HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Lab 1
C Programming on Linux, Mac OS X
Process

October 1, 2022

Goal: The lab helps student to

- Review basic shell commands and practice with *vim* on Linux, Mac OS X.
- Review C programming with compiling and running a program on Linux, Mac OS X.
- Understand the definition of process, and how an operating system can manage the execution of a process.

Content:

- Practice with *vim* - text editor.
- Programming with C language.
- Compile a program with Makefile.
- How to retrieve the information of running process? How is PCB table controlled by OS?
- Create a program with multiple processes.

Result: After doing the lab, students are able to:

- Write a program without GUI on Linux/Mac OS C by *vim*.
- Compile and run a program using Makefile.
- Distinguish a program and a process. They can create a program with multiple process and retrieve the information of processes.

Contents

1. Introduction

1.1. Basic commands

- Identify an account: $whoami
- Display Home directory: $echo $HOME
- Change user password: $passwd
- Remote to a distance machine: $ssh <username>@<computer-name/IP address>
- Display the user manual of any command running on the terminal: $man echo (display the user manual of command "echo") or $manpage echo ()search on website).
- List the currently running processes and their PIDs along with some other information depends on different options: $ps, $top
- To view all previous activities, use *history* command: $ history > history.txt //send output of *history* command to file "history.txt"

1.2. File operations

- View content of file: *$cat <file-name>*. Example: $cat toto
- List the content of a given directory: *$ls <directory-name>*. Example $ls /home
- View the information of a given file: *$ls -lh <file-name>*.
- Create a new directory: *$mkdir <directory-name>*.
- Create a new file to edit: *$vim <file-name>*. For installing *vim*, type *$sudo apt install vim*
Or use *nano* command: *$nano <file-name>*
- Copy file: *$cp <src-file> <dst-file>*. Example: $cp toto tata
- Rename file: *$mv <src-file> <dst-file>*: $mv toto tata //rename file from "toto" to "tata"
- Move file: *$mv tata file/tata* to move file from the current path to the directory "file"
- Delete directory/file: *$rm -r <direc-name/filename>*.

1.3. Vim

Vim is the editor of choice for many developers and power users. It‘s a “modal” text editor based on the vi editor written by Bill Joy in the 1970s for a version of UNIX. It inherits the key bindings of vi, but also adds a great deal of functionality and extensibility that are missing from the original vi.
Vim has two modes for users:

- Command mode: allows user to do functions such as find, undo, etc.
- Insert mode: allows user to edit the content of text.

To turn the *Insert* mode to *Command* mode, we type *ESC* key or $Crtl-C$. Otherwise, to enter the *Insert* mode, type i or I, a, A, o, O. Some of basic commands in *Vim*:

- Save: enter :w
- Quit without Save and discard the change: enter :q!
- Save and Quit: enter :wq
- Move the cursor to the top of file: gg
- Move to the bottom: G
- Find a letter/string by going forward: enter /[letter/string] <Enter>
- Find a letter/string by going backward: enter ?[letter/string] <Enter>
- Repeat the previous finding: enter n
- Repeat the previous finding by going backward: enter N
- Delete a line: enter dd

- Undo: enter u
- Redo: enter Ctrl-R

Furthermore, *Vim* has a mode called “visual” that allows user to chose a paragraph for copying or, moving. To enter this mode, we need to turn the editor into *Command* mode and press “v”. After that, user use “arrow” keys to chose the paragraph, and then use the following commands:

- Copy: enter y
- Cut: enter d
- Paste: enter p

1.4. C programming on Linux/Mac OS X

GNU C Coding Standards

- **Keep the length of source lines to 79 characters or less, for maximum readability in the widest range of environments.**
- Put a comment on each function saying what the function does, what sorts of arguments it gets, and what the possible values of arguments mean and are used for.
- Please explicitly declare the types of all objects. For example, you should explicitly declare all arguments to functions, and you should declare functions to return int rather than omitting the int.

Reference: `http://www.gnu.org/prep/standards/standards.html`. Formatting your source code

Compiling process: It is important to understand that while some computer languages (e.g. Scheme or Basic) are normally used with an interactive interpreter (where you type in commands that are immediately executed). C source codes are always compiled into binary code by a program called a "compiler" and then executed. This is actually a multi-step process which we describe in some detail here.

