLANs can be deleted programmatically:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <linux/if_vlan.h>
#include <linux/sockios.h>

int main(int argc, char **argv)
{
    struct vlan_ioctl_args ioctl_args;
    int sock;
    int ret;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        return -1;
    }

    memset(&ioctl_args, 0, sizeof(ioctl_args));
    strcpy(ioctl_args.device1, argv[1]);
    ioctl_args.cmd = DEL_VLAN_CMD;

    ret = ioctl(sock, SIOCSIFVLAN, &ioctl_args);
    if (ret < 0) {
        perror("ioctl");
        return -1;
    }

    close(sock);

    return 0;
}
```

running `./a.out enp2s0.4` deletes the VLAN interface.

wireless ioctls

The header file `linux/wireless.h` has the legacy ioctl calls that are used to communicate with the wireless device.

Below are some of the ioctls listed by the header `linux/wireless.h`

S.No	wireless ioctl name	wireless ioctl description
1	`SIOCGIWNAME`	get wireless AP mode (ieee 802.11abg)
2	`SIOCGIWESSID`	get AP name
3	`SIOCGIWFREQ`	get wireless frequency
4	`SIOCGIWRATE`	get AP rate
5	`SIOCGIWAP`	get AP mac
6	`SIOCGIWNICK`	get nickname

The header file exposes the below data structure,

```
struct iwreq {
    union {
        char ifrn_name[IFNAMSIZ];
    } ifr_ifrn;
    union iwreq_data u;
};
```

1. SIOCGIWNAME:

`SIOCGIWNAME` gets the wireless AP name, in general IEEE 802.11 a/b/g/n. An example of how its used is described below. Download here

```
/**
 * @Author: Devendra NAga (devendra.aaru@gmail.com)
 *
 * Copyright 2014-present All rights reserved
 *
 */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <linux/wireless.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
```

```c
int main()
{
    int sock;
    int ret;

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        return -1;
    }

    struct iwreq req;

    memset(&req, 0, sizeof(req));
    strcpy(req.ifr_name, "wlxd0374533247a");
    ret = ioctl(sock, SIOCGIWNAME, &req);
    if (ret < 0) {
        return -1;
    }

    printf("name: %s\n", req.u.name);

    close(sock);
}
```

output is shown below:

```
name: IEEE 802.11bgn
```

2: SIOCGIWESSID:

`SIOCGIWESSID` gets the network ESSID. Below is an example of getting the ESSID. Download here

```c
/**
 * @Copyright Devendra Naga (devendra.aaru@gmail.com)
 *
 * 2014-present All rights reserved
 *
 */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```c
#include <linux/wireless.h>

int main()
{
    int sock;
    int ret;

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        return -1;
    }

    struct iwreq req;
    char essid[65];

    memset(&req, 0, sizeof(req));
    strcpy(req.ifr_name, "wlxd0374533247a");
    req.u.essid.pointer = essid;
    req.u.essid.length = sizeof(essid);
    ret = ioctl(sock, SIOCGIWESSID, &req);
    if (ret < 0) {
        return -1;
    }

    printf("Essid: %s\n", essid);
}
```

3. SIOCGIWFREQ:

`SIOCGIWFREQ` getes the network frequency. Below is an example of getting the frequency. Download here

```c
/**
 * @Author Devendra Naga (devendra.aaru@gmail.com)
 *
 * @Copyright 2014-present All rights reserved
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <linux/wireless.h>
```

```c
int main()
{
    int sock;
    int ret;

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        return -1;
    }

    struct iwreq req;

    memset(&req, 0, sizeof(req));
    strcpy(req.ifr_name, "wlxd0374533247a");
    ret = ioctl(sock, SIOCGIWFREQ, &req);
    if (ret < 0) {
        return -1;
    }

    printf("freq: %f\n", (double)req.u.freq.m/100000000);
}
```

The package wireless-tools provides the needed API to perform many wireless functions.

Time and timers

Time

Timer

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.

```
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

```
#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

```
#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.

```
char *ctime(const time_t *t);
```

Below is an example of `ctime` API. Download here

```
#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.

```c
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

```c
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`.

```c
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

```
#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.

POSIX.1 timer functions (timer_create, timer_settime)

POSIX.1 provides the following system calls for timers.

1. timer_create
2. timer_delete
3. timer_settime
4. timer_gettime

The `timer_create` creates a timer and sets the timerid.

The `timer_create` prototype is as follows.

```
int timer_create(clockid_t clockid, struct sigevent *sevp, timer_t *timerid);
```

The `clockid_t` must be one of the following.

1. CLOCK_REALTIME
2. CLOCK_MONOTONIC
3. CLOCK_PROCESS_CPUTIME_ID
4. CLOCK_THREAD_CPUTIME_ID
5. CLOCK_BOOTTIME
6. CLOCK_REALTIME_ALARM
7. CLOCK_BOOTTIME_ALARM

The `timer_delete` deletes the timer created by `timer_create`. Its prototype is as follow.

```
int timer_delete(timer_t timerid);
```

A timer can be armed with the `timer_settime`. Its prototype is as follows.

