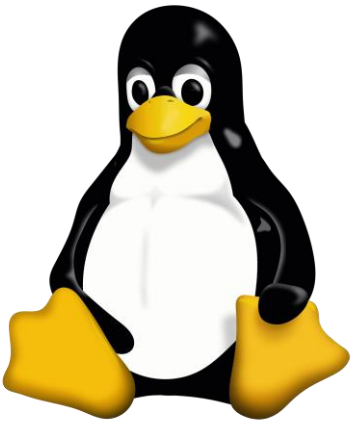
Documentation for the work done in

Crimson Innovative Technologies Pvt. Ltd.



Record work of Trainee **B. Balaji Ganesh**

Industrial Training

Where we learnt skills those meet actual needs..

Record of

BODA BALAJI GANESH

17001-CM-008

For the training done in
Crimson Innovative Technologies (CIT) Pvt. Ltd.

Crimson Innovative technologies Team

N. Madhu Sir (B.Tech ECE)

K. Swarnakanth Sir(B. Tech CSE)

Y. Shiva Kumar Sir (B.Tech ECE)

College staff

R. Bharath Kumar Sir (M.Tech CSE)

J.Govardhan Reddy Sir (M.Tech CSE)

Section -1

Linux Programming with C

1. Programmers text editor – ViM

2. Compiling a C program with GCC – Behind the Scenes

3. Compilation Phases of a Program (High level view ~ Abstract view)

4. Libraries ~ Static and Dynamic Libraries (in C)

5. Dealing with Processes in Linux

6. Behind the Internet Communication -- Sockets

8. Communication between the processes in a system -- IPC mechanisms

Coming up.... next...

Section -2

GPU Architectures and GPU Virtualization

- Early GPU architectures
- CUDA GPU Architecture
- GvirtuS – GPU Virtualization Service across Cross platforms

Section-3

Data Science

- Python
- Python Core Libraries
  - OS library
  - And other..
- Python libraries by Community
  - numpy --- mathematical calculations
  - matplotlib – visualizing graphs
  - plotly – advanced library for dealing with various types of graphs
  - opencv – open source computer vision libraries
  - .....
- Machine Learning Libraries for Python
  - Scikit-learn
  - Keras
  - Tensorflow
  - Theano
- ...
- R language

Programmers Text Editor: ViM Editor

Getting started with Vi editor

Vi editor is basically a common default text editor for the UNIX based systems. Unlike normal Word processors it doesn't have any menus to get our work done, instead commands (set of keyboard strokes) are used to perform an operation like how we perform an action by selecting a menu-option in the menu like in the menu-based graphical word-processors.

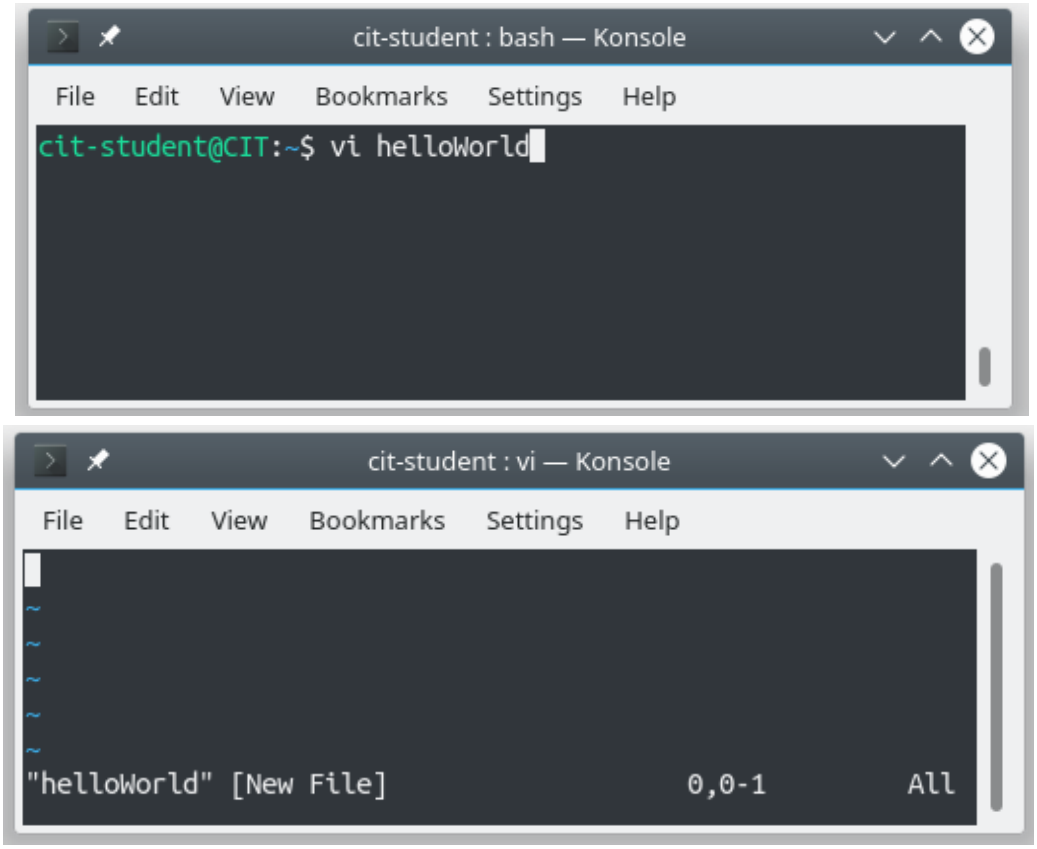
Opening a file

Type `vi [filename[.extension]]` on the command line to perform either action.

Open the file if it already exists.

Create a new file.

Ex:



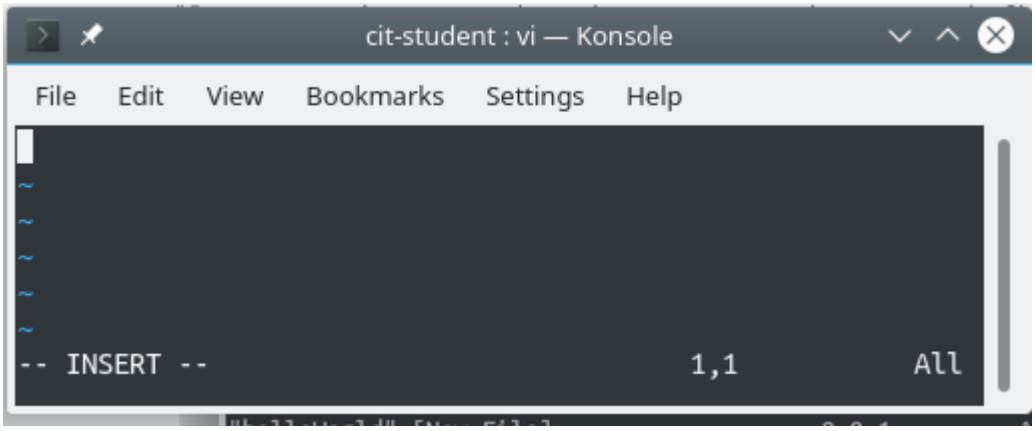
Vi editor basically works in 2 modes:

i. Insert mode

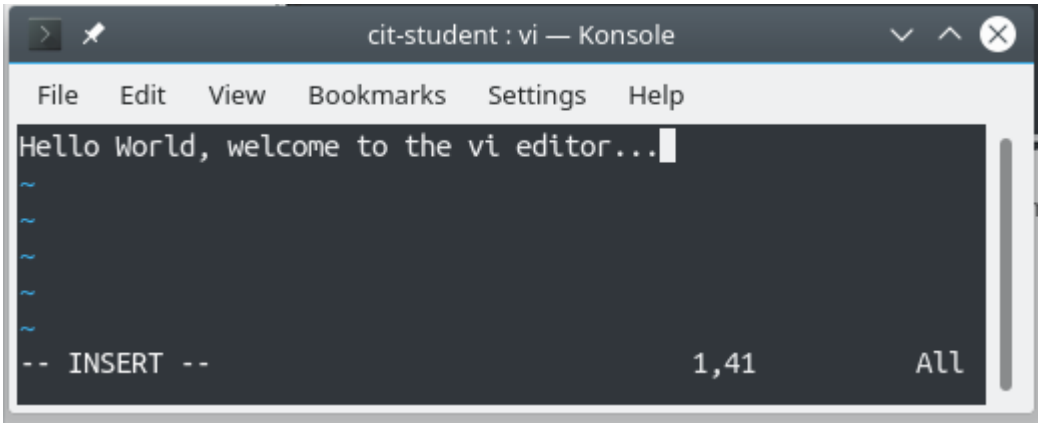
ii. Command mode

i. Insert Mode

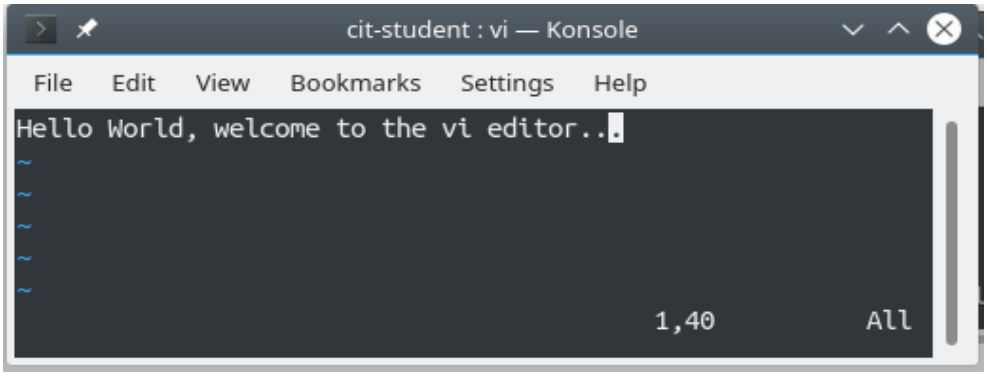
By default the vi editor will be opened in the command mode, press “Insert” key or press “i” to enter into the Insert mode, so that we can type/edit our text in the file. In this Mode we can enter our required text in the document which we’ve opened via vim.



Now we can see the mode of the editor at bottom-left. Now we can type the text we desired.



Again to come out of the Insert mode, press the “Esc” key.



ii. Command Mode

For the ease of understanding and remembering the commands are grouped together based on the action they perform, as followed:

NOTE: All the commands are case-sensitive.

1. Cursor Movement

○ Moving one space at time

Sometimes in the editor, if the arrow keys doesn't work, then these keys can be used alternatively for that, those are:

- `h` → To move one space left
- `j` → To move one space down
- `k` → To move one space right
- `l` → To move one space top

○ Moving among words and lines

`w` → Moves the cursor one word forward

`b` → Moves the cursor one word backward (if in the middle of the word, then goes of the beginning of that word)

`e` → Moves to the end of the word

○ Moving along the line

`0` → Moves the cursor to the beginning of the line (same as "Home" key).

`$` → Moves the cursor to the end of the line (same as "End" key).

○ Screen Movement

- `ctrl + f` → Scrolls down one screen
- `ctrl + b` → Scrolls up one screen
- `ctrl + u` → Scrolls up one half screen
- `ctrl + d` → Scrolls down one half screen

○ Scrolling based on Search

Type "/" and search the string to be searched, then press enter. Automatically the cursor will be moved to the next occurrence of the string.

- `n` → to move to the next occurrence.
- `N` → to move to the previous occurrence.

2. Basic Editing (Copy - Paste - Cut (Delete))

a. Copy (Yank)

`yw` → To copy a word (from the current position in the word to the end).
`yy` → To copy the entire line.

b. Paste

`p` → To paste from the next(or below) of the cursor.
`P` → To paste from the left (or above) of the cursor.

c. Cut (Delete)

`x` → Deletes the character under the cursor.
`X` → Deletes the character to the left of the cursor.
`dw` → Deletes the word from the current character to the end of the word.
`dd` → Deletes the entire line.
`D` → Deletes all the characters from the current character to the end of the line.

3. Replacing

`r` → To replace the current character with the next character that we type.

`R` → To replace each and every character with the character that we type *(also known as changing into Replace mode).*

`/` → Used to replace each and every word that matches the search
(Extension to the Search).

Ex: `/hello/world`

This will replace the next occurance of “hello” with the “world”. To continue replacing the next word, type “n” to replace the next occurance, type “N” to replace the previous occurrence.

4. Undo

`u` → undo the last change made to the document. *(The change can be anywhere in the file).*
`U` → Undo the last change made to the current line.

5. Inserting Empty lines

`o` → To insert the empty line below the cursor.

`O` → To insert the empty line above the cursor.

NOTE: After this command execution, the mode will be immediately changed to Insert mode.

6. Closing and Saving files

`:w` → To save the file. (Doesn't Quit the file)

`:q` → To quit (Without making any changes to the document) when opened read-only.

`:wq` → Save the file before quitting.

Simple Shortcut: type "ZZ" (Capital Z's) to save and quit.

`:q!` : Exits without making any changes to the file when opened for editing.

7. Combining commands, objects and numbers

General format of combining:

```
(number) (command) (text object)
                or
(command) (number) (text object)
```

a. Combining commands with numbers

Basically all the commands which are explained above will work only on either obe word or one line or one character. This feature can be extended by just prefixing the number before the command.

Ex-1: To delete the 5 words from the current cursor position:

"`5dw`" will delete the next 5 words from the current cursor position.

Ex-2: To copy 5 lines

"`5yy`" or "`y5y`" will copy the next 5 lines from the cursor.

Ex-3: to copy 5 Words

"`5yw`" or "`y5w`" will copy 5 words from the current word.

Ex-4: To delete 10 lines from the current line

"`10dd`" or "`d10d`" will delete the 10 lines from the current line.

b. Combining command with objects

Commands	Objects
d *(delete)*	**w** *(word to the left)*
y *(yank/copy)*	**b** *(word to the right or backward)*
c *(change)*	**e** *(end of word)*
	H *(top of the screen)*
	L *(bottom of the screen)*
	M *(middle of the screen)*
	0 *(zero - first character on a line)*
	$ *(end of a line)*
	((previous sentence)
	) *(next sentence)*
	[*(previous section)*
	] *(next section)*

Ex-1: To delete till the end of the line from current position:

"`d$`" deletes till the end of line from the current position.

Ex-2: To delete 5 lines from the current position of the cursor.

(Just by adding number to above cmd)

"`5d$`" deletes the 5 lines starting from the current cursor position.

Additional Vi commands

- ## Repeating the command (.)

  Just type the dot (`.`) to repeat the basic last command issued.

- ## Jump to line

  Type "`:[n]`" to jump to the nth line.

- ## Undo the last issued command

  Type "`:u`" to undo the last issued command.

- ## Cut/Paste Commands:

  `"[a-z]nyy` Yank next n lines into the named buffer [a-z] .

  `"[a-z]p/P` Place the contents of the selected buffer below/above the current line.

- ## Extensions to the delete, copy and move

  `:3,18d` Delete lines 3 through 18.

  `:16,25m30` Move lines 16 through 25 to after line 30.

  `:23,29co62` Copy lines 23 to 29 and place after line 62.

- ## Searching and Substitution commands:

  `/ [string]` Search forward for string

  `? [string]` Search backwards for string

  `:1,$s/s1/s2/g` Global replacement of string1(s1) with string2(s2).

- ## Customizability and Extensions

  `:set number` To set the line numbers Fancy Stuff:

  `:1,10w file` Write lines 1 through 10 to file newfile

  `:340,$w >> file` Write lines 340 through the end of the file and append to file new file.

`:sh`	Escape temporarily to a shell
`^d`	Return from shell to VI
`:![command]`	Execute UNIX command without leaving VI
`:r![command]`	Read output of command into VI
`:r[filename]`	Read filename into VI
`:$r newfile`	Read in newfile and attach at the end of current document
`:r !sort file`	Read in contents of file after it has been passed through the UNIX sort
`:n`	Open next file (works with wildcard filenames, ex: vi file*
`:^g`	List current line number
`:set number`	Show line numbers
`:set showinsert`	Show flag ("I") at bottom of screen when in insert mode
`:set all`	Display current values of VI variables
`:set ai`	`Set autoindent;` after this enter the insert mode and tab, from this point on VI will indent each line to this location. Use ESC to stop the indentations.
`^T`	Set the auto indent tab one tab stop to the right
`^D`	Set the auto indent tab one stop to the left
`:set tabstop=n`	Sets default tab space to number n
`>>`	Shift contents of line one tab stop to the right
`<<`	Shift contents of line one tab stop to the left

References

https://www.ccsf.edu/Pub/Fac/vi.html

Compiling a C program with GCC -- Behind the Scenes

GCC stands for GNU Compiler Collection.

(Let's assume that we need to compile this simple C program, So, our first step will be writing a C program)

```
#include <stdio.h>
int main ()
{
    printf("Hello world\n");
}
```

Now save the file, and compile it as:

```
gcc  helloworld.c -o helloworld
```

And can be executed as

Now save the file, and compile it as:

```
./helloworld
```

The output would be like

```
 Hello world
```

What's happens behind the Scenes of Compilation, come let's explore it...

Unlike compiling having the required executable file directly from the source file, gcc supports a feature of stopping the compilation at each and every stage/phase.

Broadly there will be 4 stages from the conversion of the source file to the executable file, they are:

1. Preprocessing
2. Compilation
3. Assembling
4. Linking

Stage-1: Preprocessing

This phase performs:

- Removal of Comments in the program.
- Expansion of Macros.
- Expansion of Header files.
- Conditional Compilation.

To perform this step, GCC executes this command internally:

Approach-1:

```
# cpp helloworld.c > helloworld.i
```

This step can also be done through

Approach-2:

```
# gcc -E helloworld.c -o helloworld.i
```

The Result after performing either the commands is "**Preprocessed file**" (i.e., `helloworld.i`).

It would like: (Bottom of the file)

We can notice that.... Macros were expanded, comments were trimmed off, header file was expanded (here `"stdio.h"` header file) after conditional compilation.

