Basics of File Handling in C

So far the operations using C program are done on a prompt / terminal which is not stored anywhere. But in the software industry, most of the programs are written to store the information fetched from the program. One such way is to store the fetched information in a file. Different operations that can be performed on a file are:

1. Creation of a new file (**fopen with attributes as "a" or "a+" or "w" or "w+")**
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf or fgetc**)
4. Writing to a file (**fprintf**)
5. Moving to a specific location in a file (**fseek, rewind**)
6. Closing a file (**fclose**)

Opening or creating file

For opening a file, **fopen** function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

File opening modes in C:

- r opens a text file for reading.
- r+ opens a text file for reading and writing

  If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.

- w opens a text file for writing.
- w+ opens a text file for both reading and writing. it first cuts the length of the file to zero if it exists, or create a file if it does not exist). Returns NULL, if unable to open file.

- a opens a text file for appending. (appending means to add text at the end)
- a+ opens a text file in both reading and writing mode. (It creates a file if it does not exist. Reading will start from the beginning but Writing can only be done at the end). Returns NULL, if unable to open file

Reading from a file :

The file read operations can be performed using functions **fscanf or fgets**. Both the functions performed the same operations as that of printf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.

And the code snippet for reading a file is as:

```
FILE * filePointer;
filePointer = fopen("fileName.txt", "r");
fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);
```

Writing a file :

The file write operations can be performed by the functions **fprintf and fputs** with similarities to read operations. The snippet for writing to a file is as :

```
FILE *filePointer;
filePointer = fopen("fileName.txt", "w");
fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2020);
```

Closing a file :

After every successful fie operations, **you must always close a file**. For closing a file, you have to use **fclose** function. The snippet for closing a file is given as :

```
FILE *filePointer ;
filePointer= fopen("fileName.txt", "w");
---------- Some file Operations -------
fclose(filePointer)
```

C I/O Library

- Defined in **<stdio.h>**
  - scanf, printf
    - **scanf("%d", &num);**
    - **printf("Hello World\n");**
  - fscanf, fprintf
    - **fprintf(file, "Hello World\n");**
    - **fscanf(file, "%d", &num);**
    - **file** is a FILE* pointer
      - stdout, stderr, stdin
- Not required, but highly recommended
  - more suitable and efficient than **cout** and **cin** in our problems

Conversion specifiers

Works with family of printf() and scanf() functions such as :

```
printf("Hello %s\n", name);
scanf("%d", x);
```

- %d
  - print argument as an integer
- %f
  - print argument as a floating-point value
  - formatting specifiers also available
- %c
  - print argument as a char
  - value argument should generally be between 32 and 126
- %s
  - print argument as an array of char
  - argument is expected to be a pointer to a char
  - string is expected to be null terminated

printf() and fprintf()

The function **printf()** is used to print the message along with the values of variables

The function **fprintf()** is known as format print function. It writes and formats the output to a stream. It is used to print the message but not on stdout console.

Header

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

Prototype

```
printf(const char *format, ...);
fprintf(FILE *stream, const char *format, ...);
```

Call

```
printf("Hello %s\n", name);
fprintf(stderr, "Hello %s\n", name);
```

- displays data to a stream
- notion of streams similar to C++
    - for **printf()** it's always stdout
    - for **fprintf()** it's stdout, stderr or any regular file (FILE)

scanf() and fscanf()

The function **scanf()** is used to **read formatted input from stdin** in C language. It returns the whole number of characters written in it otherwise, returns a negative value.

The function **fscanf()** is used to **read the formatted input from** stdin **or given stream / regular er file** in C language. It returns zero, if unsuccessful. Otherwise, it returns The input string, if successful.

Header

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

Prototype

```
scanf(const char *format, ...);
fscanf(FILE *stream, const char *format, ...);
```

Call

```
scanf("Hello no %d\n", &num);
fscanf(stdin, "Hello %s\n", name);
```

- All arguments should be pointers or memory addresses
- Will add null-termination if value is read into a string
- Will discard CRLF if value read from keyboard

Example : scanf , printf

```
#include <stdio.h>

int main() {
	char str1[20], str2[30];

	printf("Enter name: ");
	scanf("%s", str1);

	printf("Enter your website name: ");
	scanf("%s", str2);

	printf("\n\nEntered Name: %s\n", str1);
	printf("Entered Website:%s", str2);

	return (0);
}
```

Sample Run

Enter name: Husain
Enter your website name: www.husaingholoom.com

Entered Name: Husain
Entered Website:www.husaingholoom.com

Example : fscanf , fprintf

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

int main() {
        char str1[10], str2[10], str3[10];
        int year;
        FILE * fp;

        fp = fopen("file.txt", "w+");
        fprintf(fp, "%s %s %s %d", "We", "are", "in", 2020);


        rewind(fp);
        fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

        printf("Read String1 |%s|\n", str1);
        printf("Read String2 |%s|\n", str2);
        printf("Read String3 |%s|\n", str3);
        printf("Read Integer |%d|\n", year);

        fclose(fp);

        return (0);
}
```

Sample Run

Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2020|

Example : fprintf

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

char *weekday = { "Saturday" };
char *month = { "October" };

int main( void )
{
   fprintf( stdout, "%s, %s %d, %d\n",
      weekday, month, 24 , 2020 );

   return EXIT_SUCCESS;
}
```

Sample Run

Saturday, October, 24, 2020

puts()

The C library function **puts()** writes the string s and a trailing newline to stdout.

Example

```
#include <stdio.h>
int main()
{
   puts("Unix  ");
   puts("Programming ");

   getchar();
   return 0;
}
```

Sample Run

```
Unix
Programming
```

fputs()

The C library function **fputs()** writes a string to the specified stream up to but not including the null character.

Example

```
#include <stdio.h>

int main () {

  FILE *fp;

  fp = fopen("file.txt", "w+");

  fputs("This is c programming.", fp);
  fputs("This is a system programming language.", fp);

  fclose(fp);

  return(0);
}
```

System Functions in C – system()

The C library function **int system(const char *command) is used to pass the commands that can be executed in the command processor or the terminal of the operating system, and finally returns the command after it has been completed.**

Header file

#include <stdlib.h>

Prototype

int system(const char *string);

Call

system("echo hello world");

- Can separate multiple commands within the string using semicolons or newlines
- The executed commands have access to current environment variables

Return value

- **Returns** the **exit** status of the ***last*** command executed
- The value does not always match the exit status you see in the shell
- **Returns -1 if the system() function call fails to execute**

Example : system()

```
#include <stdlib.h>

int main() {

    system("echo hello world");
    system("echo");
    system("date");
    system("echo");
    system("cal");


    return 0;
}
```

Sample run

```
hello world

Sat Oct 25 10:52:03 CDT 2020

   October 2020
Su  Mo  Tu  We  Th  Fr  Sa
                 1   2   3
 4   5   6   7   8   9  10
11  12  13  14  15  16  17
18  19  20  21  22  23  24
25  26  27  28  29  30  31
```

What is the output of the following

```
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* system, NULL, EXIT_FAILURE */

int main() {
        int i;
        printf("Checking if processor is available...");
        if (system (NULL))
                puts("Ok");
        else
                exit(EXIT_FAILURE);
        printf("Executing a command ...\n");
        i = system("ls");
        printf("The value returned was: %d.\n", i);
        return 0;
}
```

Sample run

Checking if processor is available...Ok
Executing a command ...