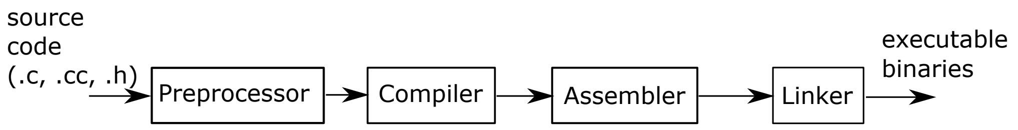

Figure 1.1: C Compiling scheme

Steps in compiling process:

- Preprocessor: takes the source code, the preprocessor directives and then expand the source code.
- Compiler: converts the expanded code into assembly code.
- Assembler: converts the assembly code into object code by using an assembler
- Linker: combines the object code of library files with the object code of our program.

COMPILERS AND LIBRARIES: Apple provides a customized/optimized GNU CC, with backends for C, C++, Objective-C and Objective-C++. Compilers for many other languages are available either precompiled (such as the XL Fortran Advanced Compiler from IBM), or can be compiled from source, which is not any harder in general than compiling the same source on, say, Linux or FreeBSD. The LLVM compiler is the next-generation compiler, introduced in Mac OS X. In Xcode of Mac OS X, the LLVM compiler uses the Clang front end to parse source code and turn it into an interim format.

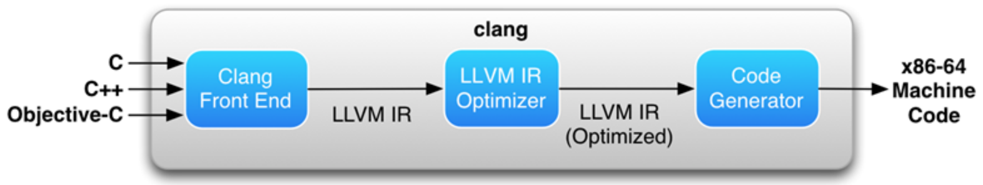

Figure 1.2: Clang in Mac OS X

Figure below shows a C program compiled in step by step.

```
% Preprocessed source file
$ gcc -E [-o hello.cpp] hello.c  //output hello.i

% Assembly code
$ gcc -S [-o hello.S] hello.c //output hello.s

% Binary file
$ gcc -c [-o hello.o] hello.c  //output hello.o

% Executable file
$ gcc [-o hello] hello.c
```

1.5. PROCESS

Informally, as mentioned earlier, a process is a program in execution. A process is more than the program code, which is sometimes known as the text section. It also

includes the current activity, as represented by the value of the program counter and the contents of the processor‘s registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.
As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

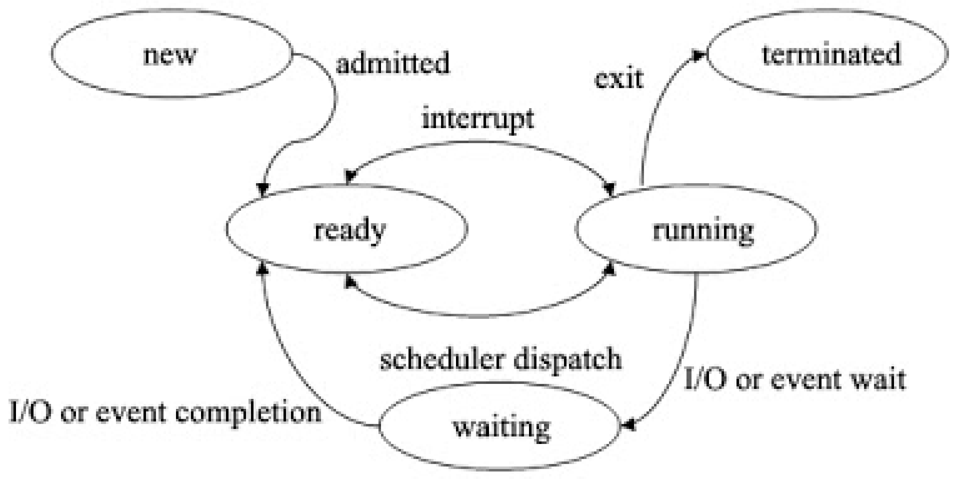

Figure 1.3: Diagram of process state.

To keep track of processes, the operating system maintains a process table (or list). Each entry in the process table corresponds to a particular process and contains fields with information that the kernel needs to know about the process. This entry is called a Process Control Block (PCB). Some of these fields on a typical Linux/Unix system PCB are:

- Machine state (registers, program counter, stack pointer)
- Parent process and a list of child processes
- Process state (ready, running, blocked)
- Event descriptor if the process is blocked
- Memory map (where the process is in memory)
- Open file descriptors
- Owner (user identifier). This determines access privileges & signaling privileges

- Scheduling parameters
- Signals that have not yet been handled
- Timers for accounting (time & resource utilization)
- Process group (multiple processes can belong to a common group)

A process is identified by a unique number called the process ID (PID). Some operating systems (notably UNIX-derived systems) have a notion of a process group. A process group is just a way to lump related processes together so that related processes can be signaled. Every process is a member of some process group.