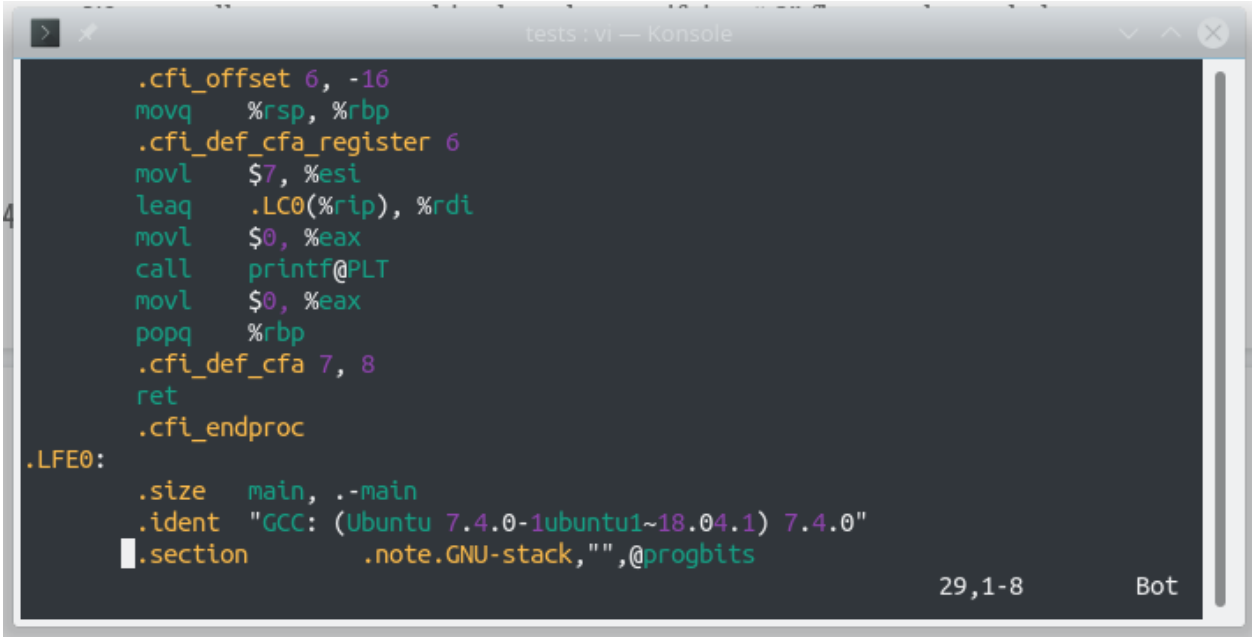
Stage-2: Compiling

This phase takes the pre-processed file as input and translates it to the **IR (Intermediate Representation) code**, which can also be called the **Assembly code.**

We can tell gcc to stop at this phase by specifying "`-S`" flag, as shown below:

```
# gcc -S helloworld.c
```

Upon execution of this command the "helloworld.s" file will be generated. It will be almost similar like..



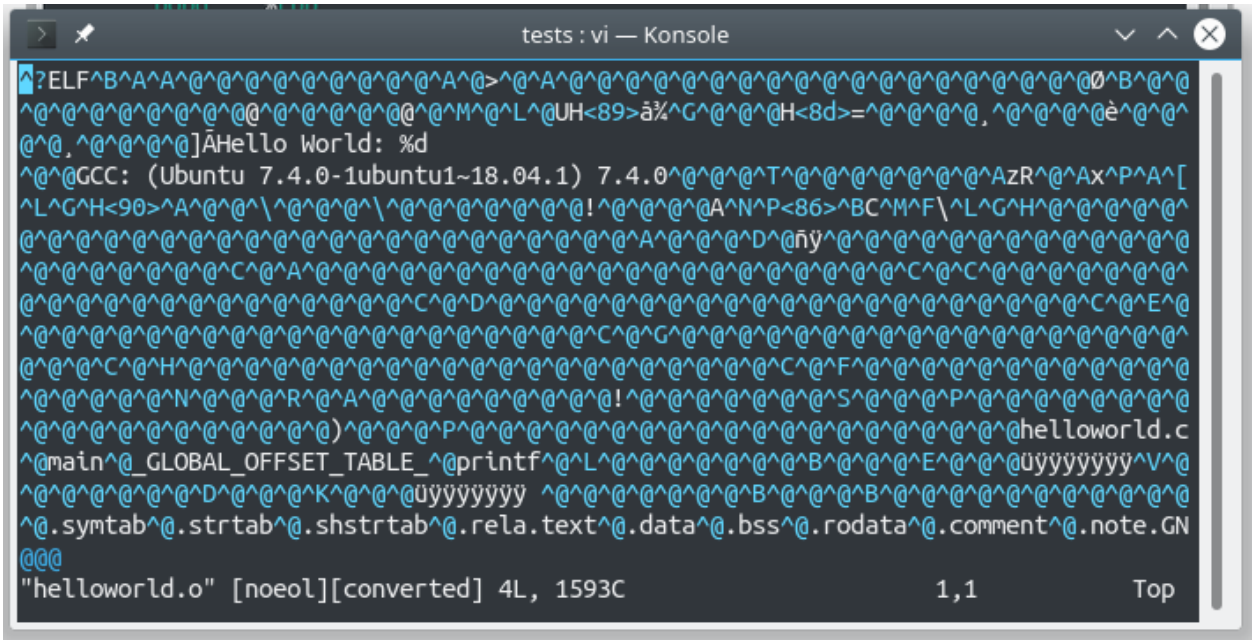
Stage-3: Assembling

- This phase takes the IR code (i.e., Assembly-language file) and transforms it into object code. i.e., to machine understandable language (binary language).
- We can tell gcc to stop at this phase by specifying "`-c`" flag as

```
# gcc -c hellworld.c
```

- This command will result in the object file as `helloworld.o` (in UNIX machines) and `helloworld.obj` (in Windows machines).

It will look like:



Stge-4: Linking

- Even though now we have got the file that the processor can understand, still it is not enough to perform the execution. That's because all the required function definitions are not yet available in the file.
- This lack is fulfilled by this phase. This phase will link all the necessary files that are required by the program to run in the form of libraries. By default gcc uses Dynamic linking to link all the required dynamic libraries.

We can tell gcc to this task from the source-file as:

```
# gcc helloworld.c
```

By default This will generate a file called executable file which will be saved with the name of `"a.out".`

Custom name can also be given through `"-o"` flag as shown below:

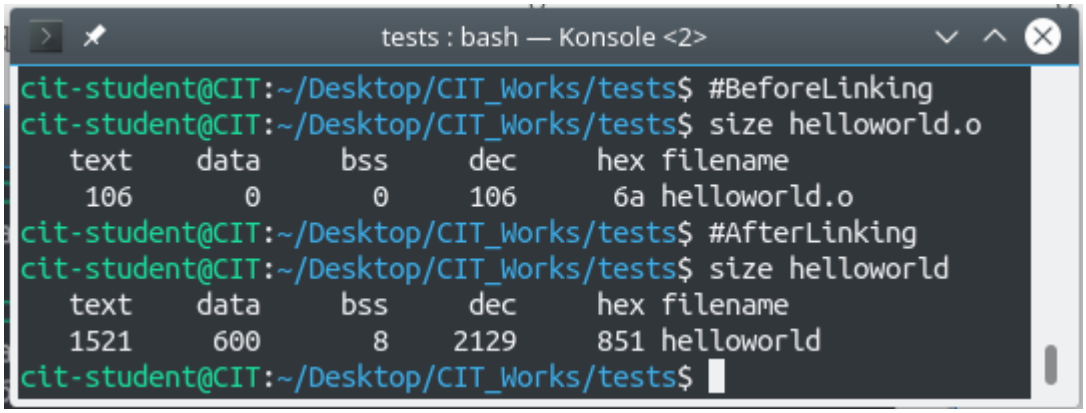
```
# gcc helloworld.c -o helloworld
```

- It's even possible to combine all the above steps with the simple flag called "-save-temps", this will perform similar like `gcc helloworld.c -o helloworld`, but this command generates the output executable file, by discarding all the intermediary temporary files `(*.i, *.s, *.o files).`

By specifying the -save-temps flag will tell the gcc to not discard the files as soon after performing the required operation. As

```
# gcc helloworld.c -save-temps –helloworld
```

The size of the file varies a lot before and after linking the file. It can be observed through “size” command



Observe the difference between the various segment values....

Miscellaneous info...

By default some of the warnings are discarded by the gcc while performing the task, if would like to show the warnings, it can be controlled through:

`-Wall`	To display all warnings (many)
`-Wpedantic`	Pedantic
`-Wextra`	Extra

We would like to show all the warnings, specify the `“-Wall”` flag (means that **W**arnings**all**) as :

```
# gcc helloworld.c -Wall -save-temps -o helloworld
```

By default gcc optimizes the code to some extent, this optimization level can be controlled by

`-O0`	None
`-O1`	Moderate
`-O2`	Full
`-O3`	Maximum

References:

https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/

https://medium.com/@laura.derohan/compiling-c-files-with-gcc-step-by-step-8e78318052

http://cs-fundamentals.com/c-programming/how-to-compile-c-program-using-gcc.php

Compilation Phases of a Program (High level view)

~ What does a compiler actually do..?? How does it perform..??

To explain theoretically, there exists a total of 6 phases, namely

1. Lexical Analyzer
2. Semantic Analyzer
3. Syntax Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Target Code Generation.

But practically, there exists only two phases (Combination of the above phases only), as

1. Frontend
2. Backend

- Frontend consists of the first 4 theoretical phases which are platform-independent.
- Backend consists of the last 2 theoretical phases which are platform-dependent.

So, if we would like to design a new compiler for a specific platform, we don't need to do it from scratch, just simply taking the readily available Frontend, and building our Backend. That's it..!!

So, let's understand the task of each and every theoretical phase briefly:

Phase-1: Lexical analyzer

This is the first phase where our High-Level language passes through. The work of this phase is -

- To read our program and identify smallest individual tokens by reading either as left-to-right or character-by-character, or in any order by using Regular Expression.
- Then classify those identified tokens into constants, variables, keywords and place them in different tables.
- As well as it will trim off the extra white spaces and comments in the code.

Ex: A normal expression in the code, will be broken down as identifiers and classify them.

Expression: `X = A + B * C * 10;`

After identifying tokens: id = id + id * id * 10. (id is short for identifier) based on RegEx.
Classifying them as constants, variables, operators as:

X, A, B, C → Identifier
10 → Constants
+, =, * → Operators

Phase-2: Syntax analyzer

- This phase gets the lexical units from the Lexical Analyzer and builds a parse tree based on these lexical-units(lexemes) and grammar of the programmer language. (This is the reason that it is also called Parser).

The Grammar(set of rules) is represented as some productions.

- The main task of this phase is:
    - Obtain the tokens from the Lexical analyzer
    - Check if the expressions are syntactically correct or not.
    - Reporting the user, if the code violates any rules of the language.
        - Ex: Modifying constants, assigning the value to a constant, etc..
    - At last construct the Hierarchical tree (Parse tree).

Phase-3: Semantic Analyzer

- This phase takes the parse tree as the input and performs.
    - Type checking on that parse tree.
        - Ex: Whether resultant expression's value is assigned to a proper compatible type or not.
    - Performs implicit conversion, if type-compatible.
        - Ex: Converting integer to float.
    - In case of type mismatch, semantic error is thrown.
        - Ex: Converting object of one class to another class.
    - And also label-checking and control checking.

And results in the Synthesized Parse tree.

Phase-4: Intermediate Code Generation:

- This phase takes the synthesized parse tree, and converts them into the Intermediate code. It can be of any Intermediate Code format like Three-Address format etc.,.
  For the example which is taken above will be converted as

```
t1 = C * 10;
t2 = B * C;
t3 =  A + B;
X = t3;
```

(It is called three-address format, as it at most needs 3 addresses to perform the execution).

Phase-5: Code Optimization

- The primary goal of this phase is --- to speed up the execution process, and take less memory without wasting CPU resources (i.e., optimization) by removing unnecessary code and arranging them in the proper sequence.

Ex: For the Intermediate code generated in the above phase is optimized as:

```
t1 = C * 10;
t2 = B * C;
X = A * B;
```

This optimization varies from platform to platform as this phase is platform-dependent. So the example's output varies from platform to platform.

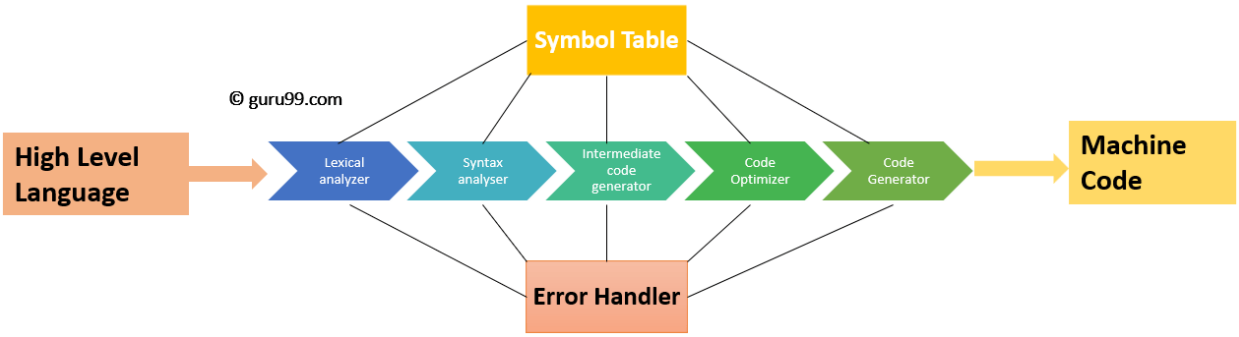
Phase-6: Target Code Generator

- The main goal of this phase is to generate the target code (i.e., code that the target platform system can run), that the code under the machine can understand and perform appropriate operations like register allocation, Instruction selection etc,.
- The output of this phase is dependent on the type of the Assembler which is platform dependent.
    - For Example, the target-code(like Assembly language) will look like:

(for 8086 platform)

```
MOV R1, AH ;    Move 10(A in Hex) to R1
MUL R1, C ;     Multiply C with R1(i.e., 10)
MUL R1, B ;    Multiply B with R1 (i.e., c * 10)
ADD R1, A ;          Add A with R1 (i.e., B * C * 10)
MOV X, R1 ;          Place Result (i.e, R1) into the X
```

Summing up...



References:

1. Compiler Design Lecture-1 By Ravindrababu Ravula Sir (GATE) -- Clear Explanation
2. https://youtu.be/Qkwj65l_96I
3. https://www.guru99.com/compiler-design-phases-of-compiler.html
4. https://www.geeksforgeeks.org/phases-of-a-compiler/

Libraries ~ Static and Dynamic Libraries (in C)

What is a library (in Linux)?

- A library is the collection of pre-compiled pieces of code called functions which are combined together, that serve a common purpose.
    - **Ex: stdio.h** is a library that contains the pre-compiled pieces of code of multiple functions that are grouped together for a common purpose i.e., Dealing with Standard Input (Keyboard) and Standard Output(Monitor).
- A key difference from the processes and a library is, a library is not an executable and processes are executable.
- We have two types of libraries
    - Static Libraries
    - Dynamic Libraries

What are Static and Dynamic libraries?

- Static and dynamic libraries are the two processes of collecting and combining multiple objects files in order to make a single executable.
- Linking can be performed at both
    - **Compile time** -- when the source code is translated into machine code(Compiler).
    - **Load time** -- when the program is loaded into memory and executed (Loader).

      And even at **runtime** *by the application programs.*

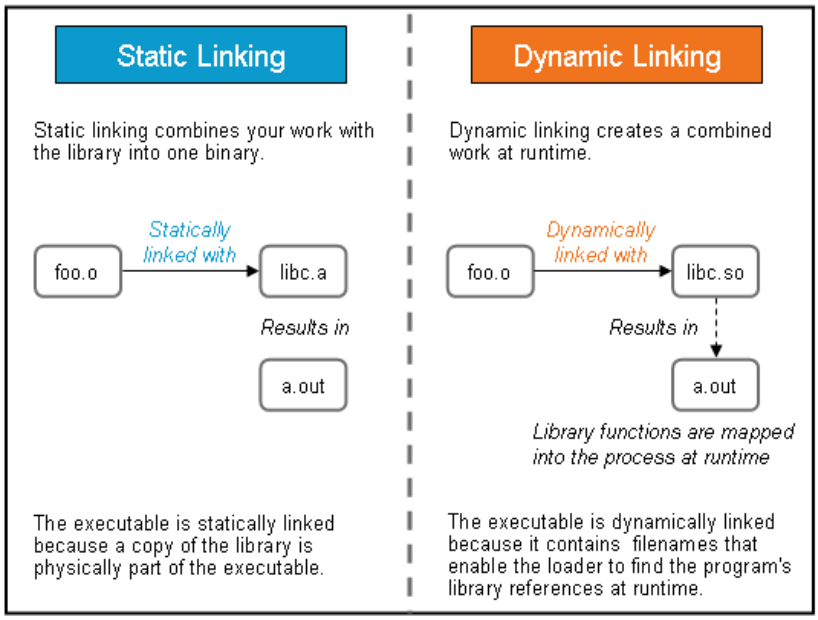
      By the **Linker** (also called Link Editor) in the last phase of compiling a program.

What is Linker?

- A linker is a program that helps in organizing a large program (source file) into some manageable chunks of source files (and even decomposing them more), that can be modified and compiled separately and link all of them together.
- The major advantage that linker provides is, whenever we modify any of the module, we just need to recompile that specific module and re-link to the application, without recompiling the rest.
- Libraries can be linked either statically or dynamically, that results as Static libraries and Dynamic libraries respectively.

Differences between Static and Dynamic Libraries

Static Libraries	Dynamic Libraries
It is the process of copying all the libraries used in the program into a final executable file which is done at the last step of compilation.	It is the process of linking the names of the shared libraries (External libraries) in the final executable file which is done at runtime. -- Actual linking takes place when both the program and the external libraries are placed in the memory.
Multiple programs use multiple copies of the library. So the **size** of programs will be **large.**	Several programs can use just the single copy of the external library. So the **size** of programs will be **small.**
In this, if any one of the programs has changed, all the programs must be re-compiled and re-linked, else changes won't take effect in the final executable file.	In this, if any shared module can be updated and recompiled at any time.
Programs that are statically linked never run into compatibility issues, as all the libraries are held together with the programs.	There is a chance of having compatibility issues with programs that are linked dynamically, as they can be changed at any time.
Static libraries have extension "`*.a`".	Dynamic libraries have extension as "`*.so`". Called as Shared Library in Linux and DLL in Windows.



Creating Static and Dynamic Libraries..

Setup:

For the sake of example let's create 3 programs as

1. A program to do Addition
2. A program to do Multiplication
3. Driver program(main program that uses the above two)

Program `addition.c`

```
int addition(int num1, int num2)
{
	return num1 + num2;
}
```

Program `multiplication.c`

```
int multiplication(int num1, int num2)
{
	return num1*num2;
}
```

A header file to include the declarations of these functions as. `basicMath.h`

```
int addition(int, int);
int multiplication(int, int);
```

And finally the driver program as `driver.c` by including the header file as:

```
#include<stdio.h>
#include<basicMath.h>
int main(int argc, char ** argv)
{
	printf("5 + 2 = ", addition(5, 2));
	printf("10 * 40 = ", multiplication(10, 40));
}
```

```
balajiganesh@helloworld:~/Desktop/C_programs$ ls
addition.c   driver.c multiplication.c
```

Now lets compile them one by one.. as

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -c addition.c
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -c multiplication.c
```

The option "`-c`", tells the GCC compiler to just compile the source file, don't proceed further. It generates files as `addition.o` and `multiplication.o` respectively when compiled.

```
balajiganesh@helloworld:~/Desktop/C_programs$ ls
addition.c  addition.o   driver.c  multiplication.c  multiplication.o
```

Now lets compile the driver.c with the header file as

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -c driver.c -I .
```

It generates the executable object file for `driver.c` as `driver.o, and` the option `-I` tells the GCC to look for the header files in the current directory(which is specified as `.` after `-I`).