```
int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value, struct iti
```

The first argument is the timerid that is created with above `timer_create`. flags are generally kept 0. `new_value` is set to the timer values.

The `struct itimerspec` is defined as follows,

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};

struct itimerspec {
    struct timerspec it_interval;
    struct timerspec it_value;
};
```

for a one shot timer, the content of `it_interval` is set to 0. For a periodic timer the `it_interval` must be set to a value.

```
struct itimerspec spec;

spec.it_value.tv_sec = 1;
spec.it_value.tv_nsec = 0;
spec.it_interval.tv_sec = 0;
spec.it_interval.tv_nsec = 0;
```

```
struct itimerspec spec;

spec.it_value.tv_sec = 1;
spec.it_value.tv_nsec = 0;
spec.it_interval.tv_sec = 1;
spec.it_interval.tv_nsec = 0;
```

Example using the `timer_create` is shown below. Compile it with `-lrt` linker flag.

```
#include <iostream>
#include <string.h>
#include <string>
#include <signal.h>
#include <time.h>
#include <unistd.h>

// callback function when the timer expires.. this will be called
void timer_handle(int sig, siginfo_t *si, void *uc)
{
    printf("timer handler function\n");
}

int main()
{
    int ret;
```

```
    struct sigevent sev;
    struct itimerspec its;
    struct sigaction sa;
    timer_t timerid;
    sigset_t mask;

    memset(&sev, 0, sizeof(sev));
    memset(&its, 0, sizeof(its));
    memset(&sa, 0, sizeof(sa));

    // set signal mask for the timer
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_handle;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGRTMIN + 1, &sa, NULL) < 0) {
        return -1;
    }

    // set signal mask for sigevent
    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIGRTMIN + 1;
    sev.sigev_value.sival_ptr = &timerid;
    ret = timer_create(CLOCK_REALTIME, &sev, &timerid);
    if (ret < 0) {
        return -1;
    }

    // set timer to tick 1 sec intervals
    its.it_value.tv_sec = 1;
    its.it_value.tv_nsec = 0;
    its.it_interval.tv_sec = 1;
    its.it_interval.tv_nsec = 0;

    ret = timer_settime(timerid, 0, &its, NULL);
    if (ret < 0) {
        return -1;
    }

    // wait indefinitely
    while (1) {
        sleep(1);
    }
}
```

The above function sets up a callback function that gets called everytime a

timer expired. The callback function setup is done at the `sigaction`. The same signal is set to the `struct sigevent` which then is passed as argument to the `timer_create`. So, an expiry would simply invoke the corresponding signal handler behavior (in this case `SA_SIGINFO`) and calls the callback.

There are more than one timer can be created for a process.

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.

```c
#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_use
                }
            }
        }
    }

    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. # Advanced IPC

Message Queues

Shared memory

Semaphores useful links:

[ipcs command]## 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;
struct mq_attr attr;
```

```c
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>
```

```c
#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

```
#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

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

```c
#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.

on a successful `mmap` the pointer to the address of the shared memory is returned, otherwise `MAP_FAILED` is returned. So a check is made on the returned memory for validity.

```
void *maped;

maped = mmap(NULL, 1024 * 1024, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (maped == MAP_FAILED) {
```

```
    return -1;
}
```

when allocating the memory, it is usually best practise to allocate on page boundary. A default page size usually 4kB and may vary from hardware to hardware. check with `sysconf(_SC_PAGE_SIZE)` for runtime page size for each platform.

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

sample code:

```
#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>
```

```c
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
```

`MAP_SHARED` is required on regular files when writing. If doing `MAP_PRIVATE`, the syncs to the underlying file will not happen.

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

```
#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.

Linux provides another shared memory POSIX API called `shm_open` and `shm_unlink` which can be used along with the `mmap` and `munmap`.

The prototype of `shm_open` is as follows,

```
int shm_open(const char *name, int oflag, mode_t mode);
```

the `name` should contain a `/` first and the name of the shared memory file. the `oflag` is same as that of `O_CREAT | O_RDWR` for creation and `O_RDWR` for read-write operation. the `mode` represents the creation mask `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP` etc.

when given as `O_CREAT` to the `oflag` the shared memory will be created. When there are more than one process that is gonna use shared memory, one process would create the memory with `O_CREAT` along with the rest of the oflags and file mode creation flags. The other processes would use the `O_RDWR` `O_RDONLY` and / or `O_WRONLY` flags. When used with out `O_CREAT`, the mode is set to 0.

The `shm_open` returns a file descriptor of the path, and this can be used by the `mmap` as the following,

Prototype of `mmap` is shown below,

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

code to perform `mmap` with the `shm_open` is as follows,

```
int mapfd;

mapfd = shm_open("/shm_path", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
if (mapfd < 0) {
    return -1;
}

ftruncate(mapfd, 1024 * 1024);

void *mapaddr;

mapaddr = mmap(NULL, 1024 * 1024, PROT_READ | PROT_WRITE, MAP_SHARED, mapfd, 0);
if (mapaddr == MAP_FAILED) {
    return -1;
}
```

notice the use of `ftruncate` system call. This is required to let `mmap` work correctly with the given length bytes.