.
.
.

The value returned was: 0.

System Calls in C

- A large collection of pre-defined system functions are available to each C program
  - When we call one of these functions from our C source file it is commonly referred to as a *system call*
  - Some functions have equivalent Unix commands
    - **getenv(), chmod()**
  - Some functions do not have equivalent Unix commands
    - **fread(), lseek() / fseek() (** change the **location** of the read/write pointer of a file to end , starting , or current position of the file)
  - Some functions behave differently than their corresponding Unix commands
    - **umask() (set permission)**
- Some low-level functions are written in assembly language but provide C-like interface
- When a process **invokes a system call** the process switches from ***user mode*** to ***kernel mode***
  - Privileged mode
    - Can access memory outside own address space
  - Does **not** make the user root!
  - There is some **overhead** in switching back and forth between the two modes
    - should use system calls with good judgment.

System Calls for the Environment

- System calls to access information about the environment
  - User
  - System
  - Process
  - Time

- We already know how to get this information in shell
- Cannot use shell for everything

Why Use System Calls?

- **Many system functions are heavily used in system-level programming**

  - Low-level file I/O
  - Client-server programming
  - Network programming
  - System functions provide easy access to system information that would otherwise be difficult to find

- System functions often take care of the dirty work underneath to provide a clean and efficient interface to user programs

  - sockets, signals, threads

Library Functions vs System Calls

- Standard library functions are associated with C/C++ compilers
  - Bundled with gcc and other commercial compilers

- Provide a convenient wrapper for low-level system calls
  - **fread(), fgets(), fgetc()** all call **read()** internally

- **Do not provide enough fine-grain control**
  - **fopen()** does not support chmod-like permissions

- **No equivalent library calls for some functions**
  - **fork(), chmod()** etc.

- **Information**

  - man -s -2 system — for system calls
  - man -s 3 function — for library functions

Example : fread()

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

int main() {
        FILE *fp;
        char c[] = "This is an example of using fread()";
        char buffer[100];

        /* Open file for both reading and writing */
        fp = fopen("file.txt", "w+");

        /* Write data to the file */
        fwrite(c, strlen(c) + 1, 1, fp);

        /* Seek to the beginning of the file */
        fseek(fp, 0, SEEK_SET);

        /* Read and display data */
        fread(buffer, strlen(c) + 1, 1, fp);
        printf("%s\n", buffer);
        fclose(fp);

        return (0);
}
```

Sample Run

This is an example of using fread()

What is the output of the following program

```
#include <stdio.h>

int main() {
        FILE *fp;

        fp = fopen("file.txt", "w+");
        fputs("This is Another Example of fseek", fp);

        fseek(fp, 7, SEEK_SET);
        fputs(" C Programming Language", fp);
        fclose(fp);

        return (0);
}
```

Sample run

This is C Programming Languageek

Example : fgets()

```
// C program to illustrate fgets()

#include <stdio.h>
#define MAX 15
int main() {
        char buf[MAX];
        printf("Enter a string    ");

        fgets(buf, MAX, stdin);
        printf("\n\nstring is: %s\n", buf);

        return 0;
}
```

Sample Run

Enter a string   Texas State University

string is: Texas State Un

Example : fgetc()

```
#include <stdio.h>

int main() {
	FILE *fp;
	int c;
	int n = 0;

	fp = fopen("file.txt", "r");
	if (fp == NULL) {
		perror("Error in opening file");
		return (-1);
	}
	do {
		c = fgetc(fp);
		if (feof(fp)) {
			break;
		}
		printf("%c  ", c);
	} while (1);

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

```
We are in 2020
```

Why Use These Functions?

When programming in Unix, it is important to make the program aware of the environment

- **Portability**
  - Conditional compilation with #ifdef

```
#ifdef REDHAT
   // then use boost for multithreading
#endif
```

- **Security**
  - If the user is not root then I really shouldn't be executing this code

- **Robustness**
  - If PWD is / then …

- **Performance**
  - Same strategy as portability, but different goal

Locating System Functions

- Difficult to provide an exhaustive list
    - list is quite large (over 1000's in most systems)
    - may vary from one Unix installation to another
- Usual location of header files
    - /usr/include
    - /usr/include/linux
    - /usr/include/sys
    - /usr/local/include
- Some header files we will be looking at
    - **<unistd.h>** (**unistd.h** is the name of the header file that provides access to the **Portable Operating System Interface** **API** . Typically , It is made up largely of system call wrapper functions such as fork, pipe and I/O primitives such as read, write, close, etc.)
    - <time.h>
    - <sys/types.h>
    - …

Frequently Used System Calls

- Getting information about the environment
  - **getenv(), putenv() , getcwd()**

- Directory and File Operation
  - **chdir(), chmod(), umask()**

- File I/O
  - Beyond **fprintf()** and **fscanf()**
    - read binary data
  - **open(), close(), read(), write(), lseek(),**

getenv()

getenv() takes ‘NAME environment variable’, searches this in the list and when found returns its value as a string. When it fails, it returns NULL pointer

Header file

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

Prototype

```
char *getenv(const char *name);
```

Call

```
char *envVar;
envVar = getenv(“HOME”);
```

Return value

- returns null if it fails or there is no match
- returns a pointer to the location of the value held in var

Example : getenv

```
#include <stdlib.h>

int main(void) {
        char *str;

/* attempt to access environment variables using getenv() */

((str = getenv("HOME")) != NULL);
printf("\nValue of \"HOME Environment Variable\" : %s\n", str);

((str = getenv("LOGNAME")) != NULL);
printf("\nValue of \"LOGNAME Environment Variable\" : %s\n", str);

/* attempt to access variable of ours' choice */
((str = getenv("MELLO")) != NULL);
printf("\nValue of \"MELLO Environment Variable\" : %s\n", str);

        return 0;
}
```

Sample Run

Value of "HOME Environment Variable" : /Users/husaingholoom

Value of "LOGNAME Environment Variable" : husaingholoom

Value of "MELLO Environment Variable" : (null)

Puten v()

The **putenv()** function **adds or changes the value of environment variables.** The argument string is of the form name=value. If name does not already exist in the environment, then string is added to the environment. If name does exist, then the value of name in the environment is changed to value. The string pointed to by string becomes part of the environment, so altering the string changes the environment

ENOMEM Insufficient space to allocate new environment

Header file

#include<stdlib.h>

Prototype

int putenv(const char *str);

Call

putenv(“HOME=/home/zz01”);

Return value

- returns -1 if it fails
- returns 0 when it succeeds

Example : putenv (change or add a value to an environment)

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

int main( void )
{
        char *path;
        path = getenv( "INCLUDE" );
        if( path != NULL ) {
                printf( "INCLUDE=%s\n", path );
        }

        printf( "\nINCLUDE After getenv =  %s\n", path );

        if( putenv( "INCLUDE=/src/include" ) != 0 ) {
                printf( "putenv() failed setting INCLUDE\n" );
                return EXIT_FAILURE;
        }

        path = getenv( "INCLUDE" );
        if( path != NULL ) {
                printf( "INCLUDE after putenv =  %s\n", path );
        }

        unsetenv( "INCLUDE" );

        return EXIT_SUCCESS;
}
```

```
INCLUDE After getenv =  (null)
INCLUDE after putenv =  /src/include
```

getcwd()

get pathname of current working directory

The getcwd() function may fail if A parent directory cannot be read to get its name.

/* The following example prints the current working directory. */

Example : getcwd (Determines the current working directory.)

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

int main() {
        char cwd[256];

        if (getcwd(cwd, sizeof(cwd)) == NULL)
            perror("getcwd() error");
        e
lse
        printf("current working directory is: %s\n", cwd);

}
```

current working directory is: /Users/husaingholoom/Documents/F2020-TestPg

Example : chdir (changing the current directory)

```
#include<stdio.h>

// chdir function is declared
// inside this header
#include<unistd.h>
int main() {
        char s[100];

        // printing current working directory
        printf("%s\n", getcwd(s, 100));

        // using the command
        chdir("..");

        // printing current working directory
        printf("%s\n", getcwd(s, 100));

        // after chdir is executed
        return 0;
}
```

