

A Short Introduction to Makefile

Make Utility and Makefile

- The *make* utility is a software tool for managing and maintaining computer programs consisting many component files. The *make* utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- *Make* reads its instruction from Makefile (called the descriptor file) by default.
- Makefile sets a set of rules to determine which parts of a program need to be recompile, and issues command to recompile them.
- Makefile is a way of automating software building procedure and other complex tasks with dependencies.
- Makefile contains: **dependency rules**, **macros** and **suffix(or implicit) rules**.

```

/* main.cpp */
#include <iostream>
#include "functions.h"

using namespace std;
int main()
{
    print_hello();
    cout << endl;
    cout << "The factorial of 5 is " <<
factorial(5) << endl;
    return 0;
}

```

```

/* hello.cpp */
#include <iostream>
#include "functions.h"

using namespace std;
void print_hello()
{
    cout << "Hello World!";
}

```

```

/* factorial.cpp */
#include "functions.h"

int factorial(int n)
{
    int i, fac = 1;
    if(n!=1){
        for(i=1; i<= n; i++)
            fac *= i;
        return fac;
    }
    else return 1;
}

```

```

/* functions.h */
#ifndef _FUNC_H_
#define _FUNC_H_

void print_hello();
int factorial(int n);

#endif /* if !define(_FUNC_H_) */

```

Command Line Approach to Compile

- `g++ -c hello.cpp main.cpp factorial.cpp`

- `ls *.o`

`factorial.o hello.o main.o`

- `g++ -o prog factorial.o hello.o main.o`

- `./ prog`

Hello World!

The factorial of 5 is 120

- Suppose we later modified `hello.cpp`, we need to:

- `g++ -c hello.cpp`

- `g++ -o prog factorial.o hello.o main.o`

Example Makefile

```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS= -c -Wall

all: prog

prog: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o prog

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm -rf *.o
```

Basic Makefile Structure

Dependency rules

- A rule consists of three parts, one or more targets, zero or more dependencies, and zero or more commands in the form:
target: **dependencies**
<tab> **commands** to make **target**
 - **<tab>** character MUST NOT be replaced by spaces.
 - A “**target**” is usually the name of a file (e.g. executable or object files). It can also be the name of an action (e.g. clean)
 - “**dependencies**” are files that are used as input to create the **target**.
 - Each “**command**” in a rule is interpreted by a shell to be executed.
 - By default, *make* uses /bin/sh shell.
 - Typing “make **target**” will:
 1. Make sure all the dependencies are up to date
 2. If target is older than any dependency, recreate it using the specified commands.

- By default, typing “make” creates first target in Makefile.
- Since prog depends on main.o factorial.o hello.o, all of object files must exist and be up-to-date. *make* will check for them and recreating them if necessary
- Phony targets
 - A phony target is one that isn't really the name of a file. It will only have a list of commands and no dependencies.

E.g. clean:

```
rm -rf *.o
```

Macros

- By using macros, we can avoid repeating text entries and makefile is easy to modify.
- Macro definitions have the form:
NAME = text string
e.g. we have: CC=g++
- Macros are referred to by placing the name in either parentheses or curly braces and preceding it with \$ sign.
 - E.g. \$(CC) main.o factorial.o hello.o -o prog

Internal macros

- Internal macros are predefined in *make*.
- “*make -p*” to display a listing of all the macros, suffix rules and targets in effect for the current build.

Special macros

- The macro **@** evaluates to the name of the current target.

– E.g.

```
prog1 : $(objs)
```

```
    $(CXX) -o $@ $(objs)
```

is equivalent to

```
prog1 : $(objs)
```

```
    $(CXX) -o prog1 $(objs)
```


Suffix rules

A way to define default rules or implicit rules that *make* can use to build a program. There are *double-suffix* and *single-suffix*.

- Suffix rules are obsolete and are supported for compatibility. Use pattern rules (a rule contains character ‘%’) if possible.
- Doubles-suffix is defined by the source suffix and the target suffix . E.g.

.cpp.o:

```
$(CC) $(CFLAGS) -c $<
```

- This rule tells *make* that .o files are made from .cpp files.
- \$< is a special macro which in this case stands for a .cpp file that is used to produce a .o file.

- This is equivalent to the pattern rule “%.o : %.cpp”

```
%.o : %.cpp
```

```
$(CC) $(CFLAGS) -c $<
```

Command line macros

- Macros can be defined on the command line.
 - E.g. make DEBUG_FLAG=-g

How Does Make Work?

- The *make* utility compares the modification time of the target file with the modification times of the dependency files. Any dependency file that has a more recent modification time than its target file forces the target file to be recreated.
- By default, the first target file is the one that is built. Other targets are checked only if they are dependencies for the first target.
- Except for the first target, the order of the targets does not matter. The make utility will build them in the order required.

A New Makefile

```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS=-c -Wall
OBJECTS = main.o hello.o factorial.o
all: prog

prog: $(OBJECTS)
    $(CC) $(OBJECTS) -o prog

%.o: %.cpp
    $(CC) $(CFLAGS) $<

clean:
    rm -rf *.o
```

- [Reference](http://www.gnu.org/software/make/manual/html_node/)

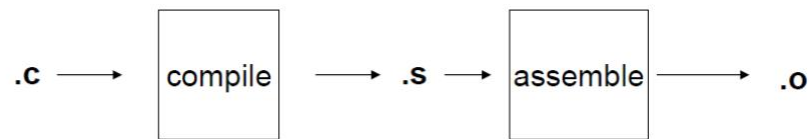
http://www.gnu.org/software/make/manual/html_node/