There is no need to `msync` because the operation is on a file descriptor returned by the call to `shm_open`.

the `shm_unlink` prototype is as follows.

```
int shm_unlink(const char *name);
```

this is called right after the program has stopped using the fd returned by the `shm_open`.

Below is an example that demo the use of `mmap` with the `shm_open`. Download here

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

#define SHM_NAME "/test_shm"
#define SHM_BYTES_MAX (1024 * 1024)
#define FIFO_NAME "/test_fifo"

static int fifd;
static int fd;

int intr_reader()
{
    int intr = 1;

    return write(fifd, &intr, sizeof(int));
}

int wait_for_intr()
{
    int intr = 0;
    int ret;

    ret = read(fifd, &intr, sizeof(intr));
    if (ret <= 0) {
        return -1;
    }

    return intr == 1;
}
```

```
void* mmap_create_buf()
{
    fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
    if (fd < 0) {
        perror("shm_open");
        return NULL;
    }

    ftruncate(fd, SHM_BYTES_MAX);

    void *addr;

    addr = mmap(NULL, SHM_BYTES_MAX, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        return NULL;
    }

    return addr;
}

void* mmap_attach_buf()
{
    fd = shm_open(SHM_NAME, O_RDWR, 0);
    if (fd < 0) {
        perror("shm_open");
        return NULL;
    }

    void *addr;

    addr = mmap(NULL, SHM_BYTES_MAX, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        return NULL;
    }

    return addr;
}

int fifo_create()
{
    int ret;

    unlink(FIFO_NAME);
```

```
    ret = mkfifo(FIFO_NAME, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
    if (ret < 0) {
        perror("mkfifo");
        return -1;
    }

    fifd = open(FIFO_NAME, O_RDWR);
    if (fifd < 0) {
        perror("open");
        return -1;
    }

    return 0;
}

int fifo_attach()
{
    fifd = open(FIFO_NAME, O_RDWR);
    if (fifd < 0) {
        perror("open");
        return -1;
    }

    return 0;
}

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

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

    if (!strcmp(argv[1], "producer")) {
        addr = mmap_create_buf();
        if (!addr) {
            return -1;
        }

        ret = fifo_create();
        if (ret < 0) {
            return -1;
        }
```

```
        while (1) {
            strcpy(addr, "Hello ");

            intr_reader();
            sleep(1);
        }
    } else if (!strcmp(argv[1], "consumer")) {
        addr = mmap_attach_buf();
        if (!addr) {
            return -1;
        }

        ret = fifo_attach();
        if (ret < 0) {
            return -1;
        }

        while (1) {
            wait_for_intr();
            printf("data from prod: %s\n", addr);

            memset(addr, 0, 10);
        }
    }
}
```

The above example demonstrates the use of `mmap` and the `shm_open` along with the `mkfifo` for synchronisation.

The creator is simply the producer and the reader is simply the consumer. The creator creates the shared memory fd, maps using mmap and then creates a fifo with `mkfifo` and calls `open`.

The consumer opens the shread memory with the `shm_open` and maps the memory, it also opens the fifo. The consumer never creates any of the shared fd, memory or the fifo. It opens and waits for the data.

The consumer waits on the read calling the `wait_for_intr` call. The producer sleeps every second and writes "Hello" to the shared memory and interrupts the reader via the `intr_reader`.

Compile the program and run it in two separate terminals and observe the communication between the two programs.

Linux defines another system call called `mprotect`, allowing to change the permissions of the existing region of memory.

The prototype of `mprotect` is as follows.

```
int mprotect(const void *addr, size_t len, int prot);
```

where the `addr` is a page aligned memory. The `len` is the portion of the memory to be protected. the `prot` is a combination of `PROT_READ` and `PROT_WRITE`. The `mprotect` overrides the existing protection bits when the memory is created via `mmap`. this means that if the memory needs to be in read only then the `prot` field must be only `PROT_READ`. if its in read write then the combination of OR must be used such as `PROT_READ | PROT_WRITE`.

`mprotect` returns 0 if protection bits are changed success and -1 on failure.

Linux provides a system call `madvise` to provide the kernel an advise. The prototype is as follows,

```
int madvise(void *addr, size_t length, int advise);
```

The `madvise` system call gives the advise to the kernel about the address. The advise is one of the following.

type	meaning
MADV_NORMAL	no special treatment for the address
MADV_RANDOM	expect page references in random
MADV_SEQUENTIAL	expect page references in sequential
MADV_WILLNEED	expect the access in near future

An example call to the `madvise` is as follows.

```
void *addr;

addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (addr == MAP_FAILED) {
    return -1;
}

ret = madvise(addr, size, MADV_RANDOM);
if (ret < 0) {
    return -1;
}
```

There is another group of syscalls called `mlock` and `mlockall` that explain the system to lock a particular region of shared memory and not allow it to be paged. This means, the physical page that has been allocated to the page table entry and any accesses to it will not create a page fault. This means to let the page in the RAM always.

`munlock` unlocks the portion of the memory that has been locked by the `mlock`.

`mlock` tends to improve performance when used in a time sensitive code thus reducing the paging.

`mlock` prototypes are as follows. include `<sys/mman.h>` before using `mlock` system calls.