/Users/husaingholoom/Documents/F2020-TestPg
/Users/husaingholoom/Documents

Example : chmod

Read by owner only
```
$ chmod 400 sample.txt
```
Read by group only
```
$ chmod 040 sample.txt
```
Read by anyone
```
$ chmod 004 sample.txt
```
Write by owner only
```
$ chmod 200 sample.txt
```
Write by group only
```
$ chmod 020 sample.txt
```
Write by anyone
```
$ chmod 002 sample.txt
```
Execute by owner only
```
$ chmod 100 sample.txt
```
Execute by group only
```
$ chmod 010 sample.txt
```
Execute by anyone
```
$ chmod 001 sample.txt
```
Allow read permission to owner and group and anyone.
```
$ chmod 444 sample.txt
```
Allow everyone to read, write, and execute file.
```
$ chmod 777 sample.txt
```

You can also use the `r`, `w`, and `x` specify the read, write, and execute access for owner , group and everyone.

```
Example

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

chmod("home/xyz/file.txt", S_IRWXU|S_IRWXG)

Accessing User Information

getuid() : get user id
getgid() : get group id
getlogin() : get user logged in

getpwuid() : get more information about the id
getpwnam() : get struct passwd associated with the name such as

```
char    *pw_name      user's login name
char    *pw_passwd    user password
                      ( System Dependent )
uid_t    pw_uid       numerical user ID
gid_t    pw_gid       numerical group ID
char    *pw_dir       initial working directory
char    *pw_shell     User's login shell.
```

It is in <pwd.h> Library

getuid()

Header

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

Prototype

```
uid_t  getuid();
```

Call

```
uid_t  uid;
uid = getuid();
```

- On a Unix system every user has a *name* and a *numeric id*
- **getuid() returns the numeric user id associated with the program**
- *user id* of a process is the user id of user who ***runs*** the program

Example

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

int main() {

        printf("The process ID is %d\n", getuid());

        return 0;

}
```

The process ID is 501

getgid()

Header

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

Prototype

```
gid_t getgid();
```

Call

```
gid_t gid;
gid = getgid();
```

- On a Unix system every user belongs to a group
- Every group has a *name* and *numeric id*
- **getgid() returns the numeric group id associated with the program**
    - usually the primary group of the user who runs the program

Example

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

int main() {

        int gid = getgid();

        printf("The Group  ID is %d\n", gid);

        return 0;

}
```

The Group ID is 20

getlogin();

Header

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

Prototype

```
char* getlogin();
```

Call

```
printf("The user is %s\n", getlogin());
```

- We can also **get the login name**
- Not used a whole lot other than informational purposes

Example

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

int main() {
	char *name;
	name = getlogin();
	if (!name)
		perror("getlogin() error");
	else
		printf("This is the login info: %s\n", name);
	return 0;
}
```

This is the login info: hag10

getpwuid()

- Header

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

- Prototype

struct passwd *getpwuid();
struct passwd *getpwnam();

- **Function allows a process to gain more knowledge about user uid.**
- Uses information from **/etc/passwd** file
- Becoming non-standard for security issues

Example : Getting a User Name from a UID

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

int  main() {
        struct passwd *pw;

        if ((pw = getpwuid(getuid())) == NULL) {
                fprintf(stderr, "getpwuid: no password entry\n");
                exit(EXIT_FAILURE);
        }
        printf("login name  %s\n", pw->pw_name);
        printf("user ID     %d\n", pw->pw_uid);
        printf("group ID    %d\n", pw->pw_gid);
        printf("home dir    %s\n", pw->pw_dir);
        printf("login shell %s\n", pw->pw_shell);
        exit(EXIT_SUCCESS);
}
```

```
login name  husaingholoom
user ID     501
group ID    20
home dir    /Users/husaingholoom
login shell /bin/bash
```

Modifier Functions for user ID

- When a process is running it enjoys the privileges of the user who invoked the process
- There are system calls that can change the user id of the running process

setuid() Sets the real, effective, or saved set user IDs (UIDs) for the current process to uid.

setgid() Finds the real user ID (UID) of the calling process.

- Does it make sense to have these?
- Yes, but only for root
  - allow the program to do stuff as root
    - write to a system location
    - execute a privileged command (useradd?)
- Be careful when using these functions as root, may not be able to come back to root

Examples:

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

int main() {
        printf("\nprior to setuid(), uid=%d ",  (int) getuid());
        if (setuid(25) != 0)
        perror("\nsetuid() error");
        else
        printf("\nafter setuid(),    uid=%d ",  (int) getuid());
}
```

```
before setuid(), uid= 0
after setuid(),    uid= 25
```

Effective User ID

- Sometimes a user may **want to write a program that other users on the system can execute**
- In these situations a specific user may want to allow extra privileges to the users of the program
- This can be achieved by modifying the **effective** user ID of the process
- In addition to the real user ID, every process also has an **effective user id** associated with it
- The real user id is always set to the user who invokes the program
- By default, the effective user ID has the same value as the real user ID

geteuid()

Returns the effective user ID of the calling process.

Header

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

Prototype

```
uid_t geteuid();
```

Call

```
int euid = geteuid();
```

EXAMPLE

The following code fragment illustrates the use of geteuid to determine the effective user ID of a calling process:

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

int main()
{
	uid_t uid, euid;

	uid = getuid();
	euid = geteuid();
	printf("\nThe effective user id is  %d", (int) geteuid());
	printf("\nThe real user id is  %d", (int) getuid());
	if (uid == euid)
	printf("\nEffective user id and Real user id are the same");
}
```

Sample Run :

The effective user id is 9812
The real user id is 9812
Effective user id and Real user id are the same

seteuid()

Set the effective user ID

Header

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

Prototype

```
int seteuid(uid_t euid);
```

Call

```
int error = seteuid(getuid());
```

Examples:

```
/*
 * This process sets its effective userid to 25 (root).
 */

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

int main() {
   printf("your effective user id is %d\n", (int)
         geteuid());
       if (seteuid(25) != 0)
       perror("seteuid() error");
       else
       printf("your effective user id was changed to %d\n", (int) geteuid());
}
```

```
your effective user id is 0
your effective user id was changed to 25
```

Providing Extra Privilege with EUID

- Set the set-user-bit

  $ chmod u+s prog

  - when prog is executed its effective user id is going to be set to the **owner** of prog
  - prog will have same privileges as the owner of prog not the caller of prog

- **Use seteuid() to re-set the effective user ID back to real user ID**

types.h

- A collection of ***typedefs***
- Definitions may vary across systems
- Determines storage requirements for different types

uid_t -> short
size_t -> int
id_t -> int
gid_t -> int
pid_t -> int
mode_t -> uint
time_t -> long

- Should use these defined types even if program compiles when using primitive data types

Dynamic Memory Allocation

malloc() and calloc()

Header

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

Prototype