WEEK 1: The make utility

Date: 27/08/2020

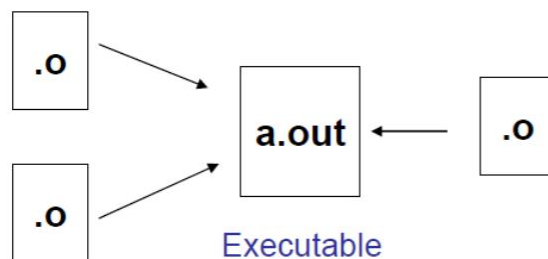
Compiling

- High level \longrightarrow Machine level



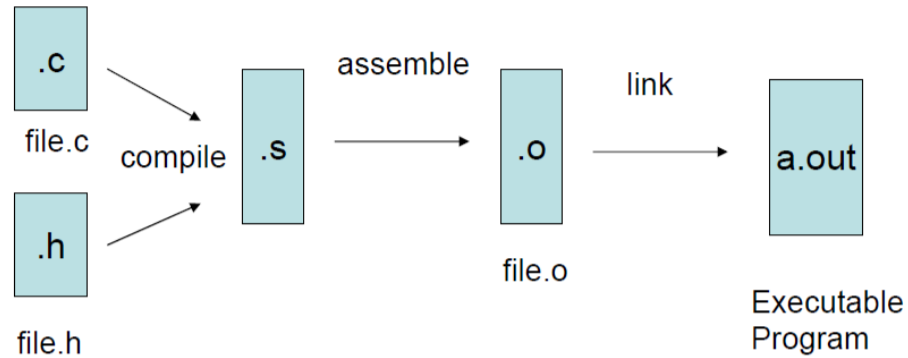
- Looks **one file at a time**
- **Function calls not resolved**
- `gcc -c file.c`

Linking



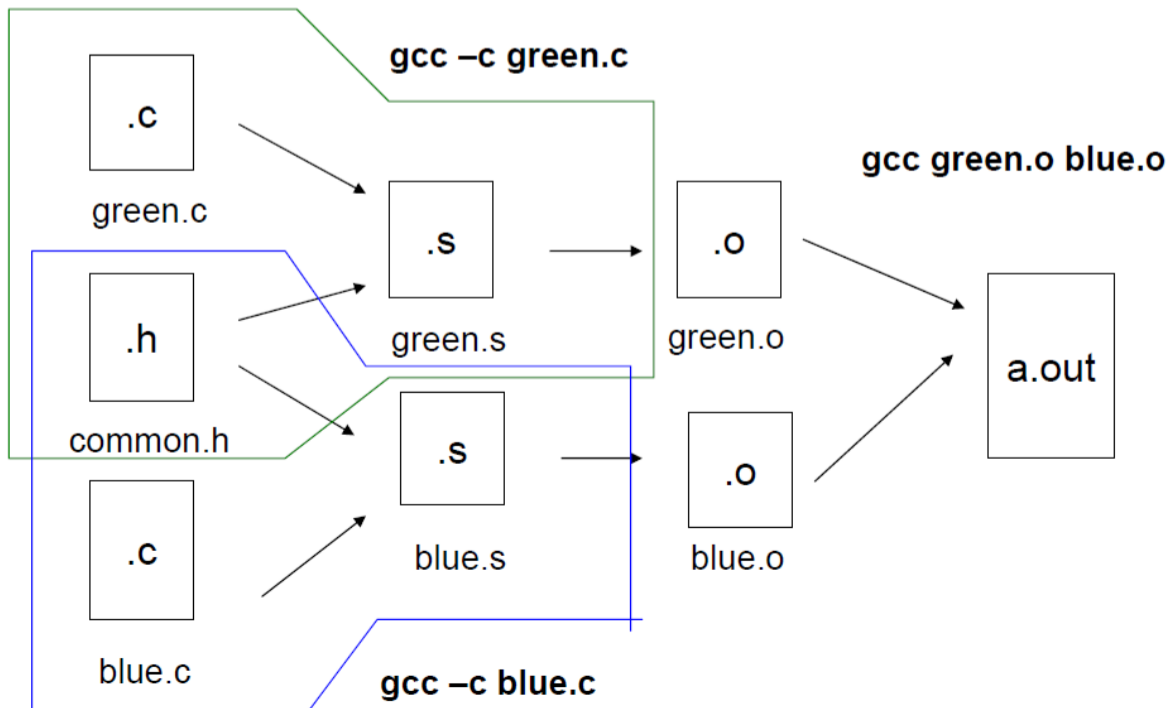
- Many files at a time
- **Resolves all cross - references**
- `gcc <file1.o> <file2.o> -o <output>`

A simple compilation



Command – `gcc file.c`

Compiling with several files



Motivation

- Small programs → single file
- "Not so small" programs :
 - Many lines of code
 - More than one programmer
- Problems:
 - Long files are harder to manage
 - Every change requires long compilation
 - Many programmers can not modify the same file simultaneously
- Solution : divide project to multiple files
- Targets:
 - Good division to components
 - Minimum compilation when something is changed

Multiple Source files

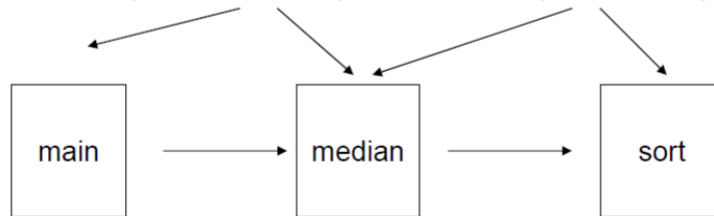
- C Programs - 2 types of files
- .c files :
 - Contain source code and global variable definitions
 - Never included
- .h files :
 - Contain function declarations, struct definitions, # define constant definitions