And finally we have..

```
balajiganesh@helloworld:~/Desktop/C_programs$ ls
addition.c  addition.o basicMath.h driver.o  driver.c
multiplication.c  multiplication.o
```

Creating Static Library

Lets create the static library with the binary object files `addition.o` and `multiplication.o` as `libBasicMath.a` by the following command as:

```
balajiganesh@helloworld:~/Desktop/C_programs$ ar rcs libbasicmath.a
addition.o multiplication.o
```

The command `ar` meant to create, modify and extract archives, the options

- `r` --- to insert the specified files in the archive
- `s` -- to add an index to the archive, or update if already exists.
- `c` -- create the archive, if it doesn't exist.

This command generates the static library `libbasicmath.a` that consists of `addition.o` and `multiplication.o`

```
balajiganesh@helloworld:~/Desktop/C_programs$ ls
addition.c  basicMath.h  driver.c    libbasicmath.a   multiplication.o
addition.o  driver.o   multiplication.c
```

Now lets create an executable file for the driver using the library created as:

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -o driver driver.o libbasicmath.a
balajiganesh@helloworld:~/Desktop/C_programs$ ./driver
5 + 6 = 11
4 *  10 = 400
```

The above process can also be done as:

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -o driver -L . driver.o -
lbasicmath
balajiganesh@helloworld:~/Desktop/C_programs$ ./driver
5 + 6 = 11
4 *  10 = 400
```

The option `-L` tells the linker to search for the library files in the current directory (which specified as `.` after `-L` flag)

Creating Dynamic Library

- To create a Dynamic library, we need to provide our object files, the feature of Position Independent Code, because, at the time of compilation, just the library names are placed, only at the time of execution all the libraries get linked with the name where their respective names were placed and compiled.
- The compiler doesn't know how big or small the library would be, so when brought in memory the link to the library should get adjusted with the name where it was placed and compiled, that means it should have the feature of adjusting anywhere in the code. That's what the Position independent code means in layman terms.
- This feature can be applied to binary object files vi `-fPIC` or `-fpic` flag (stands for **P**osition **I**ndependent **C**ode)

  Compiling the addition.c and multiplication.c via `-fPIC` flag as

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -fPIC -c -Wall addition.c
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -fPIC -c -Wall
multiplication.c
```

- `-Wall` flag to display all the warning messages, if any gets generated or present.
- `-fPIC` flag to generate the binary object file as Position Independent Code.

These commands generate the new binary object files with the feature of PIC.

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -c driver.c
```

This command generates the driver's object file without the feature of PIC.
NOTE: Only the files that make a dynamic library, should be compiled with -`fPIC`, flag not the driver (or main) program.

```
balajiganesh@helloworld:~/Desktop/C_programs$ ls
addition.c  addition.p basicMath.h   driver    driver.o   multiplication.c
multiplication.o
```

Now lets create the dynamic library named libbasicmath.so, that should contain PIC codes as:

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -shared -o libbasicmath.so
addition.o multiplication.o
```

- Now we've created the dynamic library named libbasicmath.so, but in order to use it any time, we must install it, for that copy the generated dynamic library file to the `/usr/lib` directory.
- So that whenever a program is executed, all the libraries are copied into the memory only once, the program which has need with those libraries will make a link with the required.

Copying the libbasicmath.so to the `/usr/lib` (with root-user permission)

```
balajiganesh@helloworld:~/Desktop/C_programs$ sudo cp libbasicmath.so
/usr/lib
[sudo] password for balajiganesh:
```

Can be checked as

```
balajiganesh@helloworld:~/Desktop/C_programs$ ls /usr/lib libbasicmath.so
libbasicmath.so
```

Or via `ldd` command as

```
balajiganesh@helloworld:~/Desktop/C_programs$ ldd driver
	linux-vdso.so.1 (0x00007ffe979d2000)
	libbasicmath.so => /usr/lib/libbasicmath.so (0x00007fd3d308a000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd3d2c99000)
	/lib64/ld-linux-x86-64.so.2 (0x00007fd3d348e000)
```

Now we need to link this library to our driver file..

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -o driver driver.o
libbasicmath.so
```

or as

```
balajiganesh@helloworld:~/Desktop/C_programs$ gcc -o driver driver.o -lbasicmath
```

Running the driver program

```
balajiganesh@helloworld:~/Desktop/C_programs$ ./driver
5 + 6 = 11
4 *  10 = 400
```

References:

1. https://cs-fundamentals.com/c-programming/static-and-dynamic-linking-in-c#static-and-dynamic-linking
2. https://cs-fundamentals.com/tech-interview/c/difference-between-static-and-dynamic-linking
3. https://medium.com/swlh/linux-basics-static-libraries-vs-dynamic-libraries-a7bcf8157779

Dealing with processes in Linux

What is a Process?

A process can be defined as a program under execution or A program which got the CPU time.

Each and every process which is created will have a set of attributes, that helps in managing them easily, The most common attributes are PID (Process ID), Program instructions(data), Stack area, etc.,.

As Linux is a multi-processing OS, multiple processes can be run at a time, and each individual process runs in its own virtual address space and is not capable of interacting with another except through secure kernel and managed mechanisms.

A process in its lifetime, will be only in one instant at a time, they are: Ready, Running, Suspended/Waiting, killed/terminated. In linux processes can be viewed via `ps` command as:

```
balajiganesh@helloworld:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
balajig+    2690    2665  0 15:44 pts/0    00:00:00 -bash
balajig+    9686    2690  0 17:03 pts/0    00:00:00 ps -f
```

Where:

UID	**User ID (The user running it)**
PID	Process ID (The ID of the process)
PPID	Parent Process ID (The ID of the process that started this process)
C	CPU utilization of process
STIME	Start time
TTY	Terminal type associated with the process.
TIME	Amount of time held by the CPU
CMD	The command which launched the process

Types of Processes in Linux

Fundamentally classifying, there are two types of processes

1. **Foreground processes** (*Interactive Processes*)
   Initialized and controlled through terminal, and run the foreground. These processes are not launched automatically, rather launched by the user explicitly and those processes need user interaction in the foreground.
2. **Background processes** (*non-interactive* or *automatic processes*)
   These processes are launched automatically by some processes as system functions or services and they don't require any user interaction, thus called background processes.

Broadly classifying, there are 5 types of processes

1. Parent processes

These are the processes which create other processes during runtime, and this parent process is created by some other parent. Thus parent to all the processes at runtime is the **`init`** process (its `PID` is 1 and its `PPID` is `0`, that resembles it has no parent).

2. Child processes

These are the processes which are created by other processes during runtime. When a child is terminated, its parent gets a **`SIGCHLD`** signal from the `wait()` call, so that its parent can do some action when its child gets terminated, as well as the resources attached to the child process and the entry in the page table are also removed.

3. Orphan Processes

- It is the state of process, when its parent gets terminated before the child gets terminated. Then that process is adopted by the `init` process.
- Even though `init` process takes the parentship of the orphan process, it's still called the orphan process as its original parent process does not exist.

4. Daemon processes *(~ Services that run in background)*

- These are the system related processes and don't have any association with the terminal *(i.e., runs in background)* can only be created by the **root.**
- These processes are created as - A parent process is created via terminal, that inturn creates a new process, and the parent process gets terminated before its child termination, thus the child process becomes independent of the terminal and taken care by the `init` process, thus becomes a daemon process.

5. Zombie processes (or Defunct processes)

- It is the state of a process, where, even though a process gets terminated *(or completes its execution),* the entry in the page table is not removed until the parent fetches the state of its child process.
- As long as the parent process does not fetch the child state and remove its entry in page table, it's called zombie process because, even though it has dead (*i.e., terminated)*, the entry of the page table resembles that it's still alive (i.e., active) `--a` state like the zombies, so called Zombie processes.

Killing a process

A process can be killed using the kill command as

Syntax:

```
kill [options] <pid> [..] or simply ctrl^C
```

Ex.

```
kill -9 15266 or kill -s SIGKILL 15266
```

Kill command is used to issue a signal to the process specified, its default signal is TERM (for termination), and other signals can be specified as either a number or via a signal term (like `SIGKILL(9)`, `SIGCHLD(17)`, `SIGBUS(7)`..)

The id of the process can be found by the `pidof` command, as `pidof` returns the PID of chrome if it is running as a process currently.

System Calls for Creating and Handling processes

Creating processes and Differences between fork(), vfork(), exec()

fork()

A fork() system call is used to create a new duplicate process as like itself, and the execution in both the processes starts from the next instruction of the fork() call.

Declaration: `pid_t fork(void);`

- When the fork call is executed a new exact copy of the parent is created. But a fork system call creates a child that differs from its parent only in `PID` (Process ID) and in `PPID`.
- Both the parent and the child processes run simultaneously (In other words both processes are independent to each other) the same code.

- After the successful `fork()` call, the parent process gets the child PID as the return value from the fork() call, and the child process gets 0 as return value from the `fork()` call, in case of failure, the child process won't be created and the parent gets an error code (can know the error caused by passing it to the function `perror()` with declaration of `perror.h` and `errno.h`).

- Both Whenever the child completes its execution and gets terminated, a `SIGCHLD` signal is sent to its parent to release the resources allocated to it. Even the child process can explicitly use either `exit()` or `exit_()` system call to get terminated by its own, even in this way the parent gets the SIGCHLD signal.
- Even we can send the `SIGCHLD` signal to the parent process via kill command from the terminal.

vfork()

- A vfork() system call is also used to create the child process, Unlike the fork(), where parent won't wait for the child to exit, in `vfork()` call, the parent waits *(stays in suspend state for a short period of time till child gets terminated).*
- When the child process is created, it won't take separate memory to store the instructions, stack data..etc, rather it uses its parent's resources. Due to this `vfork()` is sometimes unpredictable when the child uses its parent's resources other than its own resources.

- It is **recommended** that **not to use** the `exit()` or `exit_()` system calls, if used then there is a chance of getting terminated, both the parent and child processes.
- A worth point to not is, same like the `fork()` call, child process gets its own unique `PID`, and for a successful `vfork()` call parent gets child PID and child gets 0 as the return value from the `vfork()` system call, in case of failure child is not created and parent gets an Error code.
  Declaration: `pid_t vfork(void);`

exec()

- A exec() system call is used to replace the current running process with a new program. It loads the new program into its current process space and runs it from the new program's entry point. exec() replaces the current process with the program pointed by the function.
- A worth point to note is - Control is never returned back to the original program unless exec() fails.
- exec is a family of system calls:

```
#include <unistd.h>

extern char **environ;

int execl(const char *pathname, const char *arg, ...
                                        /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
                                        /* (char  *) NULL */);
int execle(const char *pathname, const char *arg, ...
                  /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
                                            char *const envp[]);
```

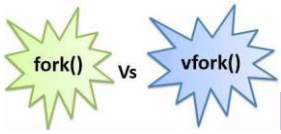
clone()

- This system call functions as like the fork() system call, the main difference between these two is just - the child process created via clone() call can share the process memory space, table of signal handlers, file descriptors with its parent and the child process created with the fork() system call can't.
- **This usage of this call is not recommended currently**.

wait()

wait() system call blocks the calling process until one of its child processes exits or sends a signal. It can exhibit either of the cases:

1. If at least one child is running while a wait() call is made, then the calling process is blocked until one of its child processes sends a signal or gets terminated, then after it gets resumed.
2. If there is no child process running while a wait() call is made, then the calling process has no effect on execution of wait.
   Declaration: `pid_t wait(int *status);`



Differences between `fork()` and `vfork()`

Basis for comparison	`fork()`	`vfork()`
Parent process suspension after creation of child process	No	Yes
Implicit synchronization b/w parent and child process	No	Yes (As parent automatically goes to suspension state)
Need of Explicit synchronization	Yes	No (Not required)
Resources sharing between Parent and child Process	No	Yes
Is performance* efficient?	No (As it copies the resources of parent, thus takes time)	Yes (Instead of a copy it shares its parents resources, this saves time)
Can Parent gets access to the variables used in child process *(even after child gets terminated)*	No	Yes

* But mostly performance is neglected ...(Sometimes).

Similarities between `fork()` and `vfork()`

- Both creates a new child process.
- Execution of child process starts from the immediate line after the `fork()` or `vfork()` (Success full call only..).
- Parent gets a SIGCHILD signal when the created child process exits (either implicitly – after successful completion of execution, explicitly – via **`_exit()`**, not recommended to use `exit()`)

Hands-On..

Creating and handling processes in Linux with system calls

- fork(), wait(), exit() and vfork()

Level-1: Normal `fork()` *ing..*

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

int main(int argc, char **argv)
{
      int count = 1;
      for(int i=0; i<3; i++)
                        fork();
      printf("Helloworld count = %d\n", count);
}
```

Output:

```
Helloworld count = 1
Helloworld count = 1
Helloworld count = 1
Helloworld count = 1
Helloworld count = 1
Helloworld count = 1
Helloworld count = 1
Helloworld count = 1
```

If '**n**' no. of fork calls in a program then no. of then no. of times, "`Hello world`" is executed is 2^3, and no. of child processes created are $(2^n) - 1$, as that 1 will be of parent.

Level-2: Handling Parent and child processes with `wait()`

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

int main(int argc, char ** argv)
{
      pid_t childPID;          // To hold the child's PID
      int childStatus;         // To hold the termination status of
                                  the child

      printf("Iam child of Parent PID %d(Bash Shell), my PID is
                  %d\n", getppid(), getpid());
      printf("Iam creating a new child process....\n");

      childPID = fork();       // Creating a child_process, and saving
                                  the fork() status inthe childPID

      if(childPID == -1)       // If forking fails..!!
      {
            printf("Couldn\'nt able to create the process..");
            return -1;
      }

      if(childPID == 0) // Will be true only in the child process
      {
            printf("Iam Child of Parent PID %d, My PID is
                        %d\n",getppid(), getpid());
            printf("Now I Am (%d) getting terminated very soon...\n",
                           getpid());
      }
      else // Will only be executed in the Parent process
      {
           wait(&childStatus);      //Wait till child gets terminated
                                       and store its return value in
                                         childStatus
           printf("Now my child has terminated whose PID is %d, with
                   return value %d\n", childPID, childStatus);
           printf("I\'ve came back...\n");
           printf("Even I am getting terminated very soon\n");
      }
}
```

Sample Output:

```
Iam child of Parent PID 10075(Bash Shell), my PID is 10111
Iam creating a new child process....
Iam Child of Parent PID 10111, My PID is 10112
Now I am(10112) getting terminated very soon...
Now my child has terminated whose PID is 10112, with return value 0
I've came back...
Even i am getting terminated very soon
```

Level-3: fork()ing with wait() and exit()

```
#include <sys/types.h>
#include <stdlib.h>
void job_reporting();
void job_reporting_and_exit();

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

	pid = fork();
	if(pid == 0)
		job_reporting_and_exit();
	else
		job_reporting();    // Cannot see exit of this..newly created
				    child process..
}

void job_reporting_and_exit()   // Without using exit()
{
	printf("Child created with PID: %d by PPID-%d\n", getpid(),
						getppid());
	printf("PID %d Reporting....\n", getpid());
	printf("EXiting PID-%d\n", getpid());

}

void job_reporting()   // With using exit()
{
	printf("Child created with PID: %d by PPID-%d\n", getpid(),
			getppid());
	printf("PID %d Reporting....\n", getpid());
	exit(EXIT_SUCCESS);
	printf("Exiting PID-%d\n", getpid());   // This won't get executed..
}
```

Sample Output:

```
Child created with PID: 12602 by PPID-12210
PID 12602 Reporting....
Child created with PID: 12603 by PPID-12602
PID 12603 Reporting....
EXiting PID-12603
```

Level-1: Normal `vfork()` *ing..*

basic_vfork.c

```
// Include required libraries...
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
      // variable to hold the return value from vfork()
      pid_t child_process;

      // create a new child with vfork()
      child_process = vfork();

      // A temporary variable to share b/w parent and child..
      int temp_var=0;

      // Evaluate action to be performed, based on obtained
                        return-value..
      switch(child_process)
      {
            case -1:
                  printf("Error in creating child
                              process....exiting...\n");
                  _exit(EXIT_FAILURE);  // use only _exit() not
                                                  exit()
                  break; /// think... not required a break;
                                          stmt..
            case 0: // Child process..
                  printf("Hello Iam child. My PID is %d and my
                          PPID is %d\n", getpid(), getppid());
                  printf("Started my work..\n");

                  for (; temp_var<10; temp_var=temp_var+2)
                        printf("%d, ", temp_var);
                  printf("Finished work...Bye from child\n\n");
                  break;

            default: // parent_process
                  printf("Iam in pause till my child finishies
execution... Now Iam active as its terminated now..\n\n");
                  printf("My work starts now..\n");
                  for (; temp_var<50; temp_var++)
                        printf("--%d--", temp_var);
                  printf("Bye from parent...\n");
      }
      return 0;
}
```

Compilation:

```
gcc basic_vfork.c -o basic_vfork
```

Execution:

```
./basic_vfork
```

Sample output:

```
Hello Iam child. My PID is 8894 and my PPID is 8893
Started my work..
0, 2, 4, 6, 8, Finished work...Bye from child

Iam in pause till my child finishies execution... Now Iam active as its terminated now..

My work starts now..
--0----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----32----33----34----35----36----37----38---
-39----40----41----42----43----44----45----46----47----48----49--Bye from parent...
```

NOTE: sometimes may get error as "`Segmentation fault (core dumped)`".. as vfork() implementation leads to some bugs..
Here in our system, when the vfork() is called, internally fork() is called, so memory space isn't shared, that's the reason -- parent's work didn't start from 11 (Where its child has left of..)

References:

1. All You Need To Know About Processes in Linux [Comprehensive Guide]
2. Linux Bootup process (6 stages)
3. https://www.geeksforgeeks.org/processes-in-linuxunix/
4. https://www.thegeekstuff.com/2012/02/unix-process-overview/
5. https://en.wikipedia.org/wiki/Zombie_process
6. https://www.coolcoder.in/2013/12/process-types-in-linux-unix-systems.html
7. https://www.geeksforgeeks.org/processes-in-linuxunix/
8. http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/wait.html
9. https://stackoverflow.com/questions/4856255/the-difference-between-fork-vfork-exec-and-clone
10. http://man7.org/linux/man-pages/man2/waitid.2.html
11. Man pages of kill

Exec() family of function and details of each...

12. https://man7.org/linux/man-pages/man3/exec.3.html
13. https://www.geeksforgeeks.org/exec-family-of-functions-in-c/

Differences between fork(), vfork(), exec(), clone()

14. Stack Overflow - Differences b/w fork(), vfork(), exec() and clone()
15. What's the difference between fork() and vfork()? (2 Solutions!!)

Behind the Internet Communication - Sockets

What is a socket communication?

Socket communication is a way of connecting two nodes or two processes in a network or within a system to communicate with each other.

Before Dwelling…let's get handy with Terminologies and concepts

`Hosts` -- runs the applications

Like browsers, applications which uses the internet.. etc.

`Routers` -- forward information

Forwarding the information from one place to the other place through a network.

`Packets` -- Sequence of bytes

Contains the control information. Ex. Source host, Destination host.

`Protocol` -- Like an agreement

What does a packet contain, what fields it stores..