```
void *malloc (size_t  size);
void *calloc (size_t  elem,  size_t  size);
```

Call

```
int *p = (int *) malloc(sizeof(int) * SIZE);
char *q = (char *) calloc(sizeof(int) * SIZE);
```

- **malloc** Allocate dynamic memory in heap
- The `malloc()` function **allocates a single block** of memory. Whereas, `calloc()` **allocates multiple blocks of memory** and initializes them to zero.
- **calloc** **stands** for contiguous memory allocation.
- **calloc()** gives memory with all zero bits, equivalent to **malloc()** and **memset()**

free()

Header

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

Prototype

```
void *free (void *ptr);
```

Call

```
int *p = malloc();
...

free(p);
```

- Gives memory back to the system
- Can't de-allocate same memory twice
- Shouldn't de-allocate memory that we may later use

Example : malloc , free

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

int main()
{
   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) malloc(n * sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }

   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }

   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

Enter number of elements: 5
Enter elements: 1
2
3
4
5
Sum = 15

Example : calloc , free

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

int main()
{
   int n, i, *ptr, sum = 0;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   ptr = (int*) calloc(n * sizeof(int));
   if(ptr == NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }

   printf("Enter elements: ");
   for(i = 0; i < n; ++i)
   {
      scanf("%d", ptr + i);
      sum += *(ptr + i);
   }

   printf("Sum = %d", sum);
   free(ptr);
   return 0;
}
```

Enter number of elements: 5
Enter elements: 1
2
3
4
5
Sum = 15

Example : memset() is used to fill a block of memory with a particular value.:

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

int main() {
        char str[50];

        strcpy(str, "This is string.h library
                        function");
        puts(str);

        memset(str, '$', 7);
        puts(str);

        return (0);
}
```

```
This is string.h library function
$$$$$$$ string.h library function
```

Things to pay attention to …

- Header files
- Return Values and Handling error codes
- Special data structures and Calls that return information in a struct
    - e.g. passwd struct

Accessing System Information

Two commonly used system calls are :-

- **gethostname()**
- **gethostbyname()**
- **uname()**

Host Name

The *hostent* structure is defined in *<netdb.h>* :

```
struct hostent {
    char  *h_name;   /* official name of host */
    char **h_aliases;    /* alias list */
    int    h_addrtype;   /* host address type */
    int    h_length;     /* length of address */
    char **h_addr_list;  /* list of addresses */

     }
```

The members of the *hostent* structure are:

- *h_name* The official name of the host.
- *h_aliases*   An array of alternative names for the host, terminated by a null pointer.
- *h_addrtype* The type of address.
- *h_length*   The length of the address in bytes.
- *h_addr_list*   An array of pointers to network addresses for the host.
- *h_addr* The first address in *h_addr_list* for backward compatibility.

gethostname()

- **These system calls are used to access the system hostname. It operates on the hostname associated with the calling process's / current machine**

- **Copies the hostname of the current machine in the first argument**

- **Second argument specifies the number of characters that are copied**

- **If the hostname is too large to fit, then the name is truncated, and no error is returned**

- Similar to uname -n or hostname

- Returns -1 if there is an error, 0 if successful

gethostname()

Header

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

Prototype

```
int gethostname(char *name, size_t namelen);
```

Call

```
char host[255];
gethostname(host, 255);
```

Example : gethostname()

// C Program to Display Host Name

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

int main() {

        char hostbuffer[256];
        int hostname;

        // To retrieve hostname
        hostname = gethostname(hostbuffer, sizeof(hostbuffer));

        if (hostname == -1) {
                perror("gethostname");
                exit(1);
        }
        printf("Hostname: %s\n", hostbuffer);

        return 0;
}
```

Hostname: husain-gholooms-macbook.local

sethostname()

- If the process has **appropriate privileges** the function shall change the host name for the current machine

- Sets the hostname to the value given in the character array *name*.

- The *len* argument specifies the number of bytes in *name*.

- (Thus, *name* does not require a terminating null byte.)

    - On success, 0 is returned. On error, -1 is returned and the global variable errno is set appropriately.

```
#include <unistd.h>

int sethostname(const char *name,
size_t len);
```

gethostbyname() : The function retrieves host information corresponding to a host name from a host database.

// C Program to use gethostbyname – Specific Host Name

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

// Returns host information corresponding to host name
void checkHostEntry(struct hostent * hostentry) {
	if (hostentry == NULL) {
		perror("gethostbyname");
		exit(1);
	}
	else
		printf("Host is found");
}

// Driver code
int main() {

	struct hostent *host_entry;


	host_entry = gethostbyname(
	   "husain-gholooms-macbook.local");
	checkHostEntry(host_entry);

	return 0;
}
```

Host is found

// C Program to use gethostbyname – Unknown Host Name

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

// Returns host information corresponding to host name

void checkHostEntry(struct hostent * hostentry) {
        if (hostentry == NULL) {
                perror("gethostbyname");
                exit(1);
        }
        else
                printf("Host is found");
}

// Driver code
int main() {

        struct hostent *host_entry;


        host_entry = gethostbyname(
                  "usain-gholooms-macbook.local");
        checkHostEntry(host_entry);

        return 0;
}
```

gethostbyname: Undefined error: 0

// C Program to Display Host Name

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

// Returns hostname for the local computer
void checkHostName(int hostname) {
        if (hostname == -1) {
                perror("gethostname");
                exit(1);                }
}

// Returns host information corresponding to host name
void checkHostEntry(struct hostent * hostentry) {
        if (hostentry == NULL ) {
                perror("gethostbyname");
                exit(1);        }
}

int main() {
        char hostbuffer[256];
        struct hostent *host_entry;
        int hostname;

        // To retrieve hostname
        hostname = gethostname(hostbuffer, sizeof(hostbuffer));
        checkHostName(hostname);

        // To retrieve host information
        host_entry = gethostbyname(hostbuffer);
        checkHostEntry(host_entry);

        printf("Hostname: %s\n", hostbuffer);

        return 0;
```

Hostname: husain-gholooms-macbook.local

uname() : Provides system information in a struct called utsname

Header

```
#include<sys/utsname.h>
```

Prototype

```
int uname(struct utsname *name);
```

Call

```
struct utsname uts;     // allocation for struct
uname(&uts);
printf("%s\n", uts.sysname);
```

- **Need to allocate struct and send address to function**

```
struct utsname {

        char sysname[];
        char nodename[];
        char release[];
        char version[];
        char machine[];

};
```

- sysname : Operating system name (e.g., "Linux")
- nodename : Name within "some implementation-defined network"
- release : Operating system release (e.g., "2.6.28")
- char version : Operating system version
- char machine : Hardware identifier

Example

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

int main()
{
        int uname(struct utsname *name);

        struct utsname uts;     // allocation for struct
        uname(&uts);
        printf("System Name  %s\n", uts.sysname);
        printf("Node  Name   %s\n", uts.nodename);
        printf("Release   %s\n", uts.release);
        printf("Version   %s\n", uts.version);
        printf("Machine   %s\n", uts.machine);

}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

System Name Darwin
Node Name   husain-gholooms-macbook.local
Release   11.4.2
Version   Darwin Kernel Version 11.4.2: Thu Aug 23 16:26:45 PDT 2012; root:xnu-1699.32.7~1/RELEASE_I386
Machine   i386

husain-gholooms-macbook:~ husaingholoom$

Time related system calls

The **time.h** header defines four variable types, and various functions for manipulating date and time.

Following are the variable types defined in the header time.h

- **size_t** : This is the unsigned integral type and is the result of
  the **sizeof** keyword.

- **clock_t** : This is a type suitable for storing the processor
  time.

- **time_t is** : This is a type suitable for storing the calendar
  time.

- **struct tm** : This is a structure used to hold the time and date.

The time struct

The **tm structure** contains the following members