Project maintenance

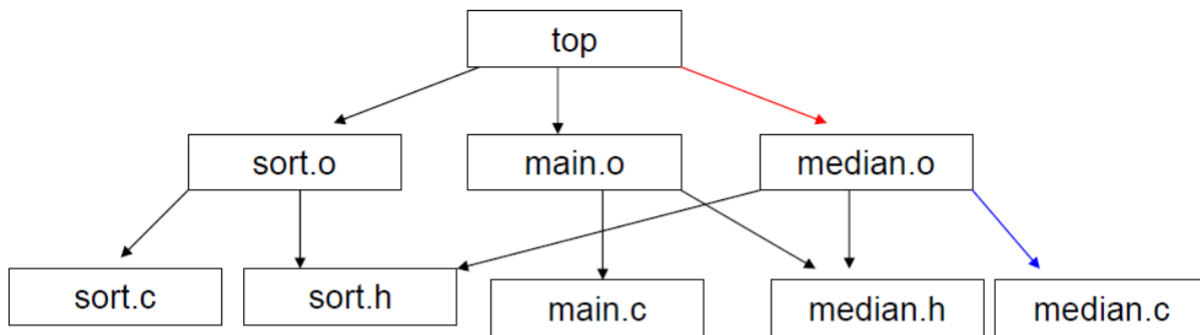
- Done in Unix by the Makefile mechanism
- A makefile is a file (script) containing :
 - Project structure (files, dependencies)
 - Instructions for files creation
- The make command reads a makefile, understands the project structure and makes up the executable
- Makefile mechanism not limited to C programs

Project structure

- Project structure and dependencies can be represented as a graph
- Example given in previous lab session :
 - Program contains 5 files
 - `main.c`, `median.h`, `median.c`, `sort.h`, `sort.c`



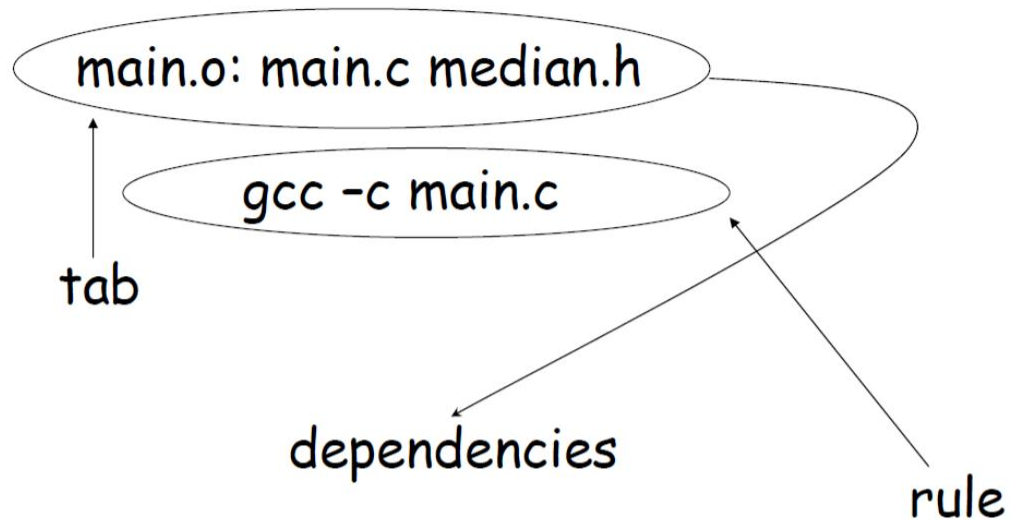
Dependency Graph



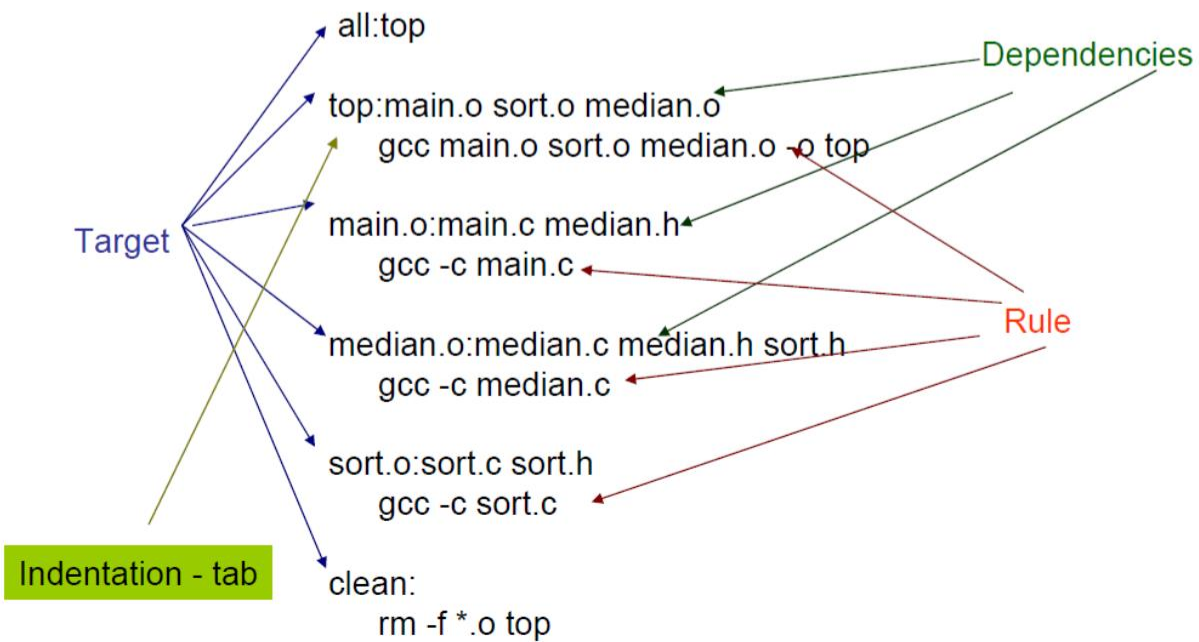
(edited)

- If `median.c` is edited
 - `gcc -c median.c`
 - `gcc median.o main.o sort.o -o top`

Makefile syntax



Makefile Eg



Makefile (contd)

```
CC=gcc
CFLAGS=-c -Wall

all:top

top:main.o sort.o median.o
    $(CC) main.o sort.o median.o -o top

main.o:main.c
    $(CC) $(CFLAGS) main.c

sort.o:sort.h sort.c
    $(CC) $(CFLAGS) sort.c

median.o:median.h median.c
    $(CC) $(CFLAGS) median.c

clean:
    rm -f *.o top
```

```
CC=gcc
CFLAGS= -Wall
OBJS=median.o main.o sort.o

all:top

top:$(OBJS)
median.o:median.h sort.h
sort.o:sort.h
main.o:median.h

clean:
    rm -f *.o top
```

Shell Scripting

What is Shell Script?