```
int mlock(const void *addr, size_t len);
int mlock2(const void *addr, size_t len, int flags);
int munlock(const void *addr, size_t len);

int mlockall(int flags);
int munlockall(void);
```

the syscalls, `mlockall` and `munlockall` locks the entire pages mapped to the address space of the calling process.

the parameter `flags` in `mlockall` represents the following:

name	meaning
`MCL_CURRENT`	lock all pages which are currently mapped into the address space of the process
`MCL_FUTURE`	lock all pages which become mapped into the address space of the process in future.
`MCL_ONFAULT`	when used together with `MCL_CURRENT` and `MCL_FUTURE`, mark all current or future pages to be locked into RAM when they are faulted

The calling convention for `mlock` is usually the following.

```
void *addr = malloc(sizeof(int));
if (!addr) {
    return -1;
}

ret = mlock(addr, sizeof(int));
if (ret < 0) {
    return -1;
}
```

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
```

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.

Modifying pid_max

`pid_max` represents the max value of the pid. Path of this file is `/proc/sys/kernel/pid_max`.

Below is an example that changes `pid_max`.

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

#define PID_MAX_PATH "/proc/sys/kernel/pid_max"

int main()
{
    char str[20];
    uint32_t val = 1234567;
    uint32_t max_pid = 0;
    FILE *fd;
    int ret;

    fd = fopen(PID_MAX_PATH, "w");
    if (fd == NULL) {
        perror("fopen");
        return -1;
    }

    sprintf(str, "%u", val);
```

```
    printf("%s\n", str);

    ret = fprintf(fd, "%s", str);
    if (ret > 0) {
        printf("successfully written %u as max pid\n", ret);
    } else {
        printf("failed to write %d\n", ret);
        perror("write");
    }

    fclose(fd);

    fd = fopen(PID_MAX_PATH, "r");
    if (fd == NULL) {
        perror("fopen");
        return -1;
    }

    void *res = fgets(str, sizeof(str), fd);
    if (res == NULL) {
        printf("failed to read back max pid\n");
    } else {
        printf("max_pid %s\n", str);
    }

    fclose(fd);
    return 0;
}
```

Note that beyond a max level, `pid_max` cannot be set beyond the limit.

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.

Here is the slightly modified program from the same list. This program does the sha1.

You can download this program here.

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

```c
#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.

```
#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

```
#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 != 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;
}
```

Above programs use the `sockaddr_alg` structure for communication with the kernel.

The structure looks as below,

```
struct sockaddr_alg {
    __u16 salg_family;
    __u8 salg_type[14];
    __u32 salg_feat;
    __u32 salg_mask;
    __u8 salg_name[64];
};
```

the `salg_family` is usually set to `AF_ALG`.

The supported crypto algorithms are defined in `/proc/crypto`.

Below is an example of reading `/proc/crypto` file. Download here

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

struct crypto_struct {
    char name[40];
    char driver[40];
    char module[40];
    int priority;
    int refcnt;
    char selftest[40];
    char internal[40];
    char type[40];
    int block_size;
    int min_keysize;
    int max_keysize;
    int ivsize;
    char geniv[40];
    int chunksize;
    int walksize;
    int digest_size;
};

void parse_name_val(char *buf, char *name, char *val)
{
    int j;

    int namelen = strlen(name);

    j = namelen;

    while (buf[j] != '\0') {
```

```
        if ((buf[j] == ':') || (buf[j] == ' ')) {
            j ++;
        } else {
            break;
        }
    }
    strcpy(val, &buf[j]);
}

int main()
{
    FILE *fp;
    const char *crypto = "/proc/crypto";
    struct crypto_struct cr[80];
    int j;
    int i;

    fp = fopen(crypto, "r");
    if (!fp) {
        return -1;
    }

    char tmp[40];
    char buf[1024];

    j = 0;
    i = -1;
    while (fgets(buf, sizeof(buf), fp)) {
        buf[strlen(buf) - 1] = '\0';

        if (strstr(buf, "name")) {
            i ++;
            parse_name_val(buf, "name", cr[i].name);
        } else if (strstr(buf, "driver")) {
            parse_name_val(buf, "driver", cr[i].driver);
        } else if (strstr(buf, "module")) {
            parse_name_val(buf, "module", cr[i].module);
        } else if (strstr(buf, "priority")) {

            parse_name_val(buf, "priority", tmp);
            cr[i].priority = atoi(tmp);
        } else if (strstr(buf, "refcnt")) {

            parse_name_val(buf, "refcnt", tmp);
            cr[i].refcnt = atoi(tmp);
        } else if (strstr(buf, "selftest")) {
```

```
        parse_name_val(buf, "selftest", cr[i].selftest);
    } else if (strstr(buf, "internal")) {
        parse_name_val(buf, "internal", cr[i].internal);
    } else if (strstr(buf, "type")) {
        parse_name_val(buf, "type", cr[i].type);
    } else if (strstr(buf, "blocksize")) {

        parse_name_val(buf, "blocksize", tmp);
        cr[i].block_size = atoi(tmp);
    } else if (strstr(buf, "digestsize")) {

        parse_name_val(buf, "digestsize", tmp);
        cr[i].digest_size = atoi(tmp);
    } else if (strstr(buf, "ivsize")) {

        parse_name_val(buf, "ivsize", tmp);
        cr[i].ivsize = atoi(tmp);
    } else if (strstr(buf, "chunksize")) {

        parse_name_val(buf, "chunksize", tmp);
        cr[i].chunksize = atoi(tmp);
    } else if (strstr(buf, "walksize")) {

        parse_name_val(buf, "walksize", tmp);
        cr[i].walksize = atoi(tmp);
    } else if (strstr(buf, "min keysize")) {

        parse_name_val(buf, "min keysize", tmp);
        cr[i].min_keysize = atoi(tmp);
    } else if (strstr(buf, "max keysize")) {

        parse_name_val(buf, "max keysize", tmp);
        cr[i].max_keysize = atoi(tmp);
    }
}

int crypto_len = i;

printf("crypto: {\n");
for (i = 0; i < crypto_len; i ++) {
    printf("\t name: %s\n", cr[i].name);
    printf("\t driver: %s\n", cr[i].driver);
    printf("\t module: %s\n", cr[i].module);
    printf("\t priority: %d\n", cr[i].priority);
    printf("\t refcnt: %d\n", cr[i].refcnt);
    printf("\t selftest: %s\n", cr[i].selftest);
```

```
        printf("\t internal: %s\n", cr[i].internal);
        printf("\t type: %s\n", cr[i].type);
        printf("\t blocksize: %d\n", cr[i].block_size);
        printf("\t digest size: %d\n", cr[i].digest_size);
        printf("\t ivsize: %d\n", cr[i].ivsize);
        printf("\t chunksize: %d\n", cr[i].chunksize);
        printf("\t walksize: %d\n", cr[i].walksize);
        printf("\t min keysize: %d\n", cr[i].min_keysize);
        printf("\t max keysize: %d\n", cr[i].max_keysize);
        printf("\n");
    }
    printf("}\n");

    fclose(fp);
}
```