```
struct tm {
        int tm_sec; /* seconds,  range 0 to 59          */
        int tm_min; /* minutes, range 0 to 59           */
        int tm_hour; /* hours, range 0 to 23            */
        int tm_mday; /* day of the month, range 1 to 31  */
        int tm_mon; /* month, range 0 to 11             */
        int tm_year; /* The number of years since 1900   */
        int tm_wday; /* day of the week, range 0 to 6    */
        int tm_yday; /* day in the year, range 0 to 365  */
        int tm_isdst; /* daylight saving time           */
};
```

- Time is handled with the defined type **time_t**
    - Size of **time_t** may **vary** from system to system
    - Should be more than 32 bits, otherwise we run into problems after 2038
    - Will probably transition to 64-bits completely by then Safe for another 293 billion years

Following are few of the functions defined in the header time.h

clock_t   clock(void)

Returns the processor clock time used since the beginning of an implementation defined era (normally the beginning of the program).

char *ctime(const time_t *timer)

Returns a string representing the **localtime** based on the argument timer.

double difftime(time_t time1, time_t time2)

Returns the **difference** of seconds between time1 and time2 (time1-time2).

struct tm *localtime(const time_t *timer)

The value of timer is broken up into the structure tm and expressed in the local time zone.

time_t   mktime(struct tm *timeptr)

Converts the structure pointed to by timeptr into a time_t value according to the local time zone.

time_t   time(time_t *timer)

Calculates the current calender time and encodes it into time_t format.

time()

Header

```
#include<time.h>
```

Prototype

```
time_t   time(time_t *tloc);
```

Call

```
time_t    thisTime;
thisTime   =   time((time_t *) 0);

printf("%d\n", thisTime);
```

- **Returns time elapsed since epoch in seconds (from January 1st 1970)**
- Can use this to measure running time for code segments

Example

```
#include<time.h>
#include<stdio.h>
int main() {

        time_t   time(time_t *tloc);

        time_t  thisTime;

        thisTime = time((time_t *) 0);

        printf("%d\n", thisTime);

}
```

Sample Run

```
192:C_Programs husaingholoom$ ./time.o

1603833273


192:C_Programs husaingholoom$
```

Example

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

int main() {

        time_t seconds;   //seconds – This is the pointer
                                        // to an object of type time_t,
                                        // where  the seconds value will
                                        // be stored

        seconds = time(NULL );


        printf("Hours since January 1, 1970 = %ld\n",
                    seconds / 3600);


        return (0);
}
```

Sample Run

Hours since January 1, 1970 = 445509

gettimeofday()

Header

```
#include<sys/time.h>
```

Prototype

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Call

```
struct  timeval       curtime;
struct  timezone     zone;
int errcode = gettimeofday(curtime, zone);
printf("%d\n", curtime.tv_sec);
printf("%d\n", curtime.tv_usec);
```

- **time()** has been deprecated by **gettimeofday()**
- **Provides both seconds and microseconds**
  - Better accuracy for performance measurement
- **Also provides time zone info**

Example : gettimeofday with time in milliseconds

```
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main(void) {

        char buffer[30];
        struct timeval tv;

        time_t curtime;

        gettimeofday(&tv, NULL);
        curtime = tv.tv_sec;

        strftime(buffer, 30, "%m-%d-%Y  %T.",
          localtime(&curtime));
        printf("%s%ld\n", buffer, tv.tv_usec);


        return 0;

}
```

04-03-2021 12:22:08.583055

gmtime() : Greenwich Mean Time (GMT)

Header

```
#include<time.h>
```

Prototype

```
struct tm* gmtime(const time_t * timeval);
```

Call

```
struct tm*    tm_ptr;
time_t     thisTime;
thisTime  =  time((time_t *) 0);
tm_ptr =  gmtime(&thisTime);
```

- Returns time in years, months, days etc
  - Returns GMT
  - Use **localtime()** to get local time

Example : gmtime()

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

#define BST (+1)
#define CCT (+8)

int main() {

        time_t rawtime;
        struct tm *info;

        time(&rawtime);
        /* Get GMT time */
        info = gmtime(&rawtime);

        printf("Current world clock:\n");
        printf("London : %2d:%02d\n", (info->tm_hour +
        BST)%24, info->tm_min);
        printf("China  : %2d:%02d\n", (info->tm_hour +
       CCT)%24, info->tm_min);

        return (0);
}
```

Current world clock:
London : 18:23
China : 1:23

Example using localtime

```
/* asctime Convert tm structure to string

#include <stdio.h>
#include <time.h>     // time_t, struct
                                  //  tm, time,
                                  //  localtime */

int main() {
        time_t rawtime;
        struct tm * timeinfo;

        time(&rawtime);
        timeinfo = localtime(&rawtime);
        printf("Current local time and date:
         %s", asctime(timeinfo));

        return 0;
}
```

Current local time and date: Sat Apr 3

12:27:37 2021

mktime() : Converts a tm structure to a calendar time

Header

```
#include<sys/time.h>
```

Prototype

```
time_t mktime(struct *tm timeptr);
```

Call

```
struct tm*    tm_ptr;
time_t    thisTime;
tm_ptr = gmtime(&thisTime);
time_t rawtime = mktime(tm_ptr);
```

Example

```
/* mktime example: weekday calculator */

#include <stdio.h>
#include <time.h>      /* time_t, struct tm, time, mktime */

int main ()
{
  time_t rawtime;
  struct tm * timeinfo;
  int year, month ,day;
  const char * weekday[] = { "Sunday", "Monday",
                             "Tuesday", "Wednesday",
                             "Thursday", "Friday", "Saturday"};

  /* prompt user for date */
  printf ("Enter year: "); scanf ("%d",&year);
  printf ("Enter month: "); scanf ("%d",&month);
  printf ("Enter day: "); scanf ("%d",&day);

  /* get current timeinfo and modify it to the user's choice */
  time ( &rawtime );
  timeinfo = localtime ( &rawtime );
  timeinfo->tm_year = year - 1900;
  timeinfo->tm_mon = month - 1;
  timeinfo->tm_mday = day;

  /* call mktime: timeinfo->tm_wday will be set */
  mktime ( timeinfo );

  printf ("That day is a %s.\n", weekday[timeinfo->tm_wday]);

  return 0;
}
```

Sample Run

```
Enter year: 2021
Enter month: 4
Enter day: 3
That day is a Saturday .
```

Accessing Process Information

Frequently used

- **getpid();** // returns the process ID of the calling process.

- **getppid();** // returns the process ID of the parent of the calling process.

- **getpriority();** // call returns the highest priority

- **setpriority();** // sets the priorities of all of the specified .

Less frequently used

- **setpgid(); //** sets the process group ID
- **getpgid(); //** returns the process group ID

setpgid() sets the process group ID of the process specified by *pid* to *pgid*.

If *pid* is zero, the process ID of the current process is used.

If *pgid* is zero, the process ID of the process specified by *pid* is used.

getpgid() returns the process group ID of the process specified by *pid*.

If *pid* is zero, the process ID of the current process is used.

getpid() : Returns the process ID of the calling process.

Header

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

Prototype