Normally shells are interactive. It means shell accepts command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands), then you can store this sequence of command to text file and have the shell execute this text file instead of entering the commands. This is known as **shell script**.

Shell script defined as:

*"Shell Script is **series of commands** written **in plain text file**. Shell script is just like a batch file is MS-DOS but more powerful than the MS-DOS batch file."*

Why to Write Shell Script?

- Shell script can take input from the user, file and output them on to screen or a file.
- Useful to create our own commands.
- Saves lots of time.
- To automate some tasks of day to day life.
- System Administration tasks can be automated.

For our **first shell script**, we'll write a "Hello World" script.

Create a file **first.sh** (using any editor of your choice) as follows:

[first.sh](#)

```
#!/bin/sh
# This is a comment!
echo Hello World    # This is a comment, too!
```

The first line tells Unix that the file is to be executed by `/bin/sh`. This is the standard location of the Bourne shell on just about every Unix system. If you're using GNU/Linux, `/bin/sh` is normally a symbolic link to `bash` (or, more recently, `dash`).

The second line begins with a special symbol: `#`. This marks the line as a comment, and it is ignored completely by the shell.

The only exception is when the *very first* line of the file starts with `#!` - as ours does. This is a special directive which Unix treats specially. It means that even if you are using `csh`, `ksh`, or anything else as your interactive shell, that what follows should be interpreted by the Bourne shell.

Similarly, a Perl script may start with the line `#!/usr/bin/perl` to tell your interactive shell that the program which follows should be executed by perl.

For Bourne shell programming, we shall stick to `#!/bin/sh`.

The third line runs a command: `echo`, with two parameters, or arguments - the first is `"Hello"`; the second is `"World"`.

Note that `echo` will automatically put a single space between its parameters.

The `#` symbol still marks a comment; the `#` and anything following it is ignored by the shell.

Now run `chmod 755 first.sh` to make the text file executable, and run `./first.sh`.

Your screen should then look like this:

```
$ chmod 755 first.sh
$ ./first.sh
Hello World
$
```

You will probably have expected that! You could even just run at the shell prompt:

```
$ echo Hello World
Hello World
$
```

How to write shell script?

Following steps are required to write shell script:

(1) Use any editor like vi, gedit or mcedit to write shell script.

(2) After writing shell script set execute permission for your script as follows
syntax:

`chmod permission your-script-name`

Examples:

`$ chmod +x your-script-name`

`$ chmod 755 your-script-name`

Note: This will set read, write & execute permission (7 or binary 111) for owner
Read and execute permission only (5 or binary 101) for group and other.

How to Execute your script?

syntax:

bash your-script-name

sh your-script-name

./your-script-name

Examples:

\$ bash bar

\$ sh bar

\$./bar

NOTE In the last syntax `./` means current directory, But only `.` (dot) means execute given command file in current shell without starting the new copy of shell, The syntax for `.` (dot) command is as follows

Syntax:

`. command-name`

Example:

\$ `. foo`

Now you are ready to write a shell script that will print "Knowledge is Power" on screen. See the [common vi command list](#) , if you are new to vi.

```
$ vi second
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

After saving the above script, you can run the script as follows:

\$ `./second`

This will not run script since we have not set execute permission for our script *second*; to do this type command

\$ `chmod 755 second`

\$ `./second`

First screen will be clear, then Knowledge is Power is printed on screen.

💡 [How Shell Locates the file](#) (My own bin directory to execute script)

Tip: For shell script file try to give file extension such as `.sh`, which can be easily identified by you as shell script.

Exercise 1:

1) Write the following shell script, save it, execute it and note down the output.

```
# Script to print user information who currently login, current date & time
# Enter the following commands in a file
#
clear
echo "Hello $USER"
echo "Today is ";date
echo "Number of user login : " ; who | wc -l
echo "Calendar"
cal
exit 0
```

At the end, why statement exit 0 is used? What is the meaning of \$? See [exit status](#) for more information.

VARIABLES IN SHELL

In Linux (Shell), there are two types of variable:

- (1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
- (2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters. Some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/ubuntu	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\\$	Our prompt settings
PWD=/home/ubuntu/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=ubuntu	User name who is currently login to this PC

NOTE that Some of the above settings can be different in your PC/Linux environment. You can print any of the above variables contains as follows:

```
$ echo $USERNAME  
$ echo $HOME
```

Exercise 2:

1) If you want to print your home directory location then you give command:

a) echo \$HOME

OR

(b)echo HOME

Which of the above command is correct & why?

Caution: Do not modify System variable, this can some time create problems.

How to define User defined variables (UDV)

To define UDV use following syntax

Syntax:

variable name=value

'value' is assigned to given 'variable name' and Value must be on right side = sign.

Example:

```
$ no=10 # this is ok
```

```
$ 10=no # Error, NOT Ok, Value must be on right side of = sign.
```

To define variable called 'vehicle' having value Bus

```
$ vehicle=Bus
```

To define variable called n having value 10

```
$ n=10
```

To print or access UDV use following syntax

To print contains of variable 'vehicle' type

```
$ echo $vehicle
```

It will print 'Bus'

To print contains of variable 'n' type command as follows