2. Practice

2.1. Compile and run a program

Steps for creating a program

In general, the compiling progress includes these steps:

1. Create source code file hello.c

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

int main(int argc, char ** argv) {
  printf("Hello, World!\n");
  return 0;
}
```

2. Create object file:

```
$ gcc -c souce_code_file.c
# Example:
$ gcc -c hello.c
# or
$ gcc -c -o hello.o hello.c
```

3. Create executable file:

```
$ gcc -o executable_file object1.o object2.o ...
# Example:
$ gcc -o hello hello.o
```

We can compile the program directly from the source code file without the step of creating object file. However, this way can cause the difficulty when identifying errors.

4. Create executable file:

```
$ gcc -o executable_file src1.c src2.c ...
# Example:
$ gcc -o hello hello.c
```

5. Run the program:

```
$ ./executable_file
# Example: to list the crated executable binary file
$ ls
hello    hello.c          hello.o
# To execute the binary file
$ ./hello
```

- During compiling a program, the source code can make some errors. The compiler provides debuggers that show the information of errors. The structure of showing errors: `<file>:<row>:<column_letter>:<type>:<detail>`
- For example, error 1:

```
$ gcc -o hello.o -c hello.c
hello.c:1:18: fatal error: stdo.h: No such file ...
compilation terminated.
```

- From the example of error 1:
  - Error file: hello.c
  - Error line: 1
  - The column of error letter: 18
  - Type of error: error
  - Detail info: stdo.h not found

2.2. Makefile

A makefile is a file containing a set of directives used with the make build automation tool. Most often, the makefile directs make on how to compile and link a program. Using C/C++ as an example, when a C/C++ source file is changed, it must be recompiled. If a header file has changed, each C/C++ source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable program.[1] These instructions with their dependencies are specified in a makefile. If none of the files that are prerequisites have been changed since the last time the program was compiled, no actions take place. For large software projects, using Makefiles can substantially reduce build times if only a few source files have changed. A makefile consists of “rules” in the following form:

```
# comment
# (note: the <tab> in the command line
# is necessary for make to work)

target:  dependency1 dependency2 ...
       <tab> command
```

Where,

- target: a target is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as "clean".
- dependency1, dependency2,...: a dependency (also called prerequisite) is a file that is used as input to create the target. A target often depends on several files. However, the rule that specifies a recipe for the target need not have any prerequisites. For example, the rule containing the delete command associated with the target "clean" does not have prerequisites.
- command: Needed commands is used for performing rules.

For example, we have three source code files including main.c, hello.h, hello.c.

```
// File: main.c
#include "hello.h"

int main() {
        helloworld();
        return 0;
}
```

```
// File: hello.h

void helloworld(void);
```

```
// File: hello.c
#include "hello.h"
#include <stdio.h>
```

```

void helloworld(void) {
        printf("Hello, world\n");
}
```

In this example, we compile .c files into object files .o, and then link all of object files into a single binary. Firstly, that is the process of compiling source code files into object files.

- main.o: main function in main.c calls helloworld() which is declared in hello.h. Thereby, to compile main.c, we need the information declared from hello.h. To create main.o, we need hello.h and main.c. Therefore, the rule for creating main.o is:

  ```
  main.o: main.c hello.h
          gcc -c main.c
  ```

- hello.o: similar to the rule of main.o, we need two files named hello.c and hello.h to create hello.o. Note that hello.c using printf() in the library stdio.h to print the output on screen. However, this is the library integrated with GCC, so we do not need to fill in the dependency of the rule.

  ```
  hello.o: hello.c hello.h
          gcc -c hello.o
  ```

- hello: Because helloworld is declared in hello.h, but it is defined in hello.c and compiled into the binary in hello.o, therefore, if the main function calls this function, we need to link hello.o with main.o to create the final binary. This file depends on hello.o and main.o.

  ```
  all: main.o hello.o
          gcc main.o hell.o -o hello
  ```

- Finally, we can add the rule of clean to remove all of object files and binaries in case of compiling an entire program.

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

The final result of Makefile:

```
# File: Makefile
all: main.o hello.o
        gcc main.o hello.o -o hello

main.o: main.c hello.h
```

```
        gcc -c main.c

hello.o: hello.c hello.h
        gcc -c hello.c

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

With this Makefile, to re-compile the whole program, we call:

```
$ make all
```

To remove all of object files and binaries, we call

```
$ make clean
```

If we need to create an object file - main.o, we call

```
$ make main.o
```

If we only call “make”, the default rule of Makefile is executed - “make all”.