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

```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

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

```
#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.

```
#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] ## 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.

```c
#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

```
#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.

```
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.

More than one process can open a file and each time a new file descriptor is returned for the same file. Concurrent accesses to the same file must be controlled via some form of the synchronisation.

A file is referenced by an inode with in the kernel. Each inode has a unique value called inode number. An inode stores the metadata of each file, such as modification timestamp, access, creation time, owner, permissions etc.

Below example demonstrates the `open` system call with the `O_EXCL` feature. Download here

```
#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

```
#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

```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

```
#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_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

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

```
#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

type	definition
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

```
#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

```
#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

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

```c
#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, 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

```
#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.

For syncing to the disk the Linux supports `fsync`.

The `fsync` system call syncs the file to the disk flushing the buffers. The prototype is as follows,

```
int fsync(int fd);
```

it accepts a file descriptor of the file and returns 0 on success and -1 on failure.

the `sync` system call syncs the entire system buffers to disk. The prototype is as

follows,

```
void sync(void);
```

the command `sync` does the smae job as well.

While shutting the system down, it is usually a good practise to `sync` the system buffers to the disk, if there is a sufficient power.

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

```
#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

```
#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

```
#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

```
#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,

```
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

```
#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.

Below example describe a bit more about checking the `UID`, `GID` and `sticky` bits. It also describe the number of hardlinks and the block sizes and number of blocks.

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

int main(int argc, char **argv)
{
    struct stat st;
    int ret;

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

    ret = stat(argv[1], &st);
    if (ret < 0) {
        return -1;
    }

    printf("set uid %d set gid %d sticky %d\n",
            st.st_mode & S_ISUID,
            st.st_mode & S_ISGID,
            st.st_mode & S_ISVTX);

    printf("number of hardlinks %d\n", st.st_nlink);

    printf("block size %d\n", st.st_blksize);
    printf("n_512 byte blocks %d\n", st.st_blocks);

    return 0;
}
```

File timestamps can be changed by using the `utime` system call. This system

call can be run on the files with access controls equals or greater.

```
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <utime.h>

int main(int argc, char **argv)
{
    struct utimbuf buf;
    time_t now = time(0);
    int ret;

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

    buf.actime = now - 200;
    buf.modtime = now - 200;

    ret = utime(argv[1], &buf);
    if (ret < 0) {
        perror("utime");
        return -1;
    }

    return 0;
}
```

Above program sets the timestamp 3 mins in the past.

Before running the above program gives,

```
ls -l utime.c
-rw-r--r--. 1 devnaga devnaga 468 Mar  4 06:20 utime.c
```

After running the above program gives,

```
ls -l utime.c
-rw-r--r--. 1 devnaga devnaga 468 Mar  4 06:17 utime.c
```

lstat system call

The `lstat` system call is similar to the `stat` system call, however if the argument to it is a symlink, then information about the symlink is returned.

The prototype of `lstat` system call is as follows.

```
int lstat(const char *restrict pathname,
          struct stat *restrict statbuf);
```

Note that to find a symlink, one can check on the file with `lstat` and then checking for `S_IFLINK` flag.

fstat system call

The `fstat` system call is similar to the `stat` system call, except that the argument is a file descriptor.