```
$ echo $n
```

It will print '10'

echo Command

Use echo command to display text or value of variable.

echo [options] [string, variables...]

Displays text or variables value on screen.

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a alert (bell)

\b backspace

\c suppress trailing new line

\n new line

\r carriage return

\t horizontal tab

\\ backslash

For e.g. `$ echo -e "An apple a day keeps away \a\t\tdoctor\n"`

💡 How to display colourful text on screen with bold or blink effects, how to print text on any row, column on screen, [click here for more!](#)

Shell Arithmetic

Syntax:

expr op1 math-operator op2

Exercise 3:

What is the output of the following expressions?

`$ expr 1 + 3`

`$ expr 2 - 1`

`$ expr 10 / 2`

`$ expr 20 % 3`

`$ expr 10 * 3`

`$ echo `expr 6 + 3``

Exercise 4:

What is the meaning of Single quote ('), Double quote (") and Back quote (`) in shell?

Wild cards (Filename Shorthand or meta Characters)

Wild card /Shorthand	Meaning	Examples	
*	Matches any string or group of characters.	\$ ls *	will show all files
		\$ ls a*	will show all files whose first name is starting with letter 'a'
		\$ ls *.c	will show all files having extension .c
		\$ ls ut*.c	will show all files having extension .c but file name must begin with 'ut'.
?	Matches any single character.	\$ ls ?	will show all files whose names are 1 character long
		\$ ls fo?	will show all files whose names are 3 character long and file name begin with fo
[...]	Matches any one of the enclosed characters	\$ ls [abc]*	will show all files beginning with letters a,b,c

Redirection of Standard output/input i.e. Input - Output redirection

It's possible to send output to a file or to read input from a file.

For e.g.

\$ ls command gives output to screen; to send output to file of ls command use
\$ ls > filename

It means put output of ls command to filename.

There are three main redirection symbols **>**, **>>** and **<**

(1) **>** Redirection symbol

To output Linux-commands result (output of command or shell script) to file.

Note that if file already exists, it will be overwritten, else new file is created.

For e.g. To send output of ls command use

Department of Computer Science and Engineering Page 7 Aug – Dec 2020

\$ ls > myfiles

Now if 'myfiles' file exist in your current directory it will be overwritten without any type of warning.

(2) >> Redirector Symbol

Syntax:

Linux-command >> filename

To output Linux-commands result (output of command or shell script) to END of file. Note that if file exists, it will be opened and new information/data will be written to END of file, without losing previous information/data. If file does not exist, then a new file is created. For e.g. To send output of date command to already exist file use command

\$ date >> myfiles

(3) < Redirector Symbol

Syntax:

Linux-command < filename

To take input to Linux-command from file instead of key-board. For e.g. To take input for cat command use command

\$ cat < myfiles

Exercise 5:

What does the following command do?

\$ sort < myfile > sorted_file

Pipes

A pipe is a way to connect the output of one program to the input of another program without any temporary file.

"A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for the second command. Pipes are used to run more than two commands (Multiple commands) from the same command line."

Syntax:

command1 | command2

<i>Examples: Command using Pipes</i>	Meaning or Use of Pipes
\$ ls more	Output of ls command is given as input to more command so that output is printed one screen (full page) at a time.
\$ who sort	Output of who command is given as input to sort command so that it will print sorted list of users
\$ who sort > user_list	Same as above except output of sort is sent to (redirected) user_list file
\$ who wc -l	Output of who command is given as input to wc command so that it will show number of user logged into the system
\$ ls -l wc -l	Output of ls command is given as input to wc command so that it will print number of files in current directory.
\$ who grep ubuntu	Output of who command is given as input to grep command so that it will print if particular user name is logged in, if not, nothing is printed

If-else-fi construct

Create a shell script using if-else-fi statements and execute it

```
osch=0

echo "1. Unix (Sun OS)"
echo "2. Linux (Red Hat)"
echo -n "Select your os choice [1 or 2]? "
read osch

if [ $osch -eq 1 ] ; then

    echo "You selected Unix (Sun OS)"

else #### nested if i.e. if within if #####

    if [ $osch -eq 2 ] ; then
        echo "You selected Linux (Red Hat)"
    else
        echo "You don't like Unix/Linux OS."
    fi
fi
```

Looping – while, until and for Loops

Create a shell script to print Welcome 5 times

```
for (( i = 0 ; i <= 4; i++ ))
do
    echo "Welcome $i times"
done
```

Create a shell script to display numbers 0 to 9

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

Process Creation & Termination – fork, exec, wait

The fork() System Call

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a *new* process, which becomes the *child* process of the caller. After a new child process is created, *both* processes will execute the next instruction following the **fork()** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the *process ID* of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Therefore, after the system call to **fork()**, a simple test can tell which process is the child. **Please note that Unix/Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.**

Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

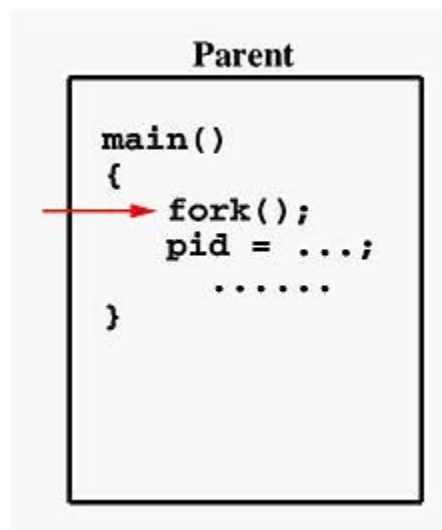
```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define MAX_COUNT 200
#define BUF_SIZE 100

void main(void)
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];