2.3. How do we find the process'ID and group?

A process can find its process ID with the *getpid* system call. It can find its process group number with the *getpgrp* system call, and it can find its parent‘s process ID with *getppid*. For example:

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

int main(int argc, char **argv) {
        printf("Process ID: %d\n", getpid());
        printf("Parent process ID: %d\n", getppid());
        printf("My group: %d\n", getpgrp());

        return 0;
}
```

2.4. Create process

The *fork* system call clones a process into two processes running the same code. *Fork* returns a value of 0 to the child and a value of the process ID number (pid) to the parent. A value of -1 is returned on failure.

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

```

```c
int main(int argc, char **argv) {
        switch (fork()) {
        case 0:
                printf("I am the child: pid=%d\n", getpid());
                break;
        default:
                printf("I am the parent: pid=%d\n", getpid());
                break;
        case -1:
                perror("Fork failed");
        }
        return 0;
}
```

2.5. Basics of Multi-process programming

Multi-process programming Implement a program using `fork()` command to create child process. Student can check the number of forked processes using `ps` command.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>      /* defines fork(), and pid_t. */

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

  pid_t child_pid;

  /* lets fork off a child process... */
  child_pid = fork();

  /* check what the fork() call actually did */
  if (child_pid == -1) {
        perror("fork"); /* print a system-defined error message */
        exit(1);
  }

  if (child_pid == 0) {
    /* fork() succeeded, we're inside the child process */
    printf("Hello, ");
    fflush(stdout);
  }
  else {
    /* fork() succeeded, we're inside the parent process */
    printf("World!\n");
    fflush(stdout);
  }

  return 0;
}
```

Compile and run the program observing the output of the program and giving the conclusion about the order of letters “Hello, World!”.

This is because of the independent among porcesses, so that the order of execution is not guaranteed. It need a mechanisim to suspend the process until its child process finised. The system call `wait()` is used in this case. The example of `wait()` is presented in appendix B.
In case of expanding the result to implement the reverse order which child process is suspended until its father process is finished. This case is not recommended due to the `orphan process` status. An example of such programs is presented in appendix **??**

2.6. Retrieve the code segment of process

2.6.1. The information of process

You need a program with the execution time being enough long.

```
/* Source code of loop_process.c */
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  int timestamp=0;
  while(1){
      printf("Time: %5d\n", timestamp++);
      sleep(1);
  }
  return 0;
}
```

```
$ gcc -o loop_process loop_process.c

$ ./loop_process
Time:     0
Time:     1
Time:     2
Time:     3
Time:     4
```

```
...
```

Find process ID `ps` command is used to find (process ID - pid) of a process.

```
$ ps -a | grep loop_process
 7277 tc       ./loop_process
 7279 tc       grep loop_process
```

Retrieve PCB information of process on Linux system, each running process is reflected in the folder of filesystem `/proc`. Information of each process is associated with the folder called /proc/<pid>, where pid is process ID. For example, with the *pid* of the process above, `./loop_process` the directory containing the information of this process is `/proc/7277`.

```
$ ls /proc/<pid>
autogroup         environ       mountstats      smaps
auxv              exe           net/            stat
cgroup            fd/           ns/             statm
clear_refs        fdinfo/       oom_adj         status
cmdline           limits        oom_score       syscall
comm              maps          oom_score_adj   task/
coredump_filter   mem           pagemap
cpuset            mountinfo     personality
cwd               mounts        root
```

Using commands such as `cat`, `vi` to show the information of processes.

```
$ cat /proc/<pid>/cmdline
./loop_process

$ cat /proc/<pid>/status
Name:   loop_process
State:  S (sleeping)
Tgid:   7277
Pid:    7277
PPid:   6760
TracerPid:      0
Uid:    1001    1001    1001    1001
Gid:    50      50      50      50
FDSize: 32
...
```

2.6.2. Compare the code segment of process and program

/proc/<pid>/ stores the information of code in `maps`