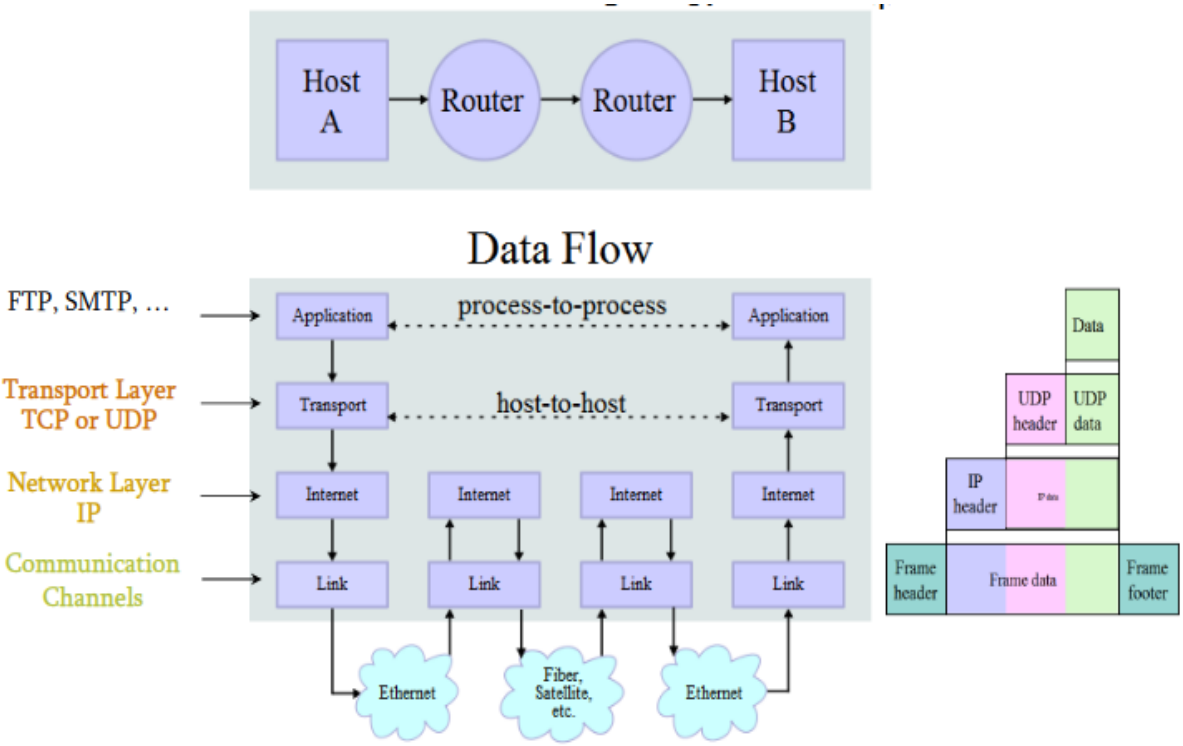
Size and structure of each packet

Ex: FTP, SNTP, HTTPs, SMTP..

Roadways of communication -- Protocol families/suites

- **A communication protocol is s a system of rules** that allows two or more entities of a communication system to transmit information via any kind of variation of a physical quantity.
- The protocol defines the rules, syntax, semantics and synchronization of communication and possible error recovery methods.
- There are several different protocols for different problems. For the communication between the two computers -- mostly TCP/IP protocol family is used because it provides
    - End-to-End connectivity
    - Specifies how data should be
        - Formatted
        - Addressed
        - Transmitted
        - Routed and received
    - Facility to use in the internet and in stand-alone private networks.

TCP/IP network topology:



Sockets types

When a socket is created, the program has to specify the address domain and the socket type. Two processes can communicate with each other only if their sockets are of the same type and in the same domain.

There are two widely used domains

1. Internet Domain
2. Unix Domain

- Among these two, Internet domain is the most widely used domain. Using Internet domain two processes running on any two hosts on the internet can communicate.
- The UNIX domain sockets are just used to communicate in the same host. Because of this this domain is also called as **IPC (Inter Process Communication) socket**.

The address of a socket in the Unix Domain is a character string which is basically an entry in the file system.

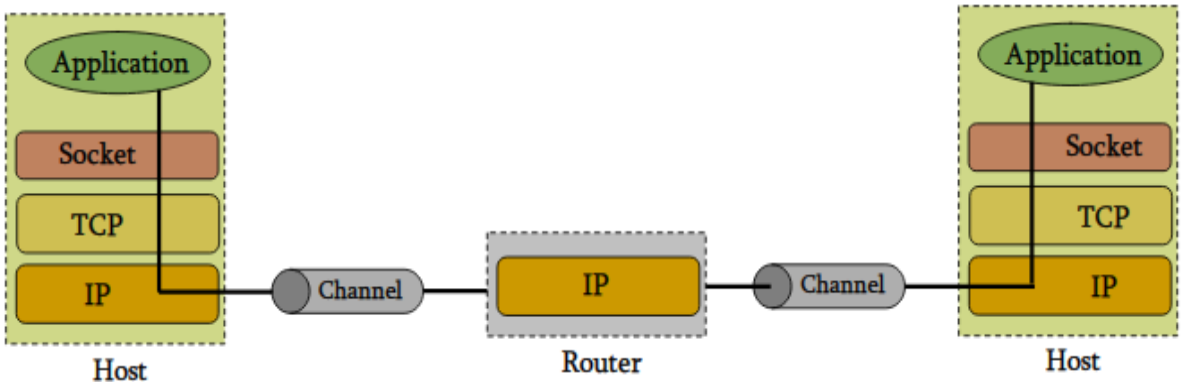
The address of a socket of the internet domain consists of the Internet address of the host machine (IP Address) along with a unique port to communicate.

Sockets (Berkeley Sockets) and Workers those help in our work

Berkeley Sockets is an API for Internet sockets and Unix domain sockets. It is commonly implemented as a library of linkable modules. It originated with the **4.2-BSD Unix OS**, which is **released in 1983**.

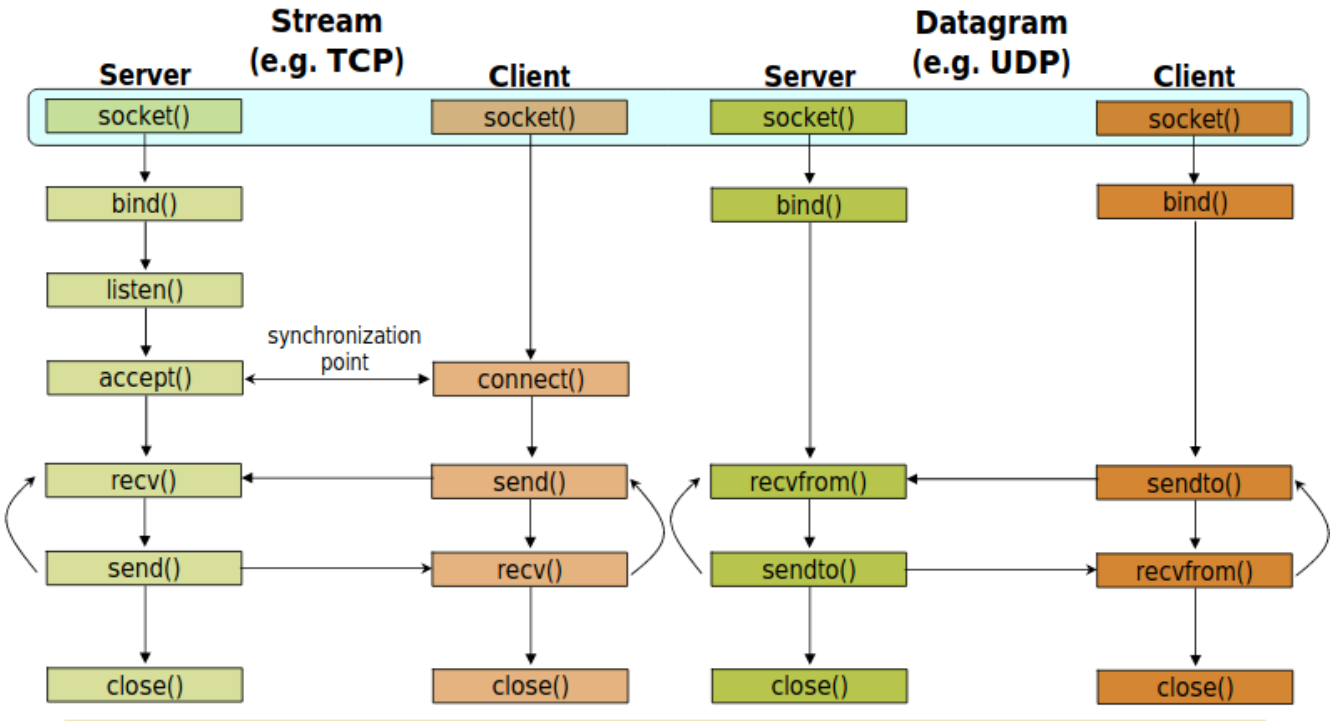
Pipeline

High level view (Abstract view)



Low level view (Actual view)

Socket API functions -- Workers those help in our work of communication are:



Socket Creation – `socket()`:

- It is used to create a new socket of a certain type, identified by an integer number, and allocates system resources to it.
- It creates an endpoint for communication and returns a `file descriptor(fd)` for the socket. It takes 3 arguments as :

```
int sockid = socket(domain, type, protocol);
```

NOTE: All the passing parameters are of type integers defined as the Macros.

- *domain* - It specifies the protocol family of the create socket. Most commonly used domains are:
  - PF_**INET** - for the network protocol of type IPv4 addresses only.
  - **PF_UNIX** - for a local socket(i.e., for a socket within a system) via a file
  - **PF_INET6** - for the network protocol of type IPv6.
- *type* - type of communication mode, can be used either of the one
  - **SOCK_STREAM** -- for stream-oriented services (used if domain is AF_INETX), and its is most reliable mode to use. So, called Stream Sockets.
  - **SOCK_DGRAM** -- for datagram service (used if domain is AF_UNIX), and it is not so reliable. So, called Datagram sockets.

    (Rarely used ones are: SOCK_SEQPACKET and SOCK_RAW)
- *protocol* - To specify the actual transport protocol to use. The most commonly used one are:
  - **IPPROTO_TCP** (Transmission Control Protocol)
  - **IPPROTO_SCTP** (Stream Control Transmission Protocol)
  - **IPPROTO_UDP** (User Datagram Protocol)
  - **IPPROTO_DCCP** (Datagram Congestion Control Protocol)

    --- all are defined in `netinet\in.h`

  [0 can be passed to select a default protocol from the selected domain and type.]

- It returns -1 if any error occurs in creating a socket, else an integer that represents newly assigned file descriptor(fd).

NOTE: **`socket()`** just creates a socket interface, with just this interface communication can't be done. For that, this interface should be associated with the IP address of the system and the port number where it should listen for the incoming request of a client or send a request to a server.

Assigning address to socket - `bind()` :

- It is typically used on the server side, and associates a socket with a socket address structure (i.e., IP Address and a port number).
  - ~ It associates a socket with an address.
- As when a socket is created, just the family is given to a socket, by this **`bind()`**, a socket is associated with the address and the port number.
- It takes 3 arguments as:

```
int status = bind(sockfd, &addrport, size);
```

  - *sockfd*- A descriptor representing the descriptor of the socket created.
  - *addrport* - A pointer to a sockaddr structure, that represents the address to bind to.

    Place the code line that performs this action..
  - *addrlen* -- A field of type socklen_t that specifies the size of the sockaddr structure.
- It returns 0 on success, and -1 if failure.

Ex:

```
int sockfd, status;
struct sockaddr_in addrport;
sockfd = socket(PF_INET, SOCK_STREAM, 0);

addrport.sin_family = AF_INET;
addrport.sin_port = htons(5100);        // Taking port as 5100
addrport.sin_addr.s_addr = htonl(INADDR_ANY);
status = bind(sockfd, (struct sockaddr * )&addrport, sizeof(addrport);
```

NOTE: Sometimes **`bind()`** can be skipped.

- For datagram sockets:
  - If only sending no need of **`bind()`**. The OS finds a port each time the socket sends a packet.
  - If receiving, need to **`bind()`**, as the OS cannot know particularly, from which port to receive the packets if multiple applications transfer the packets.
- For Stream sockets:
  - --some clarification is needed--

Listening the client requests - `listen()`: ~ *only for the stream oriented connections, a **Non-Blocking call***

- It is used on the server side, and causes a bound TCP socket to ener listening state., ..i.e., for SOCK_STREAM and SOCK_SEQPACKET socket types
- It takes two arguments as:
  Int status = listen(sockfd, backlog); // sometimes backlog is also referred as queuelimit
  - *sockfd* - A valid socket file descriptor.
  - *backlog* - An integer that represents the no. of pending connections that can be queued at any one time. For at least one connection request, this call is a blocking call for the process that calls this function. For every accepted connection by the server, it is dequeued from the listening queue.
- It returns 0 on success and -1 on failure.

Taking a Connection request – `accept()`: ~ Only for the Stream sockets, a **Blocking Call**

- It is used on the server side, to accept an incoming request to create a new TCP connection from the remote client,and creates a new socket associated with the socket address pair of the connection.
- It creates a new socket each time it dequeues the connection request from the listening queue.
- It takes the following arguments:
  - *sockfd* - The descriptor of the listening socket that has the connection queued.
  - *clientaddr* - A pointer to a sockaddr structure to receive the client's address information(client's address and port).
  - *addrlen* - A pointer to a socklen_t location that specifies the size of the client address structure passed to accept().

- It returns the new socket descriptor for the accepted connection, or -1 in case of failure. Further communication between the client and the server happens through this socket only.

NOTE: Datagram sockets don't need the `accept()` call as the receiver immediately responds to the listening socket.

Establishing Connection - `connect()` ~ A blocking call

- It is used on the **client side**, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection , if the UDP socket creates a new file descriptor.
- It takes 3 arguments as:
  - *sockid* - Socket ID, that is returned by the socket() call.
  - *foreign_addr* - Address of the server that client would like to connect of type struct sockaddr.
  - *addrlen* - size of
- It returns 0 if success and -1 in case of failure.

Exchanging data with Stream sockets - send(), recv()

`send()` ~ Blocking Call till the data(msg) is sent

- It is used to send the data to the connected socket.
- It takes 4 arguments as:

```
int count = send(sockfd, msg, msgLen, flags);
```

  - *sockfd* - Socket file descriptor of the destination(type--integer).
  - *msg* - message to be transmitted to the host(type -- const void[])
  - *msgLen* - Length of the message(in bytes) to transmit (type -- integer)
  - *flags* - Special options, to modify the sending message (type -- integer *(as Macros)*)
- It returns -1 if error in sending data, and no. of bytes transmitted if success.

`recv()` ~ Blocking Call till the entire message is received

- It is used at the receiver side to receive the content from the sender.
- It takes 4 arguments as:

```
int count = recv (sockfd, recvBuff, buffLen, flags);
```

  - *sockfd* - Socket ID of the sender (type integer).
  - *recvBuff* - To store received bytes (type -- void[])
  - *buffLen* - Length of bytes to be received (type -- integer).
- It returns total no. of bytes received and -1 if error

Exchanging data with datagram sockets - sendto(), recvfrom()

`sendto()` ~ Blocking Call till the data(msg) is sent

```
int count = sendto(sockid, msg, msgLen, flags, &foreignAddr, addrlen);
```

- *msg*, *msgLen*, *flags*, *count* `---- same as send()`
- *foreignAddr*: `struct sockaddr, address of the destination`
- *addrLen*: `sizeof(foreignAddr)`

`recvfrom()` ~ Blocking Call till the data (msg) is received

```
int count = recvfrom(sockid, recvBuffer, bufferLen, flags,
&clientAddr, addrLen)
```

- *recvBuffer*, *bufferLen*, *flags*, *count* --- same as recv()
- *clientAddr*: struct sockaddr, address of the client
- *addrLen*: sizeof(clientAddr)

Closing a Socket - `close()`

- Used to release the resources allocated to a socket in a system.
- It takes the **`sockid`** of the socket to be closed as:
- **`int status = close(sockfd);`**
- It returns **0** if successful, and **`-1`** if error.

gethostbyname, gethostbyaddr()

- These functions are used to resolve the host names and addresses in the domain name system or the local host's other resolver mechanisms.
- They return a pointer to an object of type **`struct hostent`**, which describes an internet protocol host.
- It takes the following arguments:
  - *name* - specifies the DNS name of the host.
  - *addr* specifies a pointer to a **`struct in_addr`** containing the address of the host.
  - *len* specifies the length in bytes, of **`addr`**.
  - *type* - specifies the address family type **(Ex. `AF_UNIX`, `AF_INET`..)** of the host address.
- These functions return the **`NULL`** pointer in case of any error, in which the external integer **`h_errno`** maybe checked to see whether the occured error is a temporary or an invalid host, else a valid **`struct hostent*`** is returned.

select()

This is used to suspend, waiting for one or more of a provided lust of sockets to e ready to read,, ready or write, or that have errors.----not much clear..

getsockopt()

This is used to retrieve the current value of a particular socket option for the specified socket.

setsockopt()

This is used to set a particular socket option fro the specified socket.

Hands-On..

Establishing Client-Server communication -- behind the Scenes (Working Theory)

- Setting up **a simple TCP server** involves the following steps:
  1. Creating a TCP socket, with a call to **`socket`**().
  2. Binding the socket to the listen port, with a call to **`bind`**(). Before calling **`bind`**(), declare a **`sockaddr_in`** structure, fields. Converting a *short int* to network byte order can be done by calling the function **`htons`**() (host to network short).
  3. Preparing the socket to listen for connections (making it a listening socket), with a call to **`listen`**().
  4. Accepting incoming connections, via a call to **`accept`**(). This blocks until an incoming connection is received, and then returns a socket descriptor for the accepted connection. The initial descriptor remains a listening descriptor, and **`accept`**() can be called again at any time with this socket, until it is closed.
  5. Communicating with the remote host, which can be done through **`send`**() and **`recv`**() or **`write`**() and **`read`**().
  6. Eventually closing each socket that was opened, once it is no longer needed, using **`close`**()

- Programming **a simple TCP client** application involves the following steps:
  7. Creating a TCP socket, with a call to **`socket`**().
  8. Connecting to the server with the use of connect(), passing a **`sockaddr_in`** structure with the **`sin_family`** set to **`AF_INET`**, **`sin_port`** set to the port the endpoint is listening (in network byte order), and **`sin_addr`** set to the IP address of the listening server (also in network byte order.)
  9. Communicating with the server by using **`send()`** and **`recv()`** or **`write()`** and **`read`**().
  10. Terminating the connection and cleaning up with a call to **`close`**().