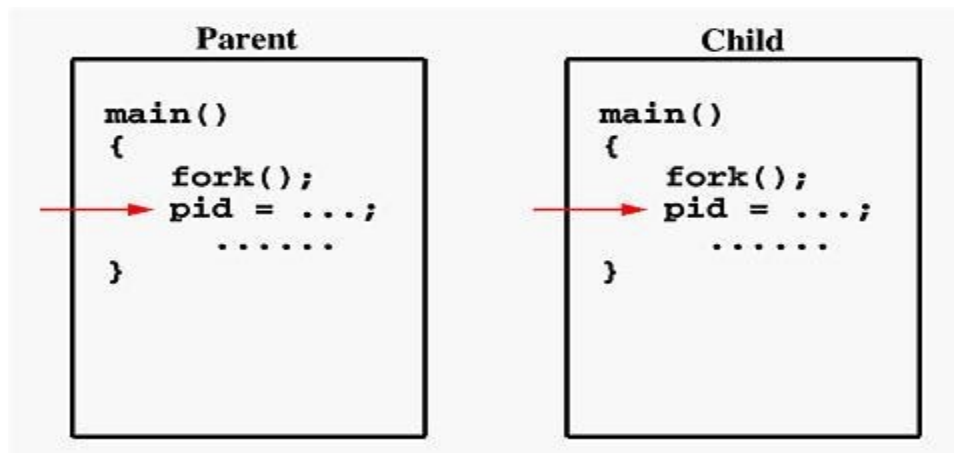
    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

Suppose the above program executes up to the point of the call to **fork()** (marked in red color):



If the call to **fork()** is executed successfully, Unix will

- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf()** is "buffered," meaning **printf()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be sent to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed up in strange ways. To overcome this problem, you may consider using the "unbuffered" **write**.

If you run this program, you might see the following (but with different PIDs as per your system) on the screen:

```

.....
This line is from pid 3456, value 13
This line is from pid 3456, value 14
.....
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
.....
This line is from pid 3456, value 21
This line is from pid 3456, value 22
.....

```

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

Consider another simple example, which distinguishes the parent from the child.

```

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

#define MAX_COUNT 200

void ChildProcess(void);          /* child process prototype */
void ParentProcess(void);        /* parent process prototype */

void main(void)
{
    pid_t pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void ChildProcess(void)
{
    int i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("    This line is from child, value = %d\n", i);
    printf("    *** Child process is done ***\n");
}

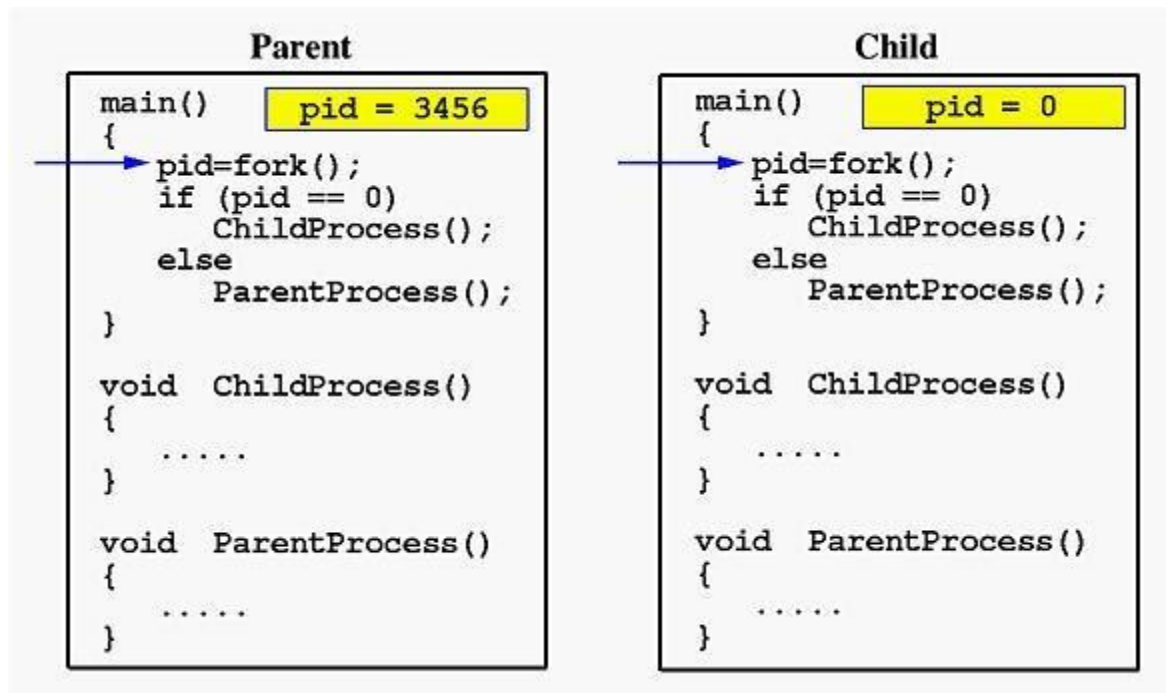
void ParentProcess(void)
{
    int i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}

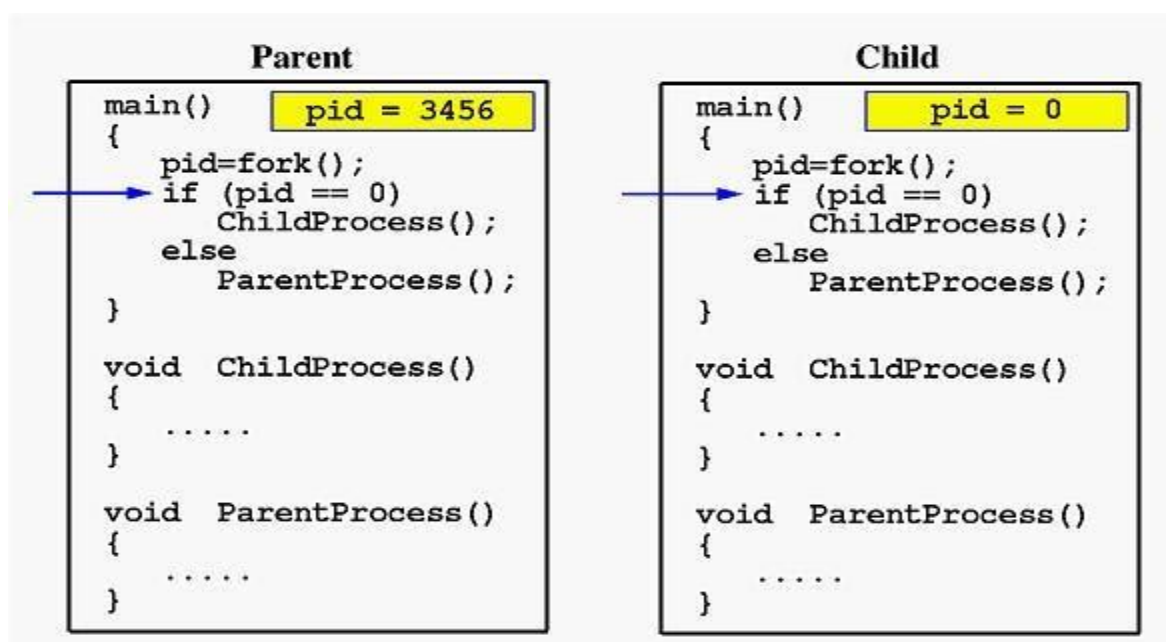
```