```
pid_t getpid();
```

Call

```
pid_t pid;
pid = getpid();
```

- On a Unix system every process has a numeric id
- **getpid()** returns the id associated with the current process
- At any point in time, process ids are unique

Note : Output(Will be different on different systems):

Example : getpid()

```
// C++ Code to demonstrate getpid()

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

int main(void) {
        //variable to store calling function's process id
        pid_t process_id;

        //getpid() - will return process id of calling function
        process_id = getpid();

        //printing the process id
        printf("The process id: %d\n", process_id);

        return 0;
}
```

Sampe Run

```
husain-gholooms-macbook:~ husaingholoom$  ./a.out
The process id: 883
husain-gholooms-macbook:~ husaingholoom$  ./a.out
The process id: 884
husain-gholooms-macbook:~ husaingholoom$  ./a.out
The process id: 885
husain-gholooms-macbook:~ husaingholoom$
```

getppid() : returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

Header

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

Prototype

```
pid_t getppid();
```

Call

```
pid_t ppid;
pid = getppid();
```

Note : Output(Will be different on different systems):

Example :

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

int main(void) {

        //variable to store calling function's process id
        pid_t process_id;
        //variable to store parent function's process id
        pid_t p_process_id;

        //getpid() - will return process id of calling function
        process_id = getpid();
        //getppid() - will return process id of parent function
        p_process_id = getppid();

        //printing the process ids
        printf("The process id: %d\n", process_id);
        printf("The process id of parent function: %d\n",
       p_process_id);

        return 0;
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out
The process id: 898
The process id of parent function: 842

husain-gholooms-macbook:~ husaingholoom$ ./a.out
The process id: 900
The process id of parent function: 842

husain-gholooms-macbook:~ husaingholoom$ ./a.out
The process id: 901
The process id of parent function: 842

getpriority()

Header

#include<sys/resource.h>

Prototype

int getpriority(int which, id_t who);

Call

getpriority(PRIO_USER, getuid());

- **Returns the priority level of a process**
- Each process on a Linux system runs with a priority
  - value ranges between -20 to 20
  - the higher the numeric value the lower the priority

getpriority()

int getpriority(int which, id_t who);

- First argument says what the second argument is
  - If **first** argument is **PRIO_PROCESS** then **second** argument is treated as a **pid**
  - If **first** argument is **PRIO_PGRP** then **second** argument is treated as a **pgid**
  - If **first** argument is **PRIO_USER** then **second** argument is treated as **uid**

Examples

```
int priority;
priority = getpriority(PRIO_USER, getuid());
priority = getpriority(PRIO_PROCESS, 17);
```

Example :

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

int main(void) {
        pid_t process_id;
        process_id = getpid();
        printf("The process id: %d\n", process_id);

        int which = PRIO_PROCESS;
        id_t pid;
        int ret;

        pid = getpid();
        ret = getpriority(which, pid);
        printf("\n\nThe priority is: %d\n", ret);

        return 0;
}
```

The process id: 2449

The priority is: 0

setpriority()

Header

```
#include<sys/resource.h>
```

Prototype

```
int setpriority(int which, id_t who, int prio);
```

Call

```
setpriority(PRIO_USER, getuid(), 0);
```

- **Sets the priority level of a process – it is a program scheduling priority**
- If you are not root you can only decrease the priority of your process

Example :

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

int main(void) {
        pid_t process_id;
        process_id = getpid();
        printf("The process id: %d\n", process_id);

        int which = PRIO_PROCESS;
        id_t pid;
        int ret;

        int priority = -1;

        pid = getpid();
        ret = getpriority(which, pid);
        printf("The priority is: %d\n", ret);
        ret = setpriority(which, pid, priority);
        printf("The priority is: %d\n", ret);

        return 0;
}
```

The process id: 2417
The priority is: 0
The priority is: -1

setpgid() & getpgid() :

Sets the process group ID of the process specified by pid to pgid and

Returns the process group ID of the process specified by pid

Example

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

int main(void) {
        if (setpgid(getpid(), 0) == -1) {
                perror("setpgid");
        }
        printf("%d belongs to process group %d\n", getpid(),
          getpgrp());

        return EXIT_SUCCESS;
}
```

2662 belongs to process group 2662

Files and Filesystem

- Files are bytes of data stored on the disk
  - Divided in 'blocks'
  - Blocks may not be contiguous
    - Almost never have to worry about that

- Filesystem is an organization of data and metadata on a storage device
  - Where should files be stored ?
  - How do we find a file ?

- **Inode** a key component of the Unix filesystem
  - **Stores basic information about files**

File Types

A file can be a ***regular*** file

- **Text**

  - If we look at a random byte, we can map the contents to a character in the ASCII table
  - e.g., value 17 stored using 8 bytes, first 4 bytes has ASCII value for '1', second 4 bytes has ASCII value of '7'

    00000000 00000000 00000000 00000001
    00000000 00000000 00000000 00000111

- **Binary**

    - Contents are encoded
    - Random byte may or not map to an ASCII character
        - Even if it does map to a character, it does not represent the data stored in that byte
    - e.g., value 17 stored in 4 bytes

      00000000 00000000 00000000 00010001

Directory

- On a Unix system a directory is also a file
    - bytes of data stored on the disk

- A directory is a special file
    - holds data about files contained within the directory
    - I/O interface is different from a regular file
        - Cannot use the same functions as we would to read and write from a regular file

- Some commands will distinguish between directories and regular files (**cat**), others will not (**less**)

Devices

- Devices are often mapped to files
    - cdrom, printer, terminal
    - does not contain data
      $ ls -l /dev/
    - can write to devices
        - reading is less common

- A special device
    - **/dev/null** : eats up any data sent to it
    - useful when you **don’t** want output of a command to appear on the screen

$ find / -name “hello” 2> /dev/null

Links

- A special **file** that contains a reference to another file or directory in the form of an absolute or relative path
- Unix supports two types of links

- **Soft links or symbolic links**

    - shares the same data but has different inodes
    - deleting original makes link a dangling pointer

    $ ln –s <file> <linkname>

- **Hard links**

    - shares the same inode, same data
    - deleting original makes the link the "owner" of the file

    $ ln <file> <linkname>

- Hard links cannot point to directories or files in other volumes

- Links do not store any data other than path information

- The path information is not stored on disk but rather in the ***inode (index node)***

- Links can lead to some security issues

Example (Soft / Hard Links) :