Establishing Client-Server communication --in the scene (Practical approach)

Server.c

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

int main(void)
{
     struct sockaddr_in sa;
     int SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
     if (SocketFD == -1)
     {
          perror("cannot create socket");
          exit(EXIT_FAILURE);
     }

     memset(&sa, 0, sizeof sa);

     sa.sin_family = AF_INET;
     sa.sin_port = htons(1100);
     sa.sin_addr.s_addr = htonl(INADDR_ANY);

     if (bind(SocketFD,(struct sockaddr *)&sa, sizeof sa) == -1)
     {
          perror("bind failed");
          close(SocketFD);
          exit(EXIT_FAILURE);
     }

     if (listen(SocketFD, 10) == -1)
     {
          perror("listen failed");
          close(SocketFD);
          exit(EXIT_FAILURE);
     }


     for (;;)
     {
            int ConnectFD = accept(SocketFD, NULL, NULL);

            if (0 > ConnectFD) {
            perror("accept failed");
            close(SocketFD);
            exit(EXIT_FAILURE);
     }
```

```
      /* perform read write operations ... */
      char buff[30];
      read(ConnectFD, buff, sizeof(buff));
      printf("Client sent: %s\n", buff);
      write(ConnectFD, "Hii client", sizeof("Hii client"));
      printf("Replied to the client..\n");


      if (shutdown(ConnectFD, SHUT_RDWR) == -1)
     {
          perror("shutdown failed");
          close(ConnectFD);
          close(SocketFD);
         exit(EXIT_FAILURE);
      }
      //close(ConnectFD);
      //}

      close(SocketFD);
      return EXIT_SUCCESS;
}
```

Compiling: `gcc server.c -o server`

client.c

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

  int main(void)
  {
       struct sockaddr_in serveraddr;
       int res;
       int SocketFD;

       SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
       if (SocketFD == -1)
      {
          perror("cannot create socket");
          exit(EXIT_FAILURE);
       }

       memset(&serveraddr, 0, sizeof serveraddr);
```

```
        serveraddr.sin_family = AF_INET;
        serveraddr.sin_port = htons(1100);
        serveraddr.sin_addr.s_addr = inet_addr("127.0.0.1");

        if(connect(SocketFD,(struct sockaddr *)&serveraddr,sizeof serveraddr)==-1)
       {
           perror("connect failed");
           close(SocketFD);
           exit(EXIT_FAILURE);
        }

        /* perform read write operations ... */
        write(SocketFD, "Hii Server..!!", sizeof("Hii Server..!!"));
        printf("Message sent to the server..\nNow waiting for the server\'s
                    message\n");
        char buff[30];
        read(SocketFD, buff, sizeof(buff));
        printf("Server sent: %s\n", buff);

        shutdown(SocketFD, SHUT_RDWR);

        close(SocketFD);
        return EXIT_SUCCESS;
    }
```

(In terminal-2..)
Compiling: `gcc client.c -o client`

(In terminal-1..)
Running the server: `./server`

(In the terminal-2)
Running: `./client`

```
Message sent to the server..
Now waiting for the server's message
Server sent: Hii client
```

(In the terminal-1)

```
Message sent to the server..
Now waiting for the server's message
Server sent: Hii client
```

Advanced-version implementations..

Chatting between the server and client
https://github.com/Balaji-Ganesh/IndustrialTrainingWorks/tree/master/sockets/2ndJan-2020/themeChatting
Communication between multiple computers serving different tasks each computer as a server and a computer as client
https://github.com/Balaji-Ganesh/IndustrialTrainingWorks/tree/master/sockets/4thJan-2020

A Simple Messaging system

chatServer.c

```
/*
*  This is the chat server..
     This server listens the chat's from various clients(currently only 1,
upgradation many[limited])
*/

// Necessary library includes..
#include<stdio.h>
#include<netinet/in.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netdb.h>
#include<sys/types.h>
#include<string.h>
#include<arpa/inet.h>
#include<unistd.h>

#define PORT 9999
#define BUFFER_LEN 100
#define BACKLOG_REQUESTS 5

void endChatSession(int client_socket)
{
    if(write(client_socket, "end-chat", 8) == -1)
    {
       perror("Error sending end-chat to the connected
client..\nTerminating abruptly\n");
       exit(1);
    }
    close(client_socket);
}


// This function wil chat with the connected client..
int chatWithClient( int client_socket)
{  /*
       * @param: client_socket: Socket file descriptor of the accepted
client request.
         return: 0 -- To terminate connection with the connected client and
again wait to accept the connections from the client.
                 1 -- To terminate connection with the connected client and
shutdown the server.
   */

   //Buffer variable..
   char buffer[BUFFER_LEN];
```

```
   int length;


   printf("\t\t\t----Welcome to the interactive chat Session between Server
and Client\n");
   printf("You can type help-me to show the list of commands that can
help..\n");


   for(;;) // Keeps on chatting with the client, till termination by the
client.
   {
       //Clear the buffer... as it might produce ambigious results while
working..
       bzero(buffer, BUFFER_LEN);

       // Reading the data from the client..
       if(read(client_socket, buffer, sizeof(buffer)) == -1)
       {
            perror("Error reading message from the client..\nTerminating
Abruptly\n");
            exit(1);
       }

       if(strncmp("end-chat", buffer, 8) == 0)  // If end-chat signal from
client..
        {
          close(client_socket);
          printf("Terminating chatting with the client...end-chat signal
from client\n");
          return 0;
        }

       // Displaying the client's data..
       printf("Client: %s\n", buffer);

       // Taking the input from the server-side..
       takeInput:
       // Sending the reply to the client..
       bzero(buffer, BUFFER_LEN);
       printf("Server: ");
       scanf("%[^\n]%*c", buffer);

       // Validating the server-side entered text..
       if(strncmp("end-chat", buffer, 8) ==0)
       {
           endChatSession(client_socket);
           return 0;
       }
       if (strncmp("quit-server", buffer, 11) == 0)
       {
          endChatSession(client_socket);
          printf("Server: Terminating connection with the client.\nQuitting
Server...\n");
          return 1;
       }
       if (strncmp("help-me", buffer, 7) == 0)
       {
           printf("\n----HELP----\n");
           printf("\"end-chat\": To terminate chat with the connected
client and wait for new client requests.\n");
           printf("\"quit-server: To terminate chat with the connected
client and terminate the server\"\n");
```

```
            printf("\"help-me\": To show list of pre-fixed commands\n");
            goto takeInput;
        }

        // Sending the reply to the client..
        if(write(client_socket, buffer, sizeof(buffer)) == -1)
        {
           perror("Error sending message to the client..\nTerminating
abruptly..\n");
           exit(1);
        }
    }
}


int main()
{
    // Socket descriptors..
    int server_socket, client_socket;
    // Socket structeres-length holders..
    int server_length, client_length;

    // Creating socket addresses variables..
    struct sockaddr_in server, client;

    // Removing old sockets if present..
    unlink("server-client-chat");
    // Creating socket interface..
    if((server_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
       perror("Socket creation failed");
       exit(1);
    }
    printf("Socket Creation done...\n");

    #if 0
    int yes = 1;
    if((setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)))==-1)
    {
       perror("setsockopt failed");
      exit(1);
    }
    #endif


    // Naming the socket..
    server.sin_family = AF_INET;
//    server.sin_addr.s_addr = inet_addr("192.168.29.62");
    server.sin_addr.s_addr = INADDR_ANY;
    //server.sin_addr.s_addr = htonl(INADDR_ANY);
    //server.sin_port = PORT;
    server.sin_port = htons(PORT);    // In client it used htonl then here
too use same..if htons then htons..

    server_length = sizeof(server);

    // Binding..
    if(bind(server_socket, (struct sockaddr *)&server, server_length) < 0)
    {
       perror("Binding Failed");
       exit(1);
    }
```

```
    //Listening for connections from the clients.. and placing them in the
Queue
    if(listen(server_socket, BACKLOG_REQUESTS) == -1)
    {
        perror("Listening failed");
        exit(1);
    }
    client_length = sizeof(client);

    // Enqueueing the connection requests and serving them...
    for(;;)  // infinite loop
    {
       printf("Waiting for the connection requests from the clients...\n");
       if((client_socket = accept(server_socket, (struct sockaddr
*)&client, &client_length)) == -1)
       {
           perror("Accepting connections failed");
           exit(1);
       }
       printf("New Connection Accepted: \n client_socket(fd): %d\nIP
address: %s\nPort: %d\n",
                              client_socket, inet_ntoa(client.sin_addr),
ntohs(client.sin_port));
       if(chatWithClient(client_socket))
       {
          printf("\nServer shutting down..\nServer down..\n");
          close(server_socket);
          break;  // To break this infinite loop...
       }
    }
}
```

(In system-2..)
Compiling: `gcc chatServer.c -o chatServer`

(In system-2..)
Running the server: `./chatClient`

chatClient.c

```
/*
*  This is the client chat program, it invokes the connection with the
server,if server has accepted then it chats with the server...
*
*/

// Necessary includes...
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<arpa/inet.h>
#include<netinet/in.h>

#define PORT 9999
#define BUFFER_LEN 100
```

```
void chatWithServer(int client_socket)
{
    printf("\t-----Welcome to the Interactive Chat Session between Server
and Client-----\n");
    printf("You are the client, type your messsage to send to the
server..\n");
    printf("You can type help-me to show the list of commands.\n\n");

    // Buffer to hold the data from the server...
    char buffer[BUFFER_LEN];

    for(;;)// infinite loop
    {
       takeInput:
       bzero(buffer, BUFFER_LEN);
       printf("Client: ");
       scanf("%[^\n]%*c", buffer);

       // If some pre-defined commands..
       if(strncmp("end-chat", buffer, 8) == 0)
       {
          printf("Terminating the chat with the server..\n");
          if(write(client_socket, "end-chat", 8) == -1)
          {
              perror("end-chat signal to the server failed\nTerminating
Abruptly..\n");
              exit(1);
          }
          break;
       }
       if(strncmp("help-me", buffer, 7) == 0)
       {
           printf("\n----HELP----\n");
           printf("\"end-chat\": To terminate the chat with the server and
have a safe quit.\n");
           printf("\"help-me\" : To show the help.\n");
           goto takeInput;
       }

       // Sending data to the server..
       if(write(client_socket, buffer, BUFFER_LEN) == -1)
       {
          perror("Sending message to failed");
          exit(1);
       }
       // Getting data from the server..
       if(read(client_socket, buffer, BUFFER_LEN) == -1)
       {
           perror("Reading mesage from server failed");
           exit(1);
       }
       // If the server sent "end-chat" then quit the chat..
       if(strncmp("end-chat", buffer, 8) == 0)
       {
          printf("Chat ended..!! Signal from Server\n");
          break;
       }
       printf("Server: %s\n", buffer);
    }
    close(client_socket);
}
```

```
int main()
{
    // Socket descriptors..
    int client_socket;

    struct sockaddr_in client;

    // Creating the socket..
    if( (client_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1 )
    {
        perror("Socket Creation failed");
        exit(1);
    }

    // Naming the socket...
    client.sin family = AF INET;
    client.sin_addr.s_addr = inet_addr("127.0.0.1");
    client.sin_port =htons(PORT);

    // connect to the server..
    if(connect(client_socket, (struct sockaddr *)&client, sizeof(client))
== -1)
    {
       perror("Connection Failed");
       exit(1);
    }
    printf("Connected to server...\nStarting chat..\n");


    chatWithServer(client_socket);

    return 0;
}
```

(In system1..)
Compiling: `gcc chatClient.c -o chatClient`

(In system-1..)
Running the server: `./chatClient`

Sample output:

It will be more interesting than to run by own rather than viewing sample output.....:)..

Sorry to say this way, but no linux machine available right now..

Server-client Model..

threadedChatServer.c

!! Don't forget to set the proper IP addresses..

```
/*
*  This is the multi-threaded chat server..
     This server listens the chat's from various clients and serves them..

   This program can achieve:
        Receiving each of the client machine, and printing those messages on
the server's console..
             (i.e., based on how they got connected to the server:
Connection Order)
              ---->> By the function named ""receiveClientsMessages()""
        Sending all of the server's messages to all of the connected
clients one by one based on how the got conected.
           (i.e., Connection Order)
              ---->> By the function named ""sendServerMessage()""

        Server handles all those connections
              (means prints all the clients messages and takes the server
input and sends them to all clients.)
              ---->> By the function named ""interactWithClients()""

   *  Upgradation + Handling Exceptions:
        If in between any client has exited, we should not send messages to
that particular client and inform the server that,
            that particular client has exited....

*/

// Necessary library includes..
#include<stdio.h>
#include<netinet/in.h>
#include<stdlib.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<netdb.h>
#include<sys/types.h>
#include<string.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<pthread.h>

#define PORT 9734
#define BUFFER_LEN 100
#define BACKLOG_REQUESTS 5   // Make sure that this and below one should be
same(i.e., SERVE_CLIENTS)
#define SERVE_CLIENTS 5      // Make sure that this and the above one
should be same (i.e., BACKLOG_REQUESTS)

/***  Global Function declarations..***/
//  Actual-Chat functions..
int chatWithClient(int client_socket);
void endChatSession(int client_socket);
//  Chat-Helping Functions..
void receiveClientsMessages();
```

```
void sendServerMessage();
void interactWithClients();
// Bug-resolving FUnctions
void setUpForChatSession(int *, struct sockaddr_in * );


/*** Global variable declarations.. ***/
// Threading variables..
pthread_t threads[SERVE_CLIENTS];
int totalThreadsServed = -1;           // Default value before serving the
clients..
char client_details[SERVE_CLIENTS][30];// To store the connected client's
IPaddress and their respective port numbers.
int savedDetails_count = -1;           // Default value, Count To store
their details of no. of clients which are being connected
// Chat-Session variables..
int connClients count = -1;           // Default value, To Store the count
of which clients got connected.

// Main code and execution starts from here..
int main()
{
    // Socket descriptors..
    int server_socket, clients_fd[SERVE_CLIENTS];
    // Socket structeres-length holders..
    int server_length, client_length;

    // Creating socket addresses variables..
    struct sockaddr_in master_server, clients_addr[SERVE_CLIENTS];

    //   Before creating socket, setting up the things..
    setUpForChatSession(clients_fd, clients_addr);

    // Creating socket interface..
    if((server_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
       perror("Socket creation failed");
       exit(1);
    }
    printf("Socket Creation done...\n");

    #if 0
    int yes = 1;
    if((setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)))==-1)
    {
       perror("setsockopt failed");
      exit(1);
    }
    #endif

    //clearing sockaddr_in object..
    memset(&server, '\0', sizeof(master server));

    // Naming the socket..
    master_server.sin_family = AF_INET;
    master_server.sin_port = htons(PORT);
    master_server.sin_addr.s_addr = inet_addr("192.168.29.62");

    server_length = sizeof(server);

    // Binding..
    if(bind(server_socket, (struct sockaddr *)&server, server_length) < 0)
```

```
    {
       perror("Binding Failed");
       exit(1);
    }

    //Listening for connections from the clients.. and placing them in the
Queue
    if(listen(server_socket, BACKLOG_REQUESTS) == -1)
    {
        perror("Listening failed");
        exit(1);
    }

    client_length = sizeof(struct sockaddr_in);

    // Enqueueing the connection requests and serving them...
    for(;;)  // infinite loop
    {
       printf("Waiting for the connection requests from the clients...\n");
       if((clients_fd[++connClients_count]=accept(master_socket, (struct
sockaddr *)&clients_addr[connClients_count] &client_length)) < 0)
       {
           char tempErr_msg;
           snprintf(tempErr_msg, "Accepting connections failed when
connClients_count is at: %d, due to ", connClients_count);
           perror( tempErr_msg );
           exit(1);
       }
       printf("Client-%d. New Connection Accepted: \n client_socket(fd):
%d\nIP address: %s:%d\n", connClients_count ,
          clients_fd[connClients_count],
inet_ntoa(clients_addr[connClientsCount].sin_addr),
ntohs(clients_fd[connClients_count].sin_port));

       //  Here we are creating the thread.. for each client that is
accepted.
       // Storing the connected client's IPaddresses and their respective
portnumbers..
       snprintf(client_details[++savedDetails_count],"%s:%s ",
inet_ntoa(clients_fd[connClients_count].sin_addr,

ntohs[connClient_count].sin_port) );
       // thread-Creation...
       pthread_create(&serveThreads[++totalThreadsServe], NULL,
chatWithClient, NULL);
       if(chatWithClient(client_socket))
       {
          printf("\nServer shutting down..\nServer down..\n");
          close(server_socket);
          break;  // To break this infinite loop...
       }
    }
}


void endChatSession(int client_socket)
{
    if(write(client_socket, "end-chat", 8) == -1)
    {
       perror("Error sending end-chat to the connected
client..\nTerminating abruptly\n");
       exit(1);
    }
```

```
    close(client_socket);
}


// This function wil chat with the connected client..
int chatWithClient( int client_socket)
{  /*
       * @param: client_socket: Socket file descriptor of the accepted
client request.
         return: 0 -- To terminate connection with the connected client and
again wait to accept the connections from the client.
                 1 -- To terminate connection with the connected client and
shutdown the server.
   */

   //Buffer variable..
   char buffer[BUFFER LEN];
   int length;


   printf("\t\t\t----Welcome to the interactive chat Session between Server
and Client\n");
   printf("You can type help-me to show the list of commands that can
help..\n");


   for(;;) // Keeps on chatting with the client, till termination by the
client.
   {
       //Clear the buffer... as it might produce ambigious results while
working..
       bzero(buffer, BUFFER_LEN);

       // Reading the data from the client..
       if(read(client_socket, buffer, sizeof(buffer)) == -1)
       {
            perror("Error reading message from the client..\nTerminating
Abruptly\n");
            exit(1);
       }

       if(strncmp("end-chat", buffer, 8) == 0)  // If end-chat signal from
client..
        {
          close(client_socket);
          printf("Terminating chatting with the client...end-chat signal
from client\n");
          return 0;
        }

       // Displaying the client's data..
       printf("Client: %s\n", buffer);

       // Taking the input from the server-side..
       takeInput:
       // Sending the reply to the client..
       bzero(buffer, BUFFER_LEN);
       printf("Server: ");
       scanf("%[^\n]%*c", buffer);

       // Validating the server-side entered text..
       if(strncmp("end-chat", buffer, 8) ==0)
       {
```

```
            endChatSession(client_socket);
            return 0;
        }
        if (strncmp("quit-server", buffer, 11) == 0)
        {
           endChatSession(client_socket);
           printf("Server: Terminating connection with the client.\nQuitting
Server...\n");
           return 1;
        }
        if (strncmp("help-me", buffer, 7) == 0)
        {
            printf("\n----HELP----\n");
            printf("\"end-chat\": To terminate chat with the connected
client and wait for new client requests.\n");
            printf("\"quit-server: To terminate chat with the connected
client and terminate the server\"\n");
            printf("\"help-me\": To show list of pre-fixed commands\n");
            goto takeInput;
        }

        // Sending the reply to the client..
        if(write(client_socket, buffer, sizeof(buffer)) == -1)
        {
           perror("Error sending message to the client..\nTerminating
abruptly..\n");
           exit(1);
        }
    }
}

void setupForChatSession(int *client_fd, struct sockaddr_in *
connectedClients)
{
    /*
        * This function will prepare for the chat session by clearing all
the memory values with nulll to avoid bugs..
        * @param: client_fd        - An array of client descriptors.
                  connectedClients - An array of the client socket
adddresses.
          return: nothing(void).
    */
    // First making the descriptors to be "-1"
    for(int i = 0; i<SERVE_CLIENTS; i++)
       client_fd[i] = -1;

    // Next filling with NULL('\0') character..
    for(int i = 0; i<SERVE_CLIENTS; i++)
      memset(&connectedClients[i], '\0', sizeof(struct sockaddr_in));
}
```

(In system1..)