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable **i**. For simplicity, **printf()** is used.

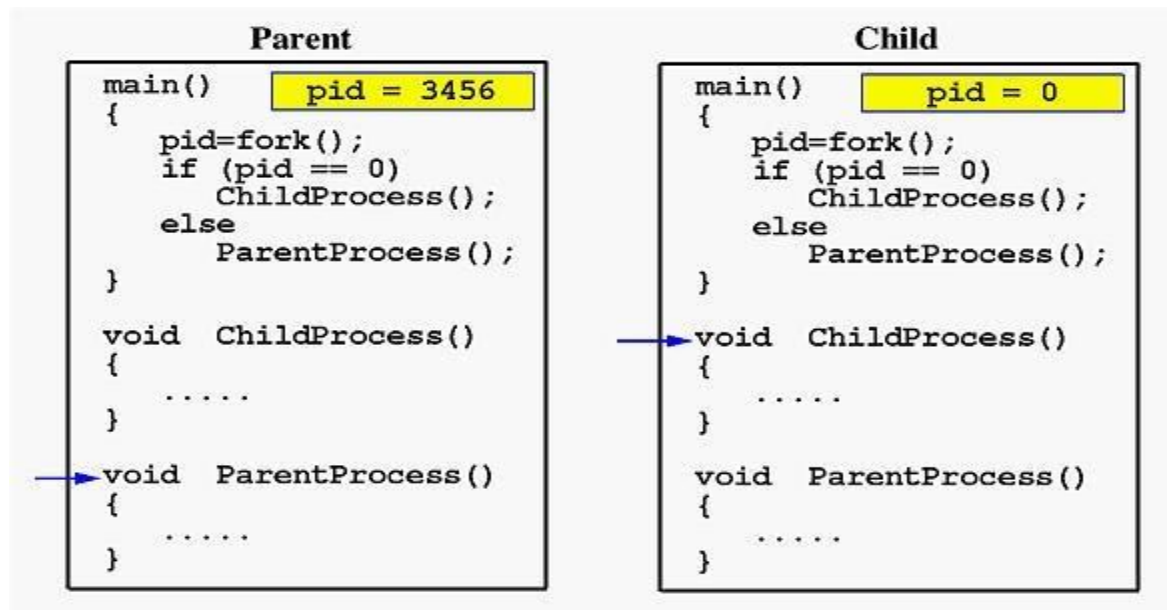
When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created. System call **fork()** returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (*i.e.*, the parent and child) will execute independent of each other starting at the next statement:



In the parent, since **pid** is non-zero, it calls function **ParentProcess()**. On the other hand, the child has a zero **pid** and calls **ChildProcess()** as shown below:



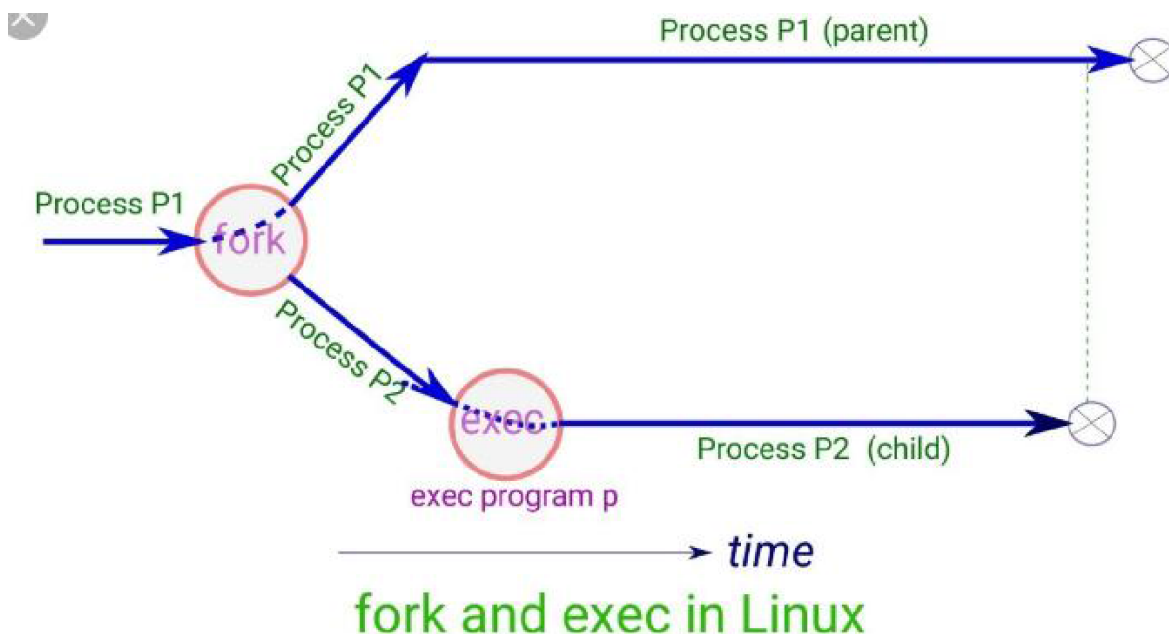
Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, the value of **MAX_COUNT** should be large enough so that both processes will run for at least two or more time quanta. If the value of **MAX_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.