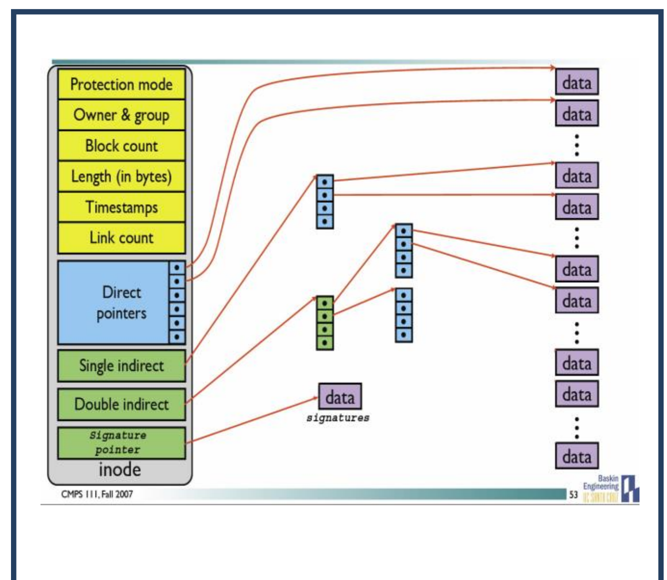
```
[hag10@zeus ~]$  ln    -s   hello.c   h.c    //  Soft link
[hag10@zeus ~]$  ln     h.c    h2.c         //  hard   link
[hag10@zeus ~]$ cat h.c
[hag10@zeus ~]$ cat h2.c
```

inode

- A data structure that stores file **metadata**
    - The main component in a Unix file system
    - The file system contains an array of inodes
    - An inode for each file on the file system
        - Directories and device files also have an inode
- The inode array resides on disk
    - Loaded to memory by kernel
        - **struct inode**

inode Entry

- An *inode* contains
    - file type
    - file access permission
    - file size
    - pointers to file's data blocks
    - number of hard links
    - timestamp



- *inode* does **not contain** the file name
    - File name is stored in the directory

- To view *inode* numbers of files, type

  $ ls –i

Example : **$ ls -i**

742913 Desktop
742915 Documents
742918 Downloads
1458408 HelloWord
17881583 ei
17587763 f1
17842567 ff
17585514 file
17917283 file.txt
17542876 file_one
17542895 file_tow
17542907 file_two
.
.
.

Example : **$ ls -i my.txt**

17532574 my.txt

Advantages and Disadvantages of inode

- This underlying organization of files and directories provides the N-ary **tree-like** directory structure on Unix Systems
- Using a tree data structure, although more intuitive, would be much more **expensive**
  - O(logn) <-> O(1)
- **Drawback** : number of files on the system **limited** by the size of the *inode* number
  - Theoretically possible to exceed this limit while disk is not full

File Descriptor

- Mechanism for the kernel to **keep track of all open files**

- Each open file on the system has a file descriptor associated with it

- The kernel maintains a data structure for each file descriptor that includes

    - File status (read, write etc.)
    - File offset
    - File size
    - Pointer to the i-node

Examining File Attributes

Header

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

Prototype

```
int stat (const char *filename, struct stat *fileinfo);
```

Call

```
struct   stat   fileinfo;
stat("foo", &fileinfo);
```

- The **stat function returns information about the attributes of the file named** by filename in the structure pointed to by fileinfo
- If filename is the name of a **symbolic link**, the attributes you get describe the file that the link points to. (**A symbolic link**, also termed a soft link, is a special kind of file that points to another file, much like a shortcut in Windows)
- If the link points to a **nonexistent file name**, then stat fails reporting a nonexistent file

File Attributes

- When you read the attributes of a file, they come back in a structure called struct stat
  - **names of the attributes**
  - **data types**
  - their meanings

#include <sys/stat.h>

```
struct stat {
    dev_t       st_dev;
    ino_t       st_ino;
    mode_t      st_mode;
    nlink_t     st_nlink;
    uid_t       st_uid;
    gid_t       st_gid;
    dev_t       st_rdev;
    off_t       st_size;
    blksize_t   st_blksize;
    blkcnt_t    st_blocks;
    time_t      st_atime;
    time_t      st_mtime;
    time_t      st_ctime;
};
```

- **dev_t st_dev**
  - ID of device containing file
- **ino_t st_ino**
  - File i-node number
- **mode_t st_mode**
  - File type information and the file **permission** bits Information is encoded
- **nlink_t st_nlink**
  - Number of hard links to the file
  - Symbolic links (soft links) are not counted here
- **uid_t st_uid**
  - User ID of the file's owner
  - Note difference from **getuid()**
- **gid_t st_gid**
  - Group ID of the file
  - Note difference from **getgid()**
- **dev_t st_rdev**
  - **device ID (if special file)**
- **off_t st_size**
  - Size of a regular file in bytes.

- **blkcnt_t st_blocks**
  - Amount of disk space that the file occupies, measured in units of (512 byte) blocks.

- **blksize_t st_blksize**
  - blocksize for filesystem I/O

- **time_t st_atime**
  - Last access time for the file

- **time_t st_mtime**
  - Last modification time to the contents of the file

- **time_t st_ctime**
  - Last modification time to the attributes of the file

Retrieving Info from st_mode

- **st_mode** stores multiple pieces of information
    - File type and access information for user, group and other

- Need to use bit operations to retrieve the encoded information

- Pre-defined macros available for checking file type

    - **int S_ISREG (mode_t m)**
        - Returns non-zero if the file is a **regular file**.

    - **int S_ISCHR (mode_t m)**
        - Returns non-zero if the file is a **character special file (a device like a terminal)**.

    - **int S_ISBLK (mode_t m)**
        - Returns non-zero if the file is a **block special file (device like a disk).**

    - **int S_ISDIR (mode_t m)**
        - Return non-zero if the **file is a directory.**

    - **int S_ISLNK (mode_t m)**
        - Returns non-zero if the file is a **symbolic link.**

    - **int S_ISSOCK (mode_t m)**
        - Returns non-zero if the file is a **socket**

Checking File Type in st_mode

- Example

```
struct   stat  fileinfo;
stat("foo", &fileinfo);

if (S_ISDIR(fileinfo.st_mode))
  printf("directory");
```

Examining File Attributes

Header

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

Prototype

```
int  fstat (int filedes, struct stat  *fileinfo);
```

Call

```
struct  stat   fileinfo;
fstat(STDOUT_FILENO, &fileinfo);
```

- The **fstat()** function is like **stat()**, except that it takes an **open file descriptor** as an argument instead of a file name

Examining File Attributes

Header

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

Prototype

```
int lstat (const char *filename, struct stat *fileinfo);
```

Call

```
struct  stat  fileinfo;
lstat(“foo”, &fileinfo);
```

- The **lstat()** function is like stat, except that **it does not** follow symbolic links
- If filename is the name of a symbolic link, **lstat()** returns information about the link itself; otherwise **lstat()** works like **stat()**
- Information returned for symbolic links
  - Type, size and link count
- Permission and access info is derived from parent directory