Compiling: `gcc threadedChatServer.c -o threadedChatServer` (Actually for proper compilation need even more files, please go to the link given above)

(In system-1..)

Running the server: `./threadedChatServer`

Client_updated.c

```
/*
 * Server-1:  (Arthematic Server)
 *      hostname : snti4
 *      IPaddress: 192.168.29.185
 * Server-2: (Sorting Server)
 *      hostname : cit
 *      IPaddress: 192.168.29.43
 * Server-3: (Searching Server)
 *      hostname : cit
 *      IPaddress: 192.168.29.128
 *
*/



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

#define PORT 9734


/** Global function Declarations  ***/
void arthematicOperations(int sockfd, struct sockaddr_in address, int
developer);
void sortingOperations(int sockf, struct sockaddr_in address, int
developer);
void searchingOperations(int sockfd, struct sockaddr in address, int
developer);

/** Server-IP addresses  **/  // Please change these before
executing...these are dynamic IPaddress
char arthematicServerIP[] = "192.168.29.185";
char sortingServerIP[]    = "192.168.29.185";
char searchingServerIP[]  = "192.168.29.128";

int main(int argc, char *argv[])
{
        // local variables
        int developer = 0;
        // Verifying the mode of execution..
        if((strcmp(argv[1], "-developer") == 0) || strcmp(argv[1], "-d") ==
0)
        {
                printf("  ** Running in Developer-mode..\n");
                developer = 1;
        }
        else
                printf("  ** Running in User-mode..\n");


        // Socket-Descrptors..
        int sockfd;
        // Socket addresses
```

```
        struct sockaddr_in address;
        // Length holders..
        int sockLength;
        // Input holders..
        int num1, num2, choice, result;
        char operation;     // Will  be used in updation...

        // Creation of socket interface..
        if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
                perror("error creating socket");
                exit(EXIT_FAILURE);
        }

        // Clearing the socket
        memset(&address, '\0', sizeof(address));
        // Naming the socket..
        address.sin_family = AF_INET;
        // IP address assigning is done as per the respective operation
selected..
        address.sin_port = htons(PORT);

        printf("1. Arthematic Operations \n2. Sorting \n3. Searching
\nSelect your choice: ");
        scanf("%d", &choice);
           if(developer) printf("Received %d as the operation type\n",
choice);
        switch(choice)
        {
                case 1: arthematicOperations(sockfd, address, developer);
                        break;
                case 2: sortingOperations(sockfd, address, developer);
                        break;
                case 3: searchingOperations(sockfd, address, developer);
                        break;
                default: printf("Received invalid Choice\n");
        }
        // closing the socket..
        close(sockfd);
           if(developer) printf("\n\n**** Socket communication successfully
ended...\n\n");
}



void arthematicOperations( int sockfd, struct sockaddr_in address, int
developer)
{
        if(developer) printf("Going to perfrom arthematic operation..\n");

        // Local variables..
        int num1 = 0, num2 = 0, result = 0, choice = 0;

        // Assigning the Arthematic Server's IP address
        address.sin_addr.s_addr = inet_addr(arthematicServerIP);

        // connecting to the server....
        if(connect(sockfd, (struct sockaddr *)&address, sizeof(address)) ==
-1)
        {
                perror("An Unexpected error occured while connecting with
the server");
                exit(EXIT_FAILURE);
```

```
        }
          // If successfully connected to the server..
          if(developer) printf("Connected to the Arthematic server(%s)
Successfully..\n", arthematicServerIP);

        // Getting the operation
        printf("1. Addition\n2. Subtraction\n3. Multiplication\n4.
Division\nEnter your choice: ");
        scanf("%d", &choice);

        // Take the operands from the user..
        printf("Enter value-1: ");
        scanf("%d", &num1);
        printf("Enter value-2: ");
        scanf("%d", &num2);

        // Sending the data to the server which is connected..
        // first sending the operands as decided..
        write(sockfd, &num1, sizeof(int));   // Sending the operand-1
           if(developer)printf("Sent %d to the server\n", num1);

        write(sockfd, &num2, sizeof(int));   // Sending the operand-2
           if(developer) printf("Sent %d to the server\n", num2);

        write(sockfd, &choice, sizeof(int)); // Sending the operator-value
           if(developer) printf("Received %d from the server\n", choice);

        // Getting the output from the server
        read(sockfd, &result, sizeof(int));
        printf("Result: %d\n", result);

        if(developer) printf("Work successfully completed with the
ArthematicServer(%s)", arthematicServerIP);

}

void sortingOperations(int sockfd, struct sockaddr_in address, int
developer)
{
        if(developer) printf("Going to perform Sorting operations..\n");

        // Local variables and their default values..
        int length = 0, *array = NULL, temp = 0, confirmation = -1, choice =
-1;
        char *confirmBuffer;

        // Assigning the Sorting Server's IP address
        address.sin_addr.s_addr = inet_addr(sortingServerIP);
        printf("hiiiiiiiiiii\n");
        // connecting to the server....
        if(connect(sockfd, (struct sockaddr *)&address, sizeof(address)) ==
-1)
        {
                perror("An Unexpected error occured while connecting with
the server");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Connected to the Sorting Server(%s)
successfully\n", sortingServerIP);

        /**  Taking the list of values from the user.. **/
        // Taking the array length
        printf("Enter the no. of values to be sorted: ");
```

```
        scanf("%d", &length);
        // Creating the array dynamically based on the user-required-length
        array = (int *) malloc(length * sizeof(int));
        if(array == NULL)
        {
                printf("Sorry user..!! Your compter doesn't have enough size
to store the values in buffer\nTerminating..\n");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Memory of length %d is successfully
allocated in the system\n", length);

        // Getting the list of values from the user..
        for(int i=0; i<length; i++)
        {
                printf("Element-%d: ", i+1);
                scanf("%d",&temp);
                    if(developer) printf("Received %d from the console-
input.\n", temp);

                array[i] = temp;
                   if("Placed %d in the array at index-%d.\n", array[i], i);
        }

        /** Sending these list of values to the sorting server **/
        // Passing the length of the array..
        if( write(sockfd, &length, sizeof(int)) == -1)
        {
                perror("Error sending the length");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Sent the length(%d) of array to the server
successfully\n", length);
        // Getting the acknowledgement from the server whether the length of
desired array is created or not..
        if( read(sockfd, &confirmation, sizeof(int)) == -1  )
        {
                perror("Error receiving the acknowledgement of the length");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Received %d as signal from the server\n",
confirmation);

        if(confirmation == 1)
        {
                if (developer)
                        printf("Received PROCEED signal from the server\n");
        }
        else
        {
                if(developer)
                        printf("Sorry server doesn't have enough
memory..\n");
                exit(EXIT_FAILURE);
        }

        // Now send the list of values to the server..
        for(int i=0; i<length; i++)
        {
                if(write(sockfd, &array[i], sizeof(int)) == -1)
                {
                        perror("Error sending value ");
```

```
                        if (developer) printf("Error sending %d at index
%d\n", array[i], i );
                        exit(EXIT_FAILURE);
                }
                   if(developer) printf("Successfully sent %d to the
server\n", array[i]);
        }

        // Getting the acknowledgment of data-sent to the
server.................later in updation re-sending the data if any
error...
        // so for that sending "data-sent" signal..
        strcpy(confirmBuffer, "data-sent");
        if(write(sockfd, &confirmBuffer, sizeof(confirmBuffer)) == -1)
        {
                perror("Error sending the \"data-sent\" to server");
                exit(EXIT FAILURE);
        }
           if(developer)printf("Successfully sent \"data-sent\" signal to
the server..\n");

        // Getting back the "data-recv" signal..
        if(read(sockfd, &confirmBuffer, sizeof(confirmBuffer)) == -1)
        {
                perror("Error in receiving acknowledgement signal from
server");
                exit(EXIT_FAILURE);
        }
        // Comparing the received signal...
        if(strcmp(confirmBuffer, "data-recv") == 0)
        {
                if(developer)
                        printf("Successfully received \"data-recv\" from
server\n");

        }
           if(developer) printf("\nData and length of data is successfuly
sent\nNow proceeding with the sorting operations...\n");
        // Getting the desired type of sorting operation from the user...
        printf("\n\n1. Bubble Sort\n2. Selection Sort\n3. Insertion srt\n4.
Merge Sort\n5. Quick Sort\nEnter your choice: ");
        scanf("%d", &choice);

        // Sending the desired-sorting operation to the server...
        if(write(sockfd, &choice, sizeof(int)) == -1)
        {
                perror("Error sending sorting operation to the server");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Sent sorting operation(%d) successfully to
the server\n", choice);

        // Now getting the sorted-values fromt the server...
        for(int i=0; i<length; i++)
        {
                if(read(sockfd, &array[i], sizeof(int)) == -1)
                {
                        perror("Error receiving value at index \n ");
                          if(developer) printf("Error receiving at index
%d\n", i);
                        exit(EXIT_FAILURE);
                }
```

```
                    if(developer) printf("Received %d from the server...\n",
array[i]);
        }

        // Displaying the output to the user...
        printf("Sorted Values are: ");
        for(int i=0; i < length; i++)
                printf("%d\t", array[i]);

          if(developer) printf("Sorting operation successfully
completed....\n");

}

void searchingOperations(int sockfd, struct sockaddr_in address, int
developer)
{

        if(developer) printf("Going to perform Searching operations\n");

        // Local variables and their default values..
        int length = 0, *array = NULL, temp = 0, confirmation = -1, choice =
-1, searchResult = -1, keyElement = 0;
        char *confirmBuffer;

        // Assigning the Sorting Server's IP address
        address.sin_addr.s_addr = inet_addr(searchingServerIP);

        // connecting to the server....
        if(connect(sockfd, (struct sockaddr *)&address, sizeof(address)) ==
-1)
        {
                perror("An Unexpected error occured while connecting with
the server");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Connected to the Searching server(%s)
successfully\n", sortingServerIP);

        /**  Taking the list of values from the user.. **/
        // Taking the array length
        printf("Enter the no. of values to be sorted: ");
        scanf("%d", &length);
        // Creating the array dynamically based on the user-required-length
        array = (int *) malloc(length * sizeof(int));
        if(array == NULL)
        {
                printf("Sorry user..!! Your compter doesn't have enough size
to store the values in buffer\nTerminating..\n");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Memory of length %d is successfully
allocated in the system\n", length);

        // Getting the list of values from the user..
        for(int i=0; i<length; i++)
        {
                printf("Element-%d: ", i+1);
                scanf("%d", &temp);
                    if(developer) printf("Received %d from the console-
input.\n", temp);

                array[i] = temp;
```

```
                    if("Placed %d in the array at index-%d.\n", array[i], i);
        }

        /** Sending these list of values to the sorting server **/
        // Passing the length of the array..
        if( write(sockfd, &length, sizeof(int)) == -1)
        {
                perror("Error sending the length");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Sent the length(%d) of array to the server
successfully\n", length);
        // Getting the acknowledgement from the server whether the length of
desired array is created or not..
        if( read(sockfd, &confirmation, sizeof(int)) == -1  )
        {
                perror("Error receiving the acknowledgement of the length");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Received %d as signal from the server\n",
confirmation);

        if(confirmation == 1)
        {
                if (developer)
                        printf("Received PROCEED signal from the server\n");
        }
        else
        {
                if(developer)
                        printf("Sorry server doesn't have enough
                                        memory..\n");
                exit(EXIT_FAILURE);
        }

        // Now send the list of values to the server..
        for(int i=0; i<length; i++)
        {
                if(write(sockfd, &array[i], sizeof(int)) == -1)
                {
                        perror("Error sending value");
                        printf("Error in sending %d at index %d\n", array[i],
i);
                        exit(EXIT_FAILURE);
                }
                   if(developer) printf("Successfully sent %d to the
server\n", array[i]);
        }

        // Getting the acknowledgment of data-sent to the
server.................later in updation re-sending the data if any
error...
        // so for that sending "data-sent" signal..
        strcpy(confirmBuffer, "data-sent");
        if(write(sockfd, &confirmBuffer, sizeof(confirmBuffer)) == -1)
        {
                perror("Error sending the \"data-sent\" to server");
                exit(EXIT_FAILURE);
        }
           if(developer)printf("Successfully sent \"data-sent\" signal to
the server..\n");

        // Getting back the "data-recv" signal..
```

```
        if(read(sockfd, &confirmBuffer, sizeof(confirmBuffer)) == -1)
        {
                perror("Error in receiving acknowledgement signal from
server");
                exit(EXIT_FAILURE);
        }
        // Comparing the received signal...
        if(strcmp(confirmBuffer, "data-recv") == 0)
        {
                if(developer)
                        printf("Successfully received \"data-recv\" from
server\n");

        }
           if(developer) printf("\nData and length of data is successfuly
sent\nNow proceeding with the sorting operations...\n");
        // Getting the desired type of sorting operation from the user...
        printf("\n\n1. Linear Search\n2. Binary Search\n3. Ternary
Search\nEnter your choice: ");
        scanf("%d", &choice);

        // Sending the desired-sorting operation to the server...
        if(write(sockfd, &choice, sizeof(int)) == -1)
        {
                perror("Error sending sorting operation to the server");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Sent sorting operation(%d) successfully to
the server\n", choice);

        printf("Enter the element to be searched: ");
        scanf("%d", &keyElement);

        if(write(sockfd, &keyElement, sizeof(int)) == -1)
        {
                perror("Error sending key-element to the server..\n");
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Sent the key-element(%d) to the server..",
keyElement);

        // Now getting the search-result from the server...
        if(read(sockfd, &searchResult, sizeof(int)) == -1)
        {
                perror("Error receiving Search status.\n ");
                if (developer) printf("Received %d as search result\n",
searchResult);
                exit(EXIT_FAILURE);
        }
           if(developer) printf("Received %d as search status from the
server...\n", searchResult);

        // Displaying the output to the user...
        if(searchResult == -1)
                printf("Sorry element(%d) is not found in the list of values
which you've entered..\n", keyElement);
        else
                printf("YaY..!! Your element-%d is found at index %d\n",
keyElement, searchResult);

          if(developer) printf("Searching Operation successfully
completed....\n");
```

```
}
```

(In system2..)
Compiling: `gcc client_updated.c -o client_updated`

(In system-1..)
Running the client: `./client_updated`

....Run the clients in multiple machines ... and test... Cannot show the sample output, as now enough linux machines to get the output... but it works fine TESTED...!! (excluding minor bugs)

Credits goes to....

Helping, testing, ideas, guidance..

- N. Madhu Sir
- K. Swarnakanth Sir

Coding Team:

- C. Srikanth
- T. Prashanth
- B. Harsha Vamsi
- B. Varshitha
- K. Vaishali
- B. Radhika
- B. Balaji Ganesh

References:

https://www.geeksforgeeks.org/socket-programming-cc/
https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/
https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf
https://users.cs.duke.edu/~chase/cps196/slides/sockets.pdf
http://home.iitk.ac.in/~chebrolu/ee673-f06/sockets.pdf
https://en.wikipedia.org/wiki/Berkeley_sockets
https://en.wikipedia.org/wiki/Communication_protocol
https://en.wikipedia.org/wiki/Internet_protocol_suite
https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
https://medium.com/swlh/understanding-socket-connections-in-computer-networking-bac304812b5c

Multiple tasks at a time - Multi Threading (in C)

What is Multitasking and Multithreading?

Multitasking is the process of doing multiple tasks at a time. But practically speaking in a uniprocessor system, it varies as: The processor switches between each task in a short period of time (micro or nano seconds) that gives us an **illusion of happening multiple tasks at a time**. Whereas in a multiprocessor system, the OS distributes the multiple tasks to multiple processors.

Multitasking is of two types:

- Process-based (called Multiprocessing)
- Thread-based (called Multithreading)

Process-based multitasking handles the concurrent execution of programs (i.e., Processes), whereas thread-based multitasking refers to the concurrent execution of pieces of the same program.

Thread, and thread properties?

A thread can be simply defined as the single sequence stream within a process. Alternatively, it can also be defined as - thread is a lightweight process (because it has less properties compared to the process which has many properties).

Only the process created the threads, maintain a thread table to manage them, but no thread knows that there are some more threads along with it.

Each thread has unique properties, they are:

- Thread ID (TID)
- Set of registers and stack pointer
- Stack for local variables and return addresses
- Signal mask
- Priority
- Return value: `errno`

Threads in the same process can share:

- The same address space
- Process instructions
- Open file descriptors (`fd's`)
- Signals and signal handlers
- Current working directory
- User and group ID
- Resources that process have permission to use

Threads v/s Processes

Similarities

- At any one instance the crucial resource of the system, CPU is only shared by an active one (i.e., an active thread or an active process).
- Instructions are executed sequentially
- Can create new children (i.e., Child processes and child threads)
- If any of the process/thread blocks, other process/thread has no effect (Assuming independence between them).

Differences

- Threads are not independent as processes, because threads in a process strive for a common task of completing the execution of a single process, whereas processes are independent as each process is created for a different reason.
- All threads can access entire resources of the process, but all processes cannot access all the resources of the system.
- Threads are designed especially to assist one another, so that execution of a process completes faster. But processes might or might not assist one another, it depends on the user who creates them.

Types of threads

User-level threads

- User-level threads are implemented in user-level libraries, rather than via system calls. So thread switching does not need to call OS to cause the interrupt to the kernel. In fact, the kernel knows nothing about the user-level threads and manages as if they were a single-threaded process.
- **Pros:**
  - These threads don't need the modification to OS [*as **..find a reason]***
  - Simple representation -- Each thread is represented simply by a PC, registers, stack and small control block, all these are stored in the user process address space.
  - Fast and efficient -- as thread switching is not much expensive as procedure call(here expensive means -- in terms of time consumption).
- **Cons:**
  - Lack of coordination between threads and OS kernel (as OS kernel doesn't know how many threads are running in a process). Therefore, a process gets one time slice irrespective of no. of threads in a process whether a process has 100 threads or a process with a single thread.
  - These threads require non-blocking system calls (i.e., multithreaded kernel). Otherwise, the entire process will be blocked in a kernel, even if there are multiple runnable threads left in the process because of a one thread block.
  - **Ex**. a thread causes a I/O request or a page fault, the entire process is blocked.

- ## Kernel-level threads
  - The existence of these threads in a process is known to the kernel, thus the kernel manages those threads.
  - No runtime system is needed in this case, instead of a thread table in each process, the kernel will have a thread table that keeps track of all threads in the system.
  - In addition, the kernel also maintains the traditional process table to keep track of process. OS kernel provides system call to create and manage threads.
  - **Pros:**
    - As Kernel has full knowledge of all the threads, scheduling will happen in a way that the process with more no. of threads is given more time compared to the process with the less no. of threads.
    - These threads are especially good for those applications, which frequently block as kernel will schedule its time it will soon get its turn to execute, whereas in user-level threads, the kernel doesn't know and the scheduling takes more time than expected.
  - **Cons:**
    - The kernel-level threads are slow and inefficient. For instance, threads operations are hundred times slower than that of user-level threads.
    - As the kernel has the responsibility of managing and scheduling threads as well as the processes, it requires a full TCB (Thread Control Block) for each process to maintain info about threads. Thus, there is a significant overhead and increased in kernel capacity that leads in a longer latency time,

Creating a simple thread using POSIX standard

- C does not contain any built-in support for multithreaded applications. It relies on a POSIX (Portable OS Interface) standard for threads.
- **Basic Thread operations** includes
  - Creation,
  - Termination,
  - Synchronization (joining, blocking),
  - scheduling,
  - data management and
  - process interaction.