The `exec()` system call

The **exec system call** is used to execute a file which is residing in an active process. When **exec** is called the previous executable file is replaced and new file is executed. More precisely, we can say that using **exec system call** will replace the old file or program from the process with a new file or program.

► The prototypes are:

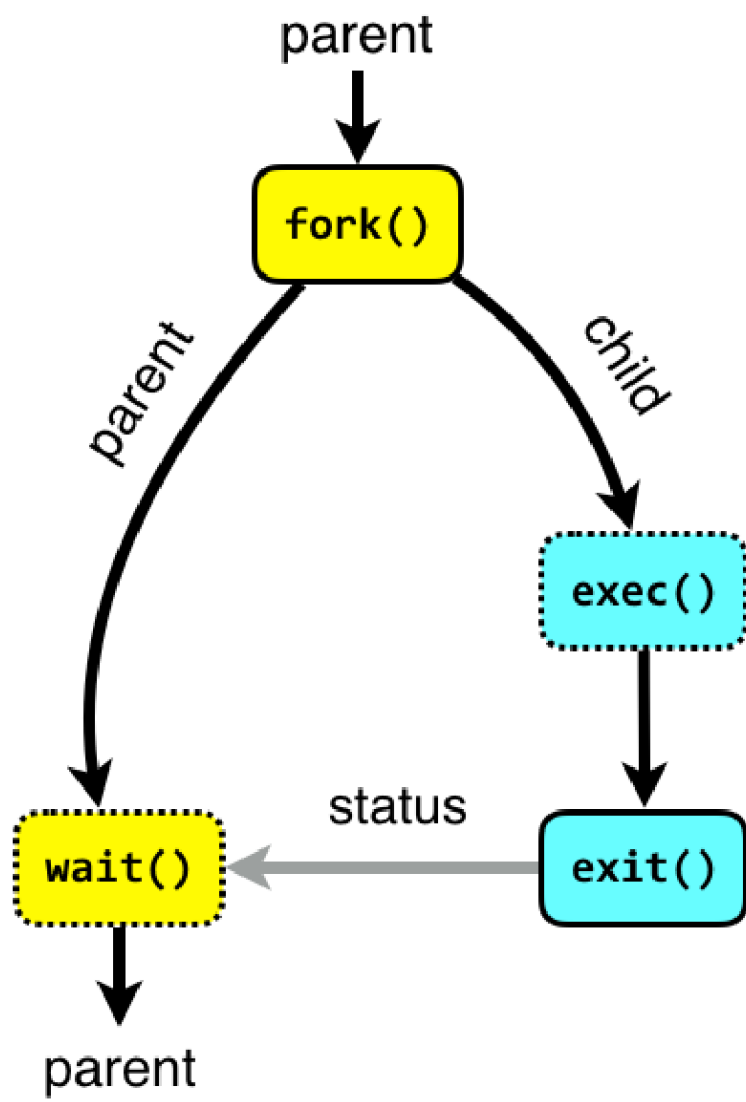
- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execl(const char *path, const char *arg, char *const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execve(const char *path, const char *argv[], char *const envp[]);`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



Wait System Call in C

1. It **calls** `exit()`;
2. It returns (an int) from main.
3. It receives a signal (from the OS or another process) whose default action is to terminate.

Suspends the **calling** process until a child process ends or is stopped. More precisely, `waitpid()` suspends the **calling** process until the **system** gets status information on the child. If the **system** already has status information on an appropriate child when `waitpid()` is called, `waitpid()` returns immediately.



EXAMPLE 1:

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

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

int main() {
    pid_t forkStatus;

    forkStatus = fork();

    /* Child... */
    if (forkStatus == 0) {
        printf("Child is running, processing.\n");
        sleep(5);
        printf("Child is done, exiting.\n");

        /* Parent... */
    } else if (forkStatus != -1) {
        printf("Parent is waiting...\n");

        wait(NULL);
        printf("Parent is exiting...\n");
    } else {
        perror("Error while calling the fork function");
    }

    return 0;
}
```

You can compile “gcc forksleep.c” without using any arguments shown below and notice the differences

```
$ gcc -std=c89 -Wpedantic -Wall forksleep.c -o forksleep -O2
$ ./forksleep
Parent is waiting...
Child is running, processing.
Child is done, exiting.
Parent is exiting...
```

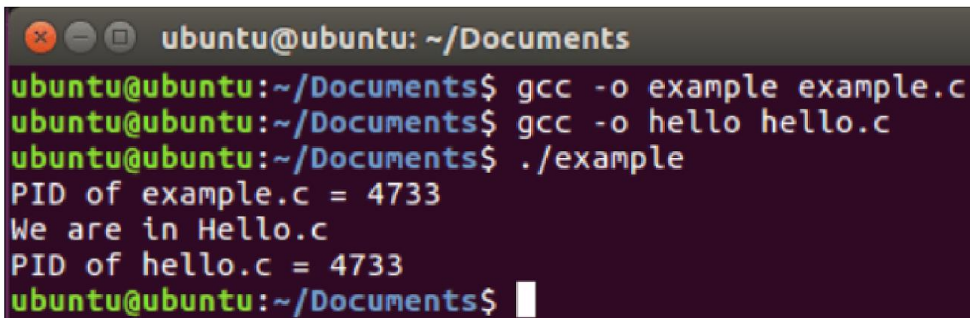
EXAMPLE 2:

Example.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

Hello.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```



```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```