The prototype of `fstat` system call is as follows.

```
int fstat(int fd, struct stat *statbuf);
```

`fstat` could be used in cases where the file descriptor of the file in question is already open. ## 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

```
#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

```
#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

```
#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;
}
```

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

```
#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.

```
#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

```
#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.

```
#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. ## 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/ ## chmod system call

The `chmod` system call changes the permissions of a file.

the header file `<sys/stat.h>` contains the prototype of the `chmod` system call.

The prototype of `chmod` is as follows.

```
int chmod(const char *path, mode_t mode);
```

There is one another system call `fchmod`, it does the same job as `chmod` does, but operates instead on a file descriptor.

The prototype of `fchmod` is below.

```
int fchmod(int fd, mode_t mode);
```

Example:

The below program adds the permissions 0777 to the file. The file is taken as input. You can also download it here.

```
#include <stdio.h>
#include <sys/stat.h>

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

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

   ret = chmod(argv[1], S_IRUSR | S_IWUSR | S_IXUSR |
                        S_IRGRP | S_IWGRP | S_IXGRP |
                        S_IROTH | S_IWOTH | S_IXOTH);
   if (ret != 0) {
       printf("failed to chmod [%s] \n", argv[1]);
```

```
        return -1;
    }

    return 0;
}
```

The same example with fchmod is given below, Download here

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <unistd.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_RDWR);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s\n", argv[1]);
        return -1;
    }

    ret = fchmod(fd, 0777);
    if (ret < 0) {
        fprintf(stderr, "fchmod failure %s\n", strerror(errno));
    } else {
        fprintf(stderr, "succesfully set permissions of file %s\n", argv[1]);
    }
    close(fd);

    // perform stat() to verify
    struct stat s_;

    ret = stat(argv[1], &s_);
    if (ret < 0) {
        fprintf(stderr, "failed to stat %s\n", argv[1]);
        return -1;
```

```
    }

    fprintf(stderr, "file permissions %o\n", s_.st_mode);

    return 0;
}
```

above example, operates on a file descriptor setting the file in read + write + execute mode (this is the mode usually operated by the executable files). After setting permissions, the file permissions are verified with the `stat` system call.

Trying the same program on `/dev/null` gets into permissions denied error. (`EPERM`)

This simply defines that the `open` system call creates this file in 0700 mode.

```
./a.out /dev/null
fchmod failure Operation not permitted
file permissions 20666
```

permission bits

The below permission bits describe the mode settings when setting them for a file.

permission value	bit mask
S_ISUID	04000
S_ISGID	02000
S_ISVTX	01000
S_IRUSR	00400
S_IWUSR	00200
S_IXUSR	00100
S_IRGRP	00040
S_IWGRP	00020
S_IXGRP	00010
S_IROTH	00004
S_IWOTH	00002
S_IXOTH	00001

To give acess to user READ , WRITE and EXEC, then we need `S_IRUSR | S_IWUSR | S_IXUSR`.

The `open` system call provides such provisioning of permissions when creating a file. Such as,

```
int fd;

fd = open("./file", O_CREATE | O_RDWR, S_IRWXU);
```

```
if (fd < 0) {
    printf("failed to open file\n");
    return -1;
}
```

However, using `S_IRWXU` simply means that we are providing execute permissions on the file. As we understand, shell / python scripts or binary files are the only ones that require execute permissions. Ideal use of the permissions filed could become `S_IRUSR | S_IWUSR`.

```
int fd;

fd = open("./file", O_CREATE | O_RDWR, S_IRUSR | S_IWUSR);
if (fd < 0) {
    printf("failed to open file\n");
    return -1;
}
```

Remember that when giving the permissions, understand first if the certain user / group really require the permissions.

1. For example, not every file needs to have execute permissions. So default mode could become 0666 for non executable files.
2. For example, not every file needs to be accessed by the group and others in write mode. So the default mode could further optimized to 0644.
3. For example, not every user need to read certain system level files. So the default mode could even become 0640.
4. Executable files does not have to be group and other accessible when run by certain users. So permissions become 0100.

Reasoning behind this is that, system permissions can be exploited by the unprivileged users such as others (everyone else) and could potentially lead to faults or privilege escalations or code execute in privilege modes. Avoid this by explicitly defining permission bits on each and every file in the system.

Now, one cannot simply define permission bits on every file, but build systems such as openwrt, yocto or the buildroot can provide such configurability while creating the root file system.

chown system call

`chown` system call changes the uid and gid of the user to he new user.

The uid and gid can be known by using the `getpwnam` system call.

Below is the example of the `chown` system call.

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

```c
#include <pwd.h>

int main(int argc, char **argv)
{
    struct passwd *pw = NULL;
    int ret;

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

    pw = getpwnam(argv[1]);
    if (pw == NULL) {
        perror("getpwnam");
        return -1;
    }

    ret = chown(argv[2], pw->pw_uid, pw->pw_gid);
    if (ret < 0) {
        perror("chown");
        return -1;
    }

    printf("Chmod of [%s] with user [%s] ok\n", argv[1], argv[2]);
}
```

Change the permissions on chown.c:

```
sudo chown root:root chown.c
[devnaga@fedora cpp]$ ls -l chown.c
-rw-r--r--. 1 root root 553 Mar  4 07:58 chown.c
```

Now run the above program shows,

```
sudo ./a.out devnaga chown.c
Chmod of [devnaga] with user [chown.c] ok

ls -l chown.c
-rw-r--r--. 1 devnaga devnaga 528 Mar  4 07:57 chown.c
```

chown system call requires special permissions on some files.

access system call

The access system call prototype is as follows.