1. Thread creation

- **Syntax:**

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*),
    void *arg);
```

- **Description:** pthread_create() function starts a new thread in the calling process. The execution of the created new thread starts by invoking the start_routine() with the arguments to that function as arg.
- **Parameters:**
  - **thread:** pointer to unsigned integer value, created new thread's TID is stored in

this variable..

- **attr:** Pointer to a structure that is used to define the thread attributes like detached state, scheduling policy, stack address etc,. Can be set to NULL to use default attributes.
- **start_routine:** Pointer to a subroutine that should be executed by the thread. The return_type and parameter_type of the subroutine must be of type void *. The function has a single attribute to pass a single value, but if needed to pass multiple values, then structure can be used.
- **arg:** Pointer to void * that contains the arguments to the function passed in the previous argument.

- **Returns:** If successful, it returns 0, else an error-number is returned to indicate the error.

2. Thread termination

- **Syntax:**

```
#include <pthread.h>
int pthread_exit(void *retval);
```

- **Description:**

It terminates the calling thread and returns a value via retval. If the thread is joinable (i.e, another thread is waiting for the termination of this thread), It uses return value for joining.

- **Parameters:**
  - **retval:** A pointer to an integer that stores the return status of the thread terminated.
- **Returns:** Does not return to the caller and this function always succeeds.

Program to explain the thread creation and thread and termination.

basic_thread_create_terminate.c

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

void *print_message_function( void *ptr );

int main()
{
       pthread_t thread1, thread2;
       char *message1 = "Thread 1";
       char *message2 = "Thread 2";
       int  iret1, iret2;

       /* Create independent threads each of which will execute function */

       iret1 = pthread_create( &thread1, NULL, print_message_function, (void*)
                                                              message1);
       iret2 = pthread_create( &thread2, NULL, print_message_function, (void*)
                                                               message2);
```

```
        /* Wait till threads are complete before main continues. Unless we  */
        /* wait we run the risk of executing an exit which will terminate   */
        /* the process and all threads before the threads have completed.   */

        pthread_join( thread1, NULL);   // The main_thread waits till the thread-1
                                                terminates
        pthread_join( thread2, NULL);   // The main_thread waits till the thread-2
                                                terminates...

        printf("Thread 1 returns: %d\n",iret1);
        printf("Thread 2 returns: %d\n",iret2);
        exit(0);
}

void * print_message_function( void *ptr )
{
        char *message;
        message = (char *) ptr;    // conversion of void type to the char pointer
                                            type..
        printf("%s \n", message);  // At last print the message
}
```

Compilation:

```
    # gcc basic_thread_create_terminate.c -o basic_thread_create_terminate
     -lpthread
```

Execution:

```
    ./basic_thread_create_terminate
```

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Creating a multiple threads using POSIX standard with Thread Synchronization Methods

Some basic thread synchronization mechanisms:

1. **Mutexes (Mutual Exclusion Lock):** Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
2. **Joins -** Make a thread wait till other threads complete (terminate).
3. **Condition variables**: Allows us to stop and resume the thread based on certain conditions.
4. **Semaphores:** A process synchronization tool. It manages the resources by a simple integer variable called "**S**" which is normally initialized to no. of resources present in the system.

1. Mutexes

- Mutexes are used to prevent data inconsistencies due to race conditions. A race condition occurs when two or more threads need to perform operations on the same memory area, but the results of computations depend on the order in which these operations are performed.
- Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than on thread the resource should have a Mutex associated with it. We can apply a mutex to protect a segment of memory (usually called as "Critical Region") from other threads.

NOTE: Mutexes can only be applied to threads in a single process, it doesn't work between processes as do Semaphores.

Ex: To apply a mutex to a global variable, a file resource which is as below:

```
FILE file1 = foopen("dummy_file.txt", "w");
void write_file()
{
       fprintf(file1, "%s", "Hello World.." );
}
```

When this function is given to handle by multiple threads, then...it leads to race conditions, which we don't require. This case can be handled via Mutexes as:

```
pthread_mutex_t mutexFile = PTHREAD_MUTEX_INITIALIZER;
FILE file1 = foopen("dummy_file.txt", "w");
void write_file()
{
    pthread_mutex_lock(&mutexFile);  // First lock here..
    fprintf(file1, "%s", "Hello World.." );   // Crucial resource handling...
    pthread_mutex_unlock(*mutexFile); // Then unlock here..
}
```

NOTE: Scope of variable ("Which is considered as critical section") and the mutex variable should be the same..!!
A program to demonstrate the mutexes

thread_mutex.c

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

void *testFunction();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

int main()
{
```

```
      int thread1_status, thread2_status;
      pthread_t thread1, thread2;


/* Create independent threads each of which will execute testFunction */

      if( (thread1_status = pthread_create(&thread1, NULL, &testFunction, NULL)))
      {
         printf("Thread creation failed: %d\n", thread1_status);
      }

      if((thread2_status = pthread_create( &thread2, NULL, &testFunction, NULL)))
      {
          printf("Thread creation failed: %d\n", thread2_status);
      }

      // Join the main_thread till both the threads get terminated..
      pthread_join( thread1, NULL);
      pthread_join( thread2, NULL);

      exit(0);
   }

   void *testFunction()
   {
       pthread_mutex_lock( &mutex1 );
       counter++;
       printf("Counter value: %d\n",counter);
       pthread_mutex_unlock( &mutex1 );
   }
```

Compilation:

```
   gcc thread_mutex.c -o thread_mutex -lpthread
```

Execution:

```
   ./thread_mutex
```

```
   Counter value: 1
   Counter value: 2
```

Now output will be same even how many time we run the program

But, if the lines "`pthread_mutex_lock( &mutex1 );`" and `pthread_mutex_unlock( &mutex1 );` are commented and compiled, then each time it may print as:

```
Counter value: 1
Counter value: 1
-----
Counter value: 2
```

```
Counter value: 1
-----
Counter value: 1
Counter value: 2
```

The output is unpredictable... this is the advantage of using mutex, by using mutex we can synchronize the situation and make scene predictable.

2.Joins

- A join is performed when a thread needs to run or resume its operation when one process finishes its execution, then join can be used to achieve this motive.

Example program to demonstrate the usage of join

thread_join.c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define NTHREADS 10
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

int main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   // Create 10 threads..
   for(i=0; i < NTHREADS; i++)
   {
       pthread_create( &thread_id[i], NULL, thread_function, NULL );
   }

   // Wait til all the 10 threads get terminated....
   for(j=0; j < NTHREADS; j++)
   {
       pthread_join( thread_id[j], NULL);
   }

   // Now print the final value...
   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );    // Locking the resource
   counter++; // Performing the operation on the data(Critical Section)..
   pthread_mutex_unlock( &mutex1 );  // Finally unlocking the resource..
}
```

Compilation:

```
gcc thread_join.c -o thread_join -lpthread
```

Execution:

```
./thread_join
```

```
Thread number 140603424020224
Thread number 140603440805632
Thread number 140603432412928
Thread number 140603337733888
Thread number 140603273017088
Thread number 140603415627520
Thread number 140603329341184
Thread number 140603312555776
Thread number 140603304163072
Thread number 140603320948480
Final counter value: 10
```

3. Condition Variables

- A condition variable is a variable of type **`pthread_cond_t ans`** is used with the appropriate functions for waiting and later, continuation of the process.
- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.
- A condition variable must always be associated with a mutex to avoid race condition by one thread preparing to wait, and another thread which may signal the condition before the first thread actually waits on it resulting in deadlock.
- The thread will be perpetually waiting for a signal that is never sent (deadlock case..).

Functions used in conjunction with the condition variable

- Creation and Destroying
  - `pthread_con_init`
  - `pthread_cond_t`
  - `Pthread_cond_destroy`
- Waiting on condition:
  - `pthread_cond_wait`
  - `pthread_cond_timedwait`
- Waiting a thread based on a condition
  - `pthread_cond_signal`
  - `pthread_cond_broadcast`

4. Semaphores

- A synchronization tool that uses "**`S`**" and integer variable to achieve synchronization.
- Normally, the value of the "**`S`**" will be the no. of resources present in the system, and the value of this is only allowed to change by only two functions – **`wait()`** and **`signal()`**

How it achieves the synchronization?

- **Technical Understanding *(for Technical definition look Types of Semaphores)…***
  When a process is modifying the value of the semaphore, no other process can simultaneously modify the value of semaphore.
- **Intuitive Understanding…**
  It's like a single entry and exit room (i.e., resource) with a door of size – only one process can go and come out. When a process is in need with the room, it enters, until the entered (i.e., capturing of resource) process finishes its work with the room (i.e., resource) and comes out (i.e., releasing of resource), no other process cannot enter the room (i.e., capture the resource).

Types of Semaphores:

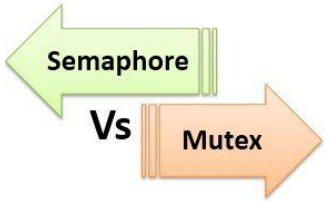
- **Counting Semaphores:** The semaphore variable S, is initialized to no. of resources present in the system.
  - Whenever a process wants to access the resource, it performs wait() operation and decrements the value of S by one.
  - When it releases the resource, it performs signal() operation, which performs the increment of S by one.

- **Binary Semaphores:** Here the value of semaphore variable (**`S`**), ranges between **`0`** and **`1`** (i.e., binary values – **`0`** and **`1`**).
  - Its value is decremented by **`wait()`** when any process wants to use the resource, and incremented by signal(), when the process releases the resource.
  - This is used extensively in the scenarios where, it s necessary that, at any instance only one process should access the resource, no other process is allowed to access the resource.
  - **Ex:** When a process is saving (i.e.,. writing) the data of a file in the hard disk in some sectors, then no other process is allowed to write to the same sectors of hard disk. (In some cases, reading may be allowed but modification or writing is not allowed in any case, as it leads to data inconsistency),

Caution: Don't get confused with mutex and Binary Semaphores. Mutex is locking mechanism, whereas [Binary] Semaphore is a signaling mechanism.

NOTE: When the value of **`S`** becomes **`0 (zero)`**, it means that all the resources of the system are in access by some processes. Then **`wait()`** becomes blocked call as no resource available yet, until some process releases the resource (then **`S`** becomes **`>0`**, so can be decremented by **`wait()`** call) and those processes will stand in a queue.

Differences between Semaphores and Mutex



Basis of Difference	Semaphores	Mutex
Mechanism	Signaling mechanism	Locking mechanism
Means of synchronization ***(i.e., through what, synchronization is achieved?)***	Via Integer variable "**S**"	Via mutex object.
Synchronization achievement (i.e., How synchronization is achieved?)	**Decrement**ing of **S** by one, when a process needs a resource, **increment**ing of S by one when it releases	**Lock**ing of the mutex object, when one process wants to access the resource, **unlock**s it after its work gets done.
When to use?	When multiple processes wants the access of a resource (but there are **finite instances** of that resource).	When multiple processes wants to access a resource (**Only single instance**).
Is resource usage exclusive access or shared?	**Exclusive access**, as there are finite quantity of resources.	**Sharing** of resource as there is only one instance of resource (Only one process is allowed to as at any instance)
Can resource blocking controlled by other process? ***Assume a blocking condition where all the instance(s) were in use, by some or the other process,***	**Yes**, any process (which is accessing resource) can control the blocking. **Explanation:** As there are multiple instances of resources if any process which release the resource, then value of **S** be **>0**, so the process which is in queue to blocking can access the released instance of resource.	**No**, blocking cannot be controlled by other process. **Explanation:** As there is only resource instance, only that process which locked the resource can unlocked, then another process gets chance.

An example program to demonstrate the concept or semaphores:

basic_semaphore.c

```
/*Import required libraries..*/
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>   // for semaphore related datatypes and functions
#include <unistd.h>

// Declare a semaphore variable..
sem_t semaphore;

void print_numbers()
{
  for (int i=0; i<=20; i++)
  {
    printf("%d, ", i);  // print the value..
    sleep(0.5);         // Wait for half second
  }
}

void *thread_worker(void *arg)
{
  printf("\n\nHello, Iam thread %s\n..",  (char *)arg);
  // perform wait() (ie., decrement 'S' value by 1)
  sem_wait(&semaphore);
  printf("Decremented the S....\nStarted Accessing the resource in critical
  section...\n");

  // critical section...
  print_numbers();

  // perform signal() (i.e., Increment the 'S' by 1)
  sem_post(&semaphore);
  printf("Incremented the S..\nExiting from critical section as my work is
              finished....:)..");
}


// Main code starts from here...
int main()
{
  // Initialize the semaphores..
  sem_init(&semaphore, 0, 1);
  // Creating threads, to show synchronization with semaphores in concurrent
  execution
  pthread_t thread1, thread2;
  // Launch thread1 by creating a new thread
  pthread_create(&thread1, NULL, thread_worker, (void *)"1");
  // wait for a while..
  sleep(1);
  // Create and launch another thread..
  pthread_create(&thread2, NULL, thread_worker, (void *)"2");;
```

```
    // Pause execution of main-thread till completion of both the threads..
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Finally destroy the semaphore created...
    sem_destroy(&semaphore);

    return 0;
}
```

Compilation:

```
# gcc basic_semaphore.c -o basic_semaphore -pthread -lrt
```

Execution:

```
./basic_semaphore
```

Sample Output:

```
Hello, Iam thread 1
..Decremented the S....
Started Accessing the resource in critical section...
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
  Incremented the S..
Exiting from critical section as my work is finished....:)..

Hello, Iam thread 2
..Decremented the S....
Started Accessing the resource in critical section...
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
  Incremented the S..
Exiting from critical section as my work is finished....:)..
```

- We can observe that semaphore made a synchronization between thread execution --- that's the reason, the output came as expected it to be, else leads to race conditions...
- Another **main usage of Semaphore** is in scenarios where it leads to the **race conditions.**

Resources:

1. http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm
2. https://www.geeksforgeeks.org/multithreading-c-2/
3. https://en.wikipedia.org/wiki/POSIX_Threads
4. https://www.geeksforgeeks.org/thread-functions-in-c-c/

Differences between Semaphores and Mutex:

5. Simple and Easy explanation - Tech differences

Further Reading:

https://randu.org/tutorials/threads/
https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp

https://computing.llnl.gov/tutorials/pthreads/

Communication between Processes - IPC Mechanisms (in C)

How can processes communicate between each other?

There are some couple of ways, through which processes can communicate between each other. The most commonly used methods are

1. Pipes
2. FiFOs
3. Shared memory
4. Semaphores

1. Pipes (unnamed pipes)

- Pipe is a byte stream, i.e., no concept of messages or message boundaries when using a pipe. The process reading from a pipe can read blocks of data of any size, regardless of the size of blocks written by the writing process.
- In pipe, the data retrieval (reading) and data filling (writing) follows the property of queue i.e.,. The is written first is read first by the reading process and data which is written last is read last by the reading process - FIFO property.
- Pipes are unidirectional i.e., data can travel only in one direction through a pipe, if one end is used for writing then another end is used for reading, similar to the physical pipe where one end is used to pump the water in and the other end is used for flushing out.
- Pipes have limited capacity i.e., pipe is simply a buffer in a kernel memory. This buffer has a maximum capacity, once the pipe is full further writes to the pipe are blocked until the reader removes some data from the pipe.

NOTE: Pipes can be created only between those processes where there exists some relation between the two processes. Like parent-child relationship between processes.

Creating and using Pipes

Syntax:

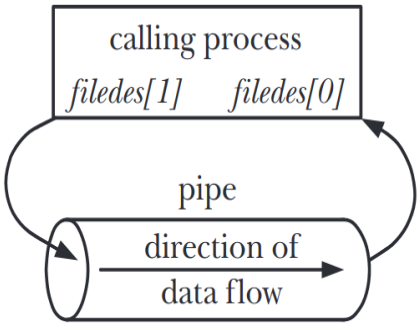
```
#include <unistd.h>
int pipe(int filedesc[2]);
```

Parameters:

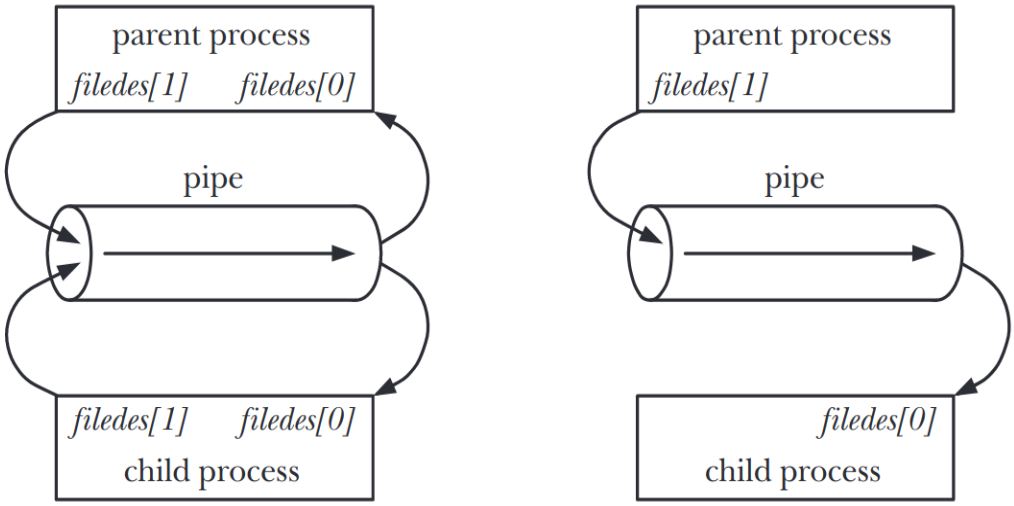
- **filedesc:** An array of length 2 of type integer, to store the two file descriptors where each descriptor for each end of the pipe. Normally filedesc[0] is for read end and filedesc[1] for write-end.

Returns: 0 on Success and -1 on error.

A Situation of the calling process after **`pipe()`** is:



- Each end of the pipe is linked to the called process.
- To establish a communication with a pipe between two processes, then we need to use `fork()` to create a new process after creating a pipe. Then the situation would look like..(a) in the below figure.



a) After *fork()* **b) After closing unused descriptors**

- With this communication, now both child and parent processes can communicate via pipe.
- But **this situation leads to some issues**, as at both the ends, each process's descriptor is associated with the pipe, this may lead to a situation like, both reading and writing at a same time.
- So the **better solution** for this problem is to **close the unwanted ends** immediately after fork. It looks like the above-(b) image, if the parent is intended to only write and the child is to read.