Example

```
//   A Program to Print some general file info

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

int main(int argc, char *argv[]) {
 struct stat file_stats;

 if(argc != 2)
     fprintf(stderr, "Usage: fstat FILE...\n"), exit(EXIT_FAILURE);

 if((stat(argv[1], &file_stats)) == -1) {
      perror("fstat");
     return 1;
 }
printf("Information for %s\n",argv[1]);
printf("---------------------------\n");
 printf("filename: %s\n", argv[1]);
 printf(" device: %lld\n",                   file_stats.st_dev);
 printf(" inode: %ld\n",                     file_stats.st_ino);
 printf(" protection: %o\n",                 file_stats.st_mode);
 printf(" number of hard links: %d\n",       file_stats.st_nlink);
 printf(" user ID of owner: %d\n",           file_stats.st_uid);
 printf(" group ID of owner: %d\n",          file_stats.st_gid);
 printf(" device type (if inode device): %lld\n",file_stats.st_rdev);
 printf(" total size, in bytes: %ld\n",      file_stats.st_size);
 printf(" blocksize for filesystem I/O: %ld\n",  file_stats.st_blksize);
 printf(" number of blocks allocated: %ld\n",    file_stats.st_blocks);
 printf(" time of last access: %ld : %s",       file_stats.st_atime,
ctime(&file_stats.st_atime));
 printf(" time of last modification: %ld : %s",  file_stats.st_mtime,
ctime(&file_stats.st_mtime));
 printf(" time of last change: %ld : %s",       file_stats.st_ctime,
ctime(&file_stats.st_ctime));

 return 0;      }
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out file.txt

Information for file.txt
----
filename: file.txt
device: 234881026
inode: 17917283
protection: 100700
number of hard links: 1
user ID of owner: 501
group ID of owner: 20
device type (if inode device): 0
total size, in bytes: 32
blocksize for filesystem I/O: 4096
number of blocks allocated: 8
time of last access: 1571503051 : Sat Oct 19 11:37:31 2019
time of last modification: 1571503136 : Sat Oct 19 11:38:56 2019
time of last change: 1571503136 : Sat Oct 19 11:38:56 2019
husain-gholooms-macbook:~ husaingholoom$

Checking File Permission in st_mode

- File permission masks
    - **S_IRWXU /* user read-write-execute mask */**
    - **S_IRWXG /* group read-write-execute mask */**
    - **S_IRWXO /* other read-write-execute mask */**
- Used to interpret file permission info in **st_mode**
- Symbolic names for file permission bits
    - **S_IRUSR user read**
    - **S_IWUSR user write**
    - **S_IXUSR user exec**
    - **S_IRGRP group read**
    - **S_IWGRP group write**
    - **S_IXGRP group exec**
    - **S_IROTH other read**
    - **S_IWOTH other write**
    - **S_IXOTH other exec**

Modifying File Permissions

- Header

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

- Prototype

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

- Call

```
mode_t  mode = S_IRUSR | S_IWUSR;
chmod("foo", mode);
```

- **chmod()** allows us **to modify permission bits** for a file
- To change file permissions the effective user id must be the same as the owner of the file
- Can set file permissions using absolute mode value or through bit manipulation of current **st_mode**

Modifying File Permissions

- Header

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

- Prototype

```
int fchmod (int filedes, int mode);
```

- Call

```
mode_t mode  =  S_IRUSR  |  S_IWUSR;
fchmod(STDOUT_FILENO, mode);
```

- Functionally exactly like **chmod()**
- Takes a file descriptor as an argument rather than a file name
- Passing on a file descriptor implies the file is currently open

Checking File Permission

Header

#include <unistd.h>

Prototype

int access (const char *filename, int how);

- Check whether the file can be accessed in the way specified by the how argument.
- Uses the real user and group IDs of the calling process to check for access permission
- The how argument can be a bitwise OR of the flags
  - **R_OK**
  - **W_OK**
  - **X_OK**
- Can also do an existence test with **F_OK**
- Useful as a check before calling **open()**

File Ownership

Header

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

Prototype

```
int chown (const char *filename, uid_t owner, gid_t
          group);
int fchown (int  filedes, int owner, int group);
```

- The **chown() function changes the owner** of the file filename to owner, and its group owner to group
- Usually useful **only** for root
- Privileged processes can use **chown()** to give ownership of files to other users
  - Can't get them back
- **fchown()**is like **chown()**, except that it changes the owner of the **open** file with descriptor filedes

File Size

Header

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

Prototype

```
int  truncate (const  char *filename, off_t length);
int  ftruncate (int  fd,  off_t length);
```

- The function **changes the size of filename to length**.
- If length is **shorter** than the previous length (typically the case), data at the end will be lost
- On some systems can use truncate to extend the file size
  - empty blocks added to end of file

Example

```
#include <stdio.h>
#include <sys/stat.h>
int main()
{
   struct   stat   stat_result;
   char filename[]= "foo2";
   if ( stat(filename, &stat_result) == -1 )
      return 1;
   printf("Size of \"%s\" is %ld bytes\n", filename, stat_result.st_size);
   return 0;
}
```

Sample Run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out

Size of "foo2" is 218 bytes

husain-gholooms-macbook:~ husaingholoom$
```

Rename File

Header

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

Prototype

```
int rename (const char   *oldname,
                    const char *newname)
```

- The rename function **renames** the file oldname to newname
- If oldname is not a directory, then any existing file named newname is removed during the renaming operation
- if newname is the name of a directory, rename fails
- If oldname is a directory, then either newname must not exist or it must name a directory that is empty

Example

```
#include <stdio.h>

int main() {
        int ret;
        char oldname[] = "foo1.txt";
        char newname[] = "foo2.txt";

        ret = rename(oldname, newname);
        if (ret == 0) {
                printf("File renamed successfully");
        } else {
                printf("Error: unable to rename the file");
        }

        return (0);
}
```

Sample Run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out
File renamed successfully

husain-gholooms-macbook:~ husaingholoom$
```

Delete File

Header

#include<stdio.h>

Prototype

int remove (const char   *filename)

- Delete a file from the system

Example

```
#include <stdio.h>

int main() {
        int status;
        char file_name[25];

        printf("Enter the name of file you wish to
          delete\n");
        gets(file_name);

        status = remove(file_name);

        if (status == 0)
                printf("%s file deleted
            successfully.\n", file_name);
        else {
                printf("Unable to delete the file\n");
                perror("Error");
        }

        return 0;
}
```

Sample Run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out
Enter the name of file you wish to delete
warning: this program uses gets(), which is unsafe.
foo2
Unable to delete the file
Error: No such file or directory
husain-gholooms-macbook:~ husaingholoom$ ./a.out
```

Sample Run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out
Enter the name of file you wish to delete
warning: this program uses gets(), which is unsafe.
zzzz.c
zzzz.c file deleted successfully.
husain-gholooms-macbook:~ husaingholoom$
```

Command Line Parameters: Arguments to main() in C

Accessing the command line arguments is a very useful facility. **It enables you to provide commands with arguments** that the command can use e.g. the command

$ cat prog.c

takes the argument "prog.c" and opens a file with that name, which it then displays. The command line arguments include the **command name itself** so that in the above example, "**cat**" and **"prog.c"** are the **command line arguments**. The first argument i.e. "cat" is argument number zero, the next argument, "prog.c", is argument number one and so on.

To access these arguments from within a C program, **you pass parameters to the function main()**. The use of arguments to main is a key feature of many C programs.

The declaration of main looks like this:

int main (int argc, char *argv[])

This declaration states that

1. **main returns an integer value (used to determine if the program terminates successfully)**
2. **argc is the number of command line arguments including the command itself i.e argc must be at least 1**
3. **argv is an array of the command line arguments**

The declaration of argv means that it is an array of pointers to strings (the command line arguments). By the normal rules about arguments whose type is array, what actually gets passed to main is the address of the first element of the array.

Example : test.c echoes its arguments to the standard output – is a form of the Unix echo command. For example ,

husain-gholooms-macbook:~ husaingholoom$ echo Texas State University

Texas State University

husain-gholooms-macbook:~ husaingholoom$

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

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

      num_args = argc ;

      while( num_args > 0)
      {
            printf("%s\n", argv[i]);
            i++ ;
            num_args--;
      }
}
```

Sample Run :

husain-gholooms-macbook:~ husaingholoom$ ./a.out hello goodbye solong bye now
./a.out
hello
goodbye
solong
bye
now
husain-ghol