```c
int access(const char *pathname, int mode);
```

`access` system call checks whether the calling process can access the `pathname`. The mode is one of the following.

mode	description
`F_OK`	check if the file exists
`R_OK`	check if the file is read accessible
`W_OK`	check if the file is write accessible
`X_OK`	check if the file is executable accessible

include the header file `<unistd.h>`. The otherway to check the existence of the file is to use `access` system call instead of using `stat` system call for it.

Example:

The below program depicts the usage of the `access` API. You can also download it in here.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    if (argc != 3) {
        fprintf(stderr, "%s [filename] [mode]\n", argv[0]);
        return -1;
    }

    int ret;
    int mode;

    if (!strcmp(argv[2], "F_OK")) {
        mode = F_OK;
    } else if (!strcmp(argv[2], "R_OK")) {
        mode = R_OK;
    } else if (!strcmp(argv[2], "W_OK")) {
        mode = W_OK;
    } else if (!strcmp(argv[2], "X_OK")) {
        mode = X_OK;
    } else {
        fprintf(stderr, "invalid option %s entered\n", argv[2]);
        return -1;
    }
```

```
    ret = access(argv[1], mode);
    if (ret < 0) {
        fprintf(stderr, "failed to access %s error : %s\n",
                                argv[1], strerror(errno));
        return -1;
    }

    fprintf(stderr, "%s is ok\n", argv[2]);
    return 0;
}
```

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));
    if (len == -1) {
        fprintf(stderr, "failed to resolve the link for %s\n", argv[1]);
        return -1;
    }
```

```
    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.

```
#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:

```
#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.

```
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

```
#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}_${number_of_signal}_${timeofc`

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.

prctl

Inotify

The `inotify` are a set of system calls that are useful to monitor the files and directories. When a directory is modified, inotify will return events from the directory itself and also for the files inside the directory.

The watching on the files for any changes is done for the following purposes / needs.

1. configuration changes
2. automatic backup program triggering
3. directory monitoring for suspicious activity
4. automatic crash dump collection of a program (by watching the core files directory and uploading them to the server when any new cores get added)

The following are the useful inotify calls.

```
int inotify_init(void);
int inotify_add_watch(int fd, const char *path, uint32_t mask);
int inotify_rm_watch(int fd, uint32_t mask);
```

Include `sys/inotify.h` to use the above API.

The `inotify_init` creates a file descriptor that can be used to receive the file or directory events. The file descriptor can also be monitored via the `select` calls.

The `inotify_add_watch` adds a new watch entry or modifies an existing watch entry to the list of monitored files or directories. The `fd` is the file descriptor returned from the `inotify_init`. The `path` is either a file or a directory.

The `inotify_rm_watch` removes the inotify watch that was previously added with the `inotify_add_watch`.

The returned fd is monitored via the `select`, `poll`, `epoll` system calls. The `read` call is then called upon a return from the `select`, `poll` or `epoll` to read the events.

The mask argument in the `inotify_add_watch` has the following types.

mask	description
`IN_ACCESS`	File was accessed via read
`IN_ATTRIB`	permissions, timestamps, such as file options have changed
`IN_CLOSE_WRITE`	file opened for writing was closed
`IN_CLOSE_NOWRITE`	file or directory not opened for writing was closed
`IN_CREATE`	File/Directory created in the watched directory
`IN_DELETE`	File/Directory deleted from watched directory
`IN_DELETE_SELF`	Watched file/directory itself is deleted
`IN_MODIFY`	File was modified
`IN_MOVE_SELF`	Watched file/directory itself moved
`IN_MOVED_FROM`	Generated for the directory containing the new filename when a file is renamed
`IN_MOVED_TO`	Generated for the directory containing the new filename when a file is renamed
`IN_OPEN`	File/Directory was opened

The `IN_ALL_EVENTS` flag is used to monitor all of the above events.

When there is an event, the `read` system call on the inotify fd might return a set of events instead of filling one event at a time.