Following code achieves the above difficulty as:

```
int filedesc[2];
if(pipe(filedesc) == -1)    // Creating the pipe
   perror("Error creating a pipe.");
switch(fork())             // Then fork() to create another process
{
   case -1: perror("Error in fork()\n"); break;   // If any error occurs..
```

```
    case 0:  // child process
          if(close(filedesc[1]) == -1)        // Closing unused write end..
                perror("Error closing Write-end of child");
          break;
    default:            // Parent process
           if(close(filedesc[0]) == -1)
                perror("Error closing the read-end of the Parent");
    break;
 }
```

A basic program to create a pipe and share some data through pipe:

basicPipe.c

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#define MAX_SIZE 10

int main()
{
    int filedesc[2];
    char buffer[MAX_SIZE];

    // Create pipe..
    if(pipe(filedesc) == -1)
     perror("Unable to create pipe");

    // Now fork(), to create child process..
    switch(fork())
    {
     case -1:  // In case of fork() failure..
           perror("Unable to create the child process"); break;
     case 0:   // In child process.. close the unwanted end, say read-end
           if(close(filedesc[0]) == -1)
                 perror("Error in closing the read-end of the child process");

           printf("Child process:\n write some thing to send to parent: ");
           scanf("%[^\n]%*c", buffer);
           // write the buffer to the write-end of the process
           if(write(filedesc[1], buffer, strlen(buffer)) != strlen(buffer))
                 perror("Error in writing");
           printf("Data read by child is: %s\n", buffer);
           break;
     default: // Parent's turn..
           if(close(filedesc[1]) == -1)
                 perror("Error in closing the write-end of the process");
```

```
            // Parent reading is blocked until child writes..
            if(read(filedesc[0], &buffer, MAX_SIZE) != MAX_SIZE)  // reads till
                                                                  end of line(\n)
                  perror("Error in reading");
            // Print the read mesage on the console..
            printf("Child sent: %s\n", buffer);
            break;
      }
 }
```

Compiling:

```
gcc basicPipe.c -o basicPipe
```

Running:

```
./basicPipe
```

Child process:

```
write some thing to send to parent: helloworld
Data read by child is: helloworld
Child sent: helloworld
```

CAUTION: Works well only if the data to be sent is 10 characters length.

2. FIFO (Named pipes)

- FIFOs are similar to the pipe, but the basic difference between these two is, FIFO has a name within the filesystem and is opened in the same way as a regular file, whereas pipe it is stored inside the kernel memory.
- As FIFO uses the local filesystem, FIFO can be used for communication between unrelated processes. Like client and server, user and admin..

Creating and using FIFOs

- **Syntax:**

```
#include<sys/stat.h>
int mkfifo(const char* pathname, mode_t mode);
```

- **Description:** Creates a new FIFO with the given pathname.
- **Parameters:**
  - **pathname:** The path where the file should be created.
  - **mode:** Specifies the permissions for the new FIFO.
- **Returns:** 0 on success and -1 on error.

NOTE: Once a FIFO has been created in a filesystem, any process can open it if the process has enough permissions to open and write to the file.

- General convention to use the FIFO is, the process which only want to read from the FIFO, should be specified by **O_RDONLY**, similarly for the process which want to write should be specified by **O_WRONLY.**

NOTE: By default opening a FIFO for reading blocks until another process opens for writing and vice versa.

A basic client server program to demonstrate the concept:

server.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#define MAX_LEN 30

int main()
{
    int filedesc;
    // FIFO file path..
    char *fifo_path = "./tempfifo";
    // buffers to store data..
    char buffer[MAX_LEN];

    // Create the fifo
    if(mkfifo(fifo_path, S_IRUSR | S_IWUSR | S_IWGRP) == -1)
        perror("Error creating the FIFO");

    // If fifo created successfully, then start communication
    while (1<2) // Always true condition for infinite loop
    {
        // Open the FIFO
        filedesc = open(fifo_path, O_WRONLY);

        // Take the input from the server-user
              printf("Server: ");
              fgets(buffer, MAX_LEN, stdin);

              // Send the message to the client..
              write(filedesc, buffer, strlen(buffer)+1);
        // check.. server would like to exit..
        if(strncmp(buffer, "exit", 4) == 0)
               break;

              // Close the file, so as to read the client's reply..
              close(filedesc);
```

```
        // Open again with Read permission.
        filedesc = open(fifo_path, O_RDONLY);

        // Read the message from the client..
        read(filedesc, buffer, sizeof(buffer));
        // Display on console..
        if(strncmp(buffer, "exit", 4) == 0) // If inteded to close
        {
               printf("Client wants to quit.\n...quitting..\n");
               break;
        }
        else
               printf("Client: %s", buffer);

        // Close the FIFO, so as to write to FIFO..
        close(filedesc);
    }
    printf("Connection terminated..\n");

}
```

Compiling:

```
     gcc server.c -o server
```

Client,c

```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#define MAX_LEN 30

int main()
{
    int filedesc;
    // FIFO file path..
    char *fifo_path = "./tempfifo";
    // buffers to store data..
    char buffer[MAX_LEN];

    // Create the fifo --- no need to create again in client..
    //if(mkfifo(fifo_path, S_IRUSR | S_IWUSR | S_IWGRP) == -1)
    //    perror("Error creating the FIFO");

    // If fifo created successfully, then start communication
    while (1<2) // Always true condition for infinite loop
    {
        // Open the FIFO
        filedesc = open(fifo_path, O_RDONLY);
```

```
        // Read the message from the server..
        read(filedesc, buffer, sizeof(buffer));
        // Display on console..
        if(strncmp(buffer, "exit", 4) == 0) // If inteded to close
        {
                printf("Server wants to quit.\n...quitting..\n");
                break;
        }
        else
                printf("Server: %s", buffer);

        // Close the FIFO, so as to write to FIFO..
        close(filedesc);
        //memset(buffer, '\0', strlen(buffer));

        // open again with write permissins..
        filedesc = open(fifo_path, O_WRONLY);
        // Take the input from the client-user
        printf("Client: ");
        fgets(buffer, MAX_LEN, stdin);
        // check if the client would like to exit..
        if(strncmp(buffer, "exit", 4) == 0)
                break;

        // Send the message to the server..
        write(filedesc, buffer, strlen(buffer)+1);
        // Close the file, so as to read the server's reply..
        close(filedesc);
        //memset(buffer, '\0', strlen(buffer));
    }
    printf("Connection terminated..\n");
}
```

Compiling:

```
gcc client.c -o client
```

Running:

./server	./client
Server: _	
Server: Hello client	Server: Hello Client Client: Hello Server
Server: Hello Client Client: Hello Server Server: What are you doing?	Server: Hello Client Client: Hello Server Server: What are you doing? Client: We are playing

Server: Hello Client Client: Hello Server Server: What are you doing? Client: We are playing Server: Oh! Which game?	Server: Hello Client Client: Hello Server Server: What are you doing? Client: We are playing Server: Oh! Which game? Client: Console..
Server: Hello Client Client: Hello Server Server: What are you doing? Client: Playing.. Server: Which game..? Client: Console.. Server: ??	Server: Hello Client Client: Hello Server Server: What are you doing? Client: Playing.. Server: Which game..? Client: Console.. Server: ?? Client: Bye..
Server: Hello Client Client: Hello Server Server: What are you doing? Client: Playing.. Server: Which game..? Client: Console.. Server: ?? Client: Bye.. Server: exit Connection terminated..	Server: Hello Client Client: Hello Server Server: What are you doing? Client: Playing.. Server: Which game..? Client: Console.. Server: ?? Client: Bye.. Server wants to quit. ...quitting.. Connection terminated..

3. Message queues

Message queues allow processes to exchange data in the form of messages. Although message queues are similar to the FIFOs and Pipes, they differ in these below aspects:

- The handle used to refer to a message queue is the identifier returned by a call to `msgget()`. These identifiers are not the same as the file descriptors, used for most other forms of I/O on UNIX systems.
- Communication via message queues is message-oriented. I.e., Reader process receives the whole message as written by the writer process. It is not possible to read part of the message and discard rest of the message, whereas it is possible with the pipes and FIFOs.
- In addition to containing data, each message has an integer type. Messages can be retrieved from a queue in FIFO order or retrieved by type.

Creating or Opening a Message Queue

- **Syntax:**

```
#include<sys/types.h>  // just for portability..
#include<sys/msg.h>

int msgget(key_t key, int msgflg);
```

- **Description:** Creates the message queue or obtains the identifier of the existing queue.
- **Parameters:**
  - **key:** It is a key generated using either IPC_PRIVATE or ftok().
  - **msgflg:** It is a mask bit that specifies the permissions to be placed on a new message queue or to be checked against an existing queue (flags can be ORred). Flags used are:
    - **`IPC_CREAT`** - If no message queue exists, create new.
    - **`IPC_EXCL`** - If **`IPC_CREAT`** was also specified, and a queue with the specified key already exists, fail with the error EEXIST.
- **Returns:** Message queue identifier on Success and -1 if error.

Exchanging Messages

The ***`msgsnd()`*** and ***`msgrcv()`*** system calls are used to perform I/O on message queues. The first argument to both system calls is ***`msqid`***, a message queue identifier and a second argument is ***`msgp`***, is a pointer to a programmer-defined structure used to hold the message being sent or received. General structure will look like:

```
struct msg
  {
        long msgtype;       // Message type
        char msgtxt[];      // Message body
  };
```

Sending Messages

- **Syntax:**

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

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- **Description:** Writes a message to the message queue.
- **Parameters:**
  - **msqid:** Message queue identifier that is used to identify eac message uniquely.
  - **msgp:** Pointer to a programmer defined structure and *msgtype* of the programmer defined structure's value should be >0
  - **msgsz:** Specifies the no. of bytes in the *msgtxt* field of programmer defined structure.

- **msgflg:** A bit mask flag that controls the operation of *msgsnd()*. Only one such flag is defined:
  - IPC_NOWAIT - Perform a non-blocking send. Usually if message queue is full, it waits till some space is available to place the message in the queue. If this flag is specified, it immediately returns with EAGAIN error.
- **Returns:** On success always returns 0 as message is completely transferred, and -1 in case of error

Receiving Messages

- **Syntax:**

```
#include <sys/types.h>  // Just for portability
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void * msgp, size_t maxmsgsz, long msgtyp,
                           int msgflg);
```

- **Description:** It reads (and removes i.e., destructive read) a message from a message queue copies its contents into the buffer pointed by *msgp.*
- **Parameters:**
  - **msqid:** Message Queue Identifier, used to identify each message uniquely.
  - **msgp:** A buffer, where the contents of *msgtxt* (in programmer defined structure) are copied.
  - **maxmsgsz:** Max size of *msgp* (buffer). If the body of the message *msgtxt,* exceeds the length of *msgp* buffer then msgrcv fails with error E2BIG.
  - **msgtyp:** Retrieval of the messages from the Message queue depends on this. If
    - Equals to 0 -- Then the first message from the queue is removed and returned to the calling process.
    - > 0 -- The first message whose *msgtype* equals *msgtyp* is removed and returned to the calling process.
    - < 0 -- Treat the message queue as a priority queue. The first message of the lowest *msgtyp* less than or equal to the absolute value of *msgtyp* is removed and returned to the calling process.
  - **msgflg:** A bit mask flag to change the behaviour of *msgrcv().*
- **Returns:** No. of bytes copied into *mtext* field, and -1 in case of any error.

Program to demonstrate the concept of Message queues..

msg_writer.c

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

// Pre-agreed structure for means of communication b/w reader and writer..
struct msg
{
    long msgtype;
    char msgtxt[40];
}msgqueue;


int main(int argc, char ** argv)
{
    // Generate a unique key..
    key_t key = ftok("msgfile", 34);

    // Get or create a new messge queue..
    int msgid = msgget(key, S_IRUSR | S_IWUSR | S_IWGRP | IPC_CREAT);
    // Assign a unique type number (other that 0)
    msgqueue.msgtype = 1;

    // Take the message from the user..
    printf("Enter some data to place in the message queue: ");
      scanf("%[^\n]%*c", msgqueue.msgtxt);

    // Send the message to message queue..
    msgsnd(msgid, &msgqueue, sizeof(msgqueue), 0);

    // Give the acknoledgemnt to the user..
    printf("Data is sucessfully sent to the message queue\n");
}
```

Compiling:

```
    gcc msg_writer.c -o msg_writer
```

Running:

```
     ./msg_writer
```

```
Received data is Hello from Message queue
```

msg_reader.c

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

// Pre-agreed structure for means of communication b/w reader and writer..
struct msg
{
```

```
    long msgtype;
    char msgtxt[40];
}msgqueue;


int main(int argc, char ** argv)
{
    // Generate a unique key..
    key_t key = ftok("msgfile", 34);

    // Get or create a new messge queue..
    int msgid = msgget(key, S_IRUSR | S_IWUSR | S_IWGRP | IPC_CREAT);

    // Send the message to message queue..
    msgrcv(msgid, &msgqueue, sizeof(msgqueue), 1, 0);

    // Display the received data to the user..
    printf("Received data is %s\n", msgqueue.msgtxt);
}
```

Compiling:

```
gcc msg_reader.c -o msg_reader
```

Running:

```
./msg_reader
```

```
Received data is Hello from Message queue
```

4. Shared Memory

- Shared memory allows two or more processes to share the same region(or called as segment) of physical memory.
- In Shared memory the communication between the processes is done via a piece of memory shared between the processes, and all the processes can view the changes made by another process.
- This provides fast IPC by comparison with techniques such as message queues, pipes and FIFOs where sending process copies data from a buffer in user space to kernel memory and receiving process copies from kernel to process memory.
- As the shared memory is not controlled by the kernel, synchronization is explicitly required else leads ambiguous results.

Creating or Opening a Shared memory segment

- **Syntax:**

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

int shmget(key_t key, size_t size, int shmflg);
```

- **Description:** Creates a new shared memory segment or obtains the identifier of an existing segment and the contents of a newly created segment are set to 0.
- **Parameters:**
  - **key:** A key generated using ftok() or IPC_PRIVATE.
  - **size:** A +ve integer that indicates the size of the segment (in bytes). If the shared memory segment already exists with the unique key then this doesn't have any effect.
  - **shmflg:** Specifies the permissions to be applied on the new shared memory segment or checked against the existing segment. IPC_CREAT and IPC_EXCL are used.
- **Returns:** Returns the shared memory segment on success and -1 on error.

Using Shared memory - Attaching Shared memory to the process

- **Syntax:**

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

void * shmat(int shmid, const void * shmaddr, int shmflg);
```

- **Description:** Attaches the shared memory segment identified by the shmid to the calling process's virtual address space.
- **Parameters:**
  - **shmid:** Shared memory identifier returned by the *shmget().*
  - **shmaddr:** This argument and the setting of the SHM_RND in the shmflg controls how the segment is attached.
    - If *shmaddr* is NULL, then the segment is attached at a suitable address selected by the kernel. --- preferred one.
    - If *shmaddr* is not NULL and SHM_RND is not set, then the segment is attached at the address specified by *shmaddr.*
    - If *shmaddr* is not NULL and the SHM_RND is not set, then the segment is mapped at the address provided in shmaddr, rounded to nearest multiple of the constant SHMLBA(Shared Memory Low Boundary Address)
  - **shmflg:** Used to manipulate the attachment of the shared memory to the process, commonly used flags are

Value	Description
SHM_RDONLY	Attach segment read-only
SHM_REMAP	Replace any existing mapping at *shmaddr*

SHM_RND	Round *shmaddr* to multiple of SHMLBA bytes

- **Returns:** Address at which the shared memory is attached on success or (void *)-1 on error.

Releasing Shared memory - Detaching Shared memory from the process

- **Syntax:**

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

int shmdt (const void * shmaddr);
```

- **Description:** Detaches the attached shared memory attached at *shmaddr.*
- **Parameters:**
  - **shmaddr:** Address where the shared memory is created.
- **Returns:** 0 on success and -1 on error.

Example program to demonstrate the concept of creating, attaching and detaching the shared memory:

shared_writer.c

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

int main(int argc, char ** argv)
{
    // Generate the unique key.
    key_t key = ftok("shmfile", 12);

    // Create or get a new Shared memory ID
    int shmid = shmget (key, 1024, IPC_CREAT | S_IRUSR | S_IWUSR | S_IWGRP);

    // Now attach the shared memory to this process..
    char *str = (char *) shmat (shmid, NULL, 0);  // Returns where shared memory
is attached..

    //printf("%s\n", str);   // Displays the contents that are previously in
shared memory..
    printf("Write some data to write in the shared memory: ");
    scanf("%[^\n]%*c", str);

    printf("Successfully data is written to the shared memory: %s", str);

    // Finally detach the shared memory..
    shmdt(str);
}
```

Compiling: `gcc shared_writer.c -o shared_writer`

Running: `./shared_writer`

Sample Output:

```
Write some data to write in the shared memory: Hello Shared memory..
Successfully data is written to the shared memory: Hello Shared memory..
```

shared_reader.c

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

    int main(int argc, char ** argv)
    {
        // Generate the unique key.
        key_t key = ftok("shmfile", 12);

        // Create or get a new Shared memory ID
        int shmid = shmget (key, 1024, IPC_CREAT | S_IRUSR | S_IWUSR | S_IWGRP);

        // Now attach the shared memory to this process..
        char *str = (char *) shmat (shmid, NULL, 0);  // Returns where shared
memory is attached..

        printf("Data read from the shared memory is :%s\n", str);

        // Finally detach the shared memory..
        shmdt(str);
    }
```

Compiling:

```
    gcc shared_reader.c -o shared_reader
```

Running:

```
     ./shared_reader
```

```
Data read from the shared memory is :Hello Shared memory..
```

Accessibility and persistence for various types of IPC facilities

Facility type	Accessibility	Persistence
Pipe FIFO	only by related processes permissions mask	process process
UNIX domain socket Internet domain socket	permissions mask by any process	process process
System V message queue System V semaphore System V shared memory	permissions mask permissions mask permissions mask	kernel kernel kernel
POSIX message queue POSIX named semaphore POSIX unnamed semaphore POSIX shared memory	permissions mask permissions mask permissions of underlying memory permissions mask	kernel kernel depends kernel

NOTE: Here the term Accessibility refers to the permissions that which process can access the object and the term Persistence refers to the lifetime of an IPC object.

Comparison of Identifiers and handles for various types of IPC facilities

Facility type	Name used to identify object	Handle used to refer to object in programs
Pipe FIFO	no name pathname	file descriptor file descriptor
UNIX domain socket Internet domain socket	pathname IP address + port number	file descriptor file descriptor
System V message queue System V semaphore System V shared memory	System V IPC key System V IPC key System V IPC key	System V IPC identifier System V IPC identifier System V IPC identifier
POSIX message queue POSIX named semaphore POSIX unnamed semaphore POSIX shared memory	POSIX IPC pathname POSIX IPC pathname no name POSIX IPC pathname	mqd_t (message queue descriptor) sem_t * (semaphore pointer) sem_t * (semaphore pointer) file descriptor

References:

https://www.geeksforgeeks.org/ipc-shared-memory/
https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/
https://users.cs.cf.ac.uk/Dave.Marshall/C/node23.html
https://en.wikipedia.org/wiki/Inter-process_communication
http://man7.org/tlpi/
https://www.amazon.in/Linux-Programming-Interface-System-Handbook-ebook/dp/B004OEJMZM
https://www.pdfdrive.com/the-linux-programming-interface-e18763503.html