The events are a set of objects of type `struct inotify_event` that looks as follows (for reference).

```
struct inotify_event {
    int wd;   // watch descriptor
    uint32_t mask;  // mask describing the event
    uint32_t cookie; // session cookie or something
    uint32_t len;  // length of the below `name` field
    char name[];  // optional null-terminated string that contains the name of the directory
};
```

The reading of the events might look as the following ...

```
int ret;
int processed = 0;

while (processed < read_bytes) {
    struct inotify_event *event;

    event = (struct inotify_event *)(buf + processed);
    // event processing
    processed += sizeof(struct inotify_event);
}
```

Below is the sample program that demonstrate the `inotify` system call usage.

```
#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/inotify.h>

int main(int argc, char **argv)
{
    int fd;
    int ret;
    char buf[4096] __attribute__((aligned(8)));
    int num_read;
    struct inotify_event *event;

    if (argc != 2) {
        printf("%s [file / directory]\n", argv[0]);
        return -1;
```

```
    }

    fd = inotify_init();
    if (fd < 0) {
        printf("failed to inotify_init\n");
        return -1;
    }

    ret = inotify_add_watch(fd, argv[1],
                                    IN_ACCESS |
                                        IN_CREATE |
                                        IN_DELETE |
                                        IN_OPEN);
    while (1) {
        int processed = 0;

        ret = read(fd, buf, sizeof(buf));
        if (ret < 0) {
            break;
        }

        while (processed < ret) {
            event = (struct inotify_event *)(buf + processed);

            if (event->mask & IN_ACCESS) {
                printf("Read event on file %s\n", event->name);
            }

            if (event->mask & IN_CREATE) {
                printf("File created %s\n", event->name);
            }

            if (event->mask & IN_DELETE) {
                printf("File deleted %s\n", event->name);
            }

            if (event->mask & IN_OPEN) {
                printf("File is in open %s\n", event->name);
            }
            processed += sizeof(struct inotify_event);
        }
    }
}
```

Example: inotify example

The `inotify` is mostly used by the build systems / continuous integration tools

to monitor directories (such as the ones that contain the release images for a new software release etc) for some interesting events. # Seventh chapter

POSIX Threads

Thread is light weight process.

- 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. ## 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
```

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

```
#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.

```
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 caught 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

```
#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

```
#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);
    }

    pthread_mutex_destroy(&lock);
```

```
    return 0;
};
```

The above example demonstrates the variable `t` that is shared between the main program and the thread. The main program increments the variable and the thread does increment it. Due to this, the varible must be protected from the concurrent access. This is where the mutex is used to protect the access to the variable.

The thread increments the variable `t` and the main thread reads the variable `t`. So while incrementing it, it must be protected.

```
pthread_mutex_lock(&lock);
(*ptr) ++;
pthread_mutex_unlock(&lock);
```

while accessing it, the below code adds the lock and unlock while being accessed.

```
pthread_mutex_lock(&lock);
printf("t value %d\n", t);
pthread_mutex_unlock(&lock);
```

Below is the prototype of `pthread_mutex_timedlock`.

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex, const struct timespec *abstime);
```

Below is the prototype of `pthread_mutex_destroy`.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Every created mutex shall be destroyed with `pthread_mutex_destroy`. The mutex after calling `pthread_mutex_destroy` will be uninitialized.

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.

the `pthread_cond_signal` prototype is as follows,

```
int pthread_cond_signal(pthread_cond_t *cond);
```

the `pthread_cond_wait` prototype is as follows,

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

The `pthread_cond_signal` is used to signal other thread about the availability. It is used in conjunction with the `pthread_cond_wait`.

An example of it is the main thread signals the worker threads about the work available. The main thread sets the work object or queues it. It then signals the other worker threads about the work availability. Each thread waiting on the condition, wakes up and executes the work assigned to it.

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

```
#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

```
#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

```c
#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 pthread_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`.**

Opensource Software tools and libraries

##Libpcap

##Tcpdump

- the very powerful Packet capturing tool
- bare minimal example:

  ```
  tcpdump -i eth0
  ```

the above command captures the packets over ethernet

- also used for debugging the packet formats, proto handshakes such as request and reply messages.

##GPSd

- The very exceptional GPS device data capture utility

##busybox

- the tooling utility to provide basic shell commands and the shell.
- supports a wide variety of services such as dropbear, telnet, init process etc.

##libnl

- Userspace netlink library to talk with the kernel space drivers.
- very versatile and highly flexible for moderately bigger data transfers

##libxml2

- C based xml parsing library.
- Simple API to read and analyse the XML file

##protobuf Google

- A serialisation format from Google
- protobuf is a series of instructions that are written in the language specified in the protobuf spec
- Developer manual is at : https://developers.google.com/protocol-buffers/
- Language neutral, platform neutral, extensible mechanism for serializing structured data

##ASN1

- encoding rules defined by X.509 specification. Specified BER, DER, XER, PER and UPER version of encoding methods.

##perf

- A linux application and system performance evaluation library.

##valgrind

- A memory leak detector (mostly) and performance evaluator for a program.

##libevent

##libphenom

- Facebook's popular event library.

##mbedtls

- very simple and versatile security software library.

Event Library

In this chapter we are going to write an event library that allows us to do multiple jobs in a single process.

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: `bash  sudo apt-get install valgrind`

   on Fedora: `bash  #  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-progr
   ```

   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-orig
   ```

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 (Ubu
--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_wrappe
--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_wrapp
--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_ifun
--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

==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 (Ubu
--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_wrappe
--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_wrapp
--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_ifun
--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
==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 (Ubu
--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_wrappe
--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_wrapp
--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_ifun
--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. ## 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.