```
$ cat /proc/<pid>/maps
08048000-08049000 r-xp 00000000 00:01 30895 /home/.../loop_process
08049000-0804a000 rwxp 00000000 00:01 30895 /home/.../loop_process
b75e0000-b75e1000 rwxp 00000000 00:00 0
b75e1000-b76f8000 r-xp 00000000 00:01 646   /lib/libc-2.17.so
b76f8000-b76fa000 r-xp 00116000 00:01 646   /lib/libc-2.17.so
b76fa000-b76fb000 rwxp 00118000 00:01 646   /lib/libc-2.17.so
b76fb000-b76fe000 rwxp 00000000 00:00 0
b7705000-b7707000 rwxp 00000000 00:00 0
b7707000-b7708000 r-xp 00000000 00:00 0     [vdso]
b7708000-b7720000 r-xp 00000000 00:01 648   /lib/ld-2.17.so
b7720000-b7721000 r-xp 00017000 00:01 648   /lib/ld-2.17.so
b7721000-b7722000 rwxp 00018000 00:01 648   /lib/ld-2.17.so
bf9a8000-bf9c9000 rw-p 00000000 00:00 0     [stack]
```

Comparing with the program (executable binary file) using *ldd* to read executable binary file and *readelf* to list libraries that are used.

```
$ ldd loop_process
        linux-gate.so.1 (0xb77dc000)
        libc.so.6 => /lib/libc.so.6 (0xb76b6000)
        /lib/ld-linux.so.2 (0xb77dd000)

$ readelf -Ws /lib/libc.so.6 | grep sleep
 388: 000857f0 105 FUNC WEAK   DEFAULT 11 nanosleep@@GLIBC_2.0
 660: 000857f0 105 FUNC WEAK   DEFAULT 11 __nanosleep@@GLIBC_2.2.6
 803: 000bda48 121 FUNC GLOBAL DEFAULT 11 clock_nanosleep@@GLIBC_2.17
1577: 000bda48 121 FUNC GLOBAL DEFAULT 11 __clock_nanosleep@@GLIBC_PRIVATE
1651: 000aa8f8  43 FUNC GLOBAL DEFAULT 11 usleep@@GLIBC_2.0
1959: 00085564 542 FUNC WEAK   DEFAULT 11 sleep@@GLIBC_2.0
```

Following that, we can see the consistency of code segment between program and process.

References

- Coding style by GNU: `http://www.gnu.org/prep/standards/standards.html`.
- C programming
  - Brian Kernighan, and Dennis Ritchie, *"The C Programming Language"*, Second Edition
  - Randal E. Bryant and David R. O'Hallaron, *"Computer systems: A Programmer's Perspective"*, Second Edition
- More information about Vim: `http://vim.wikia.com/wiki/Vim_Tips_Wiki`
- Makefile:
  - A simple Makefile tutorial `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`
  - GNU Make Manual `https://www.gnu.org/software/make/manual/make.html`

3. Exercises

NOTE: For submitting exercises solution to BKEL, the following tasks are requested:

- Create a folder for each part in $3.Exercises, e.g folders of Questions,Basic and Programming.
- Create a sub-folder of Programming for each Problem in the section $3.3 Programming exercise, make a makefile for each problem.
- Compress these folders into a ZIP file entitled as Lab1_<StudentID>.zip (RAR file will be rejected).

3.1. Questions

1. What are the advantages of Makefile? Give examples?
2. In case of source code files located in different places, how can we write a Makefile?
3. What the output will be at LINE A? Explain your answer.

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

int value = 5;

int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) {       /* child process */
          value += 15;
          return 0;
    }
    else if (pid > 0) { /* parent process */
          wait(NULL);
          printf("PARENT: value = %d",value); /* LINE A */
          return 0;
    }
}
```

3.2. Basic commands

1. Create a new directory entitled with your student ID.
2. Create a new file *example.txt* under folder <StudentID> with the following content:

//Your student ID

To start vim editor, run the command: vim

The editor is now in command mode. To start editing the file content, enter: :i[enter]

To save: :w

To save and exit: :wq

To exit: :q

3. List the content of <student-ID> directory
4. View the content of file *example.txt*.
5. Show first 5 lines from example.txt.
6. Show last 5 lines from example.txt.
7. List all commands that have been run from that terminal session and save to *<StudentID>-history.txt*.

Students save "<StudentID>-history.txt" and "example.txt" in a folder.

3.3. Programming exercises

Two header files named factorial.h and readline.h have the following contents:

```c
// factorial.h
#ifndef FACTORIAL_H
#define FACTORIAL_H

int factorial(const int aNumber);

#endif
```

```c
// readline.h
#ifndef READ_LINE_H
#define READ_LINE_H

int read_line(char *str);

#endif
```

Problem 1. Write factorial.c to implement function factorial(): the function get an integer and return its factorial. For example factorial(6) returns 720.

PROBLEM 2. Write readline.c to implement read_line(): read_line() gets data from stdin (keyboard), line-by-line. The content from stdin will be recorded on the parameter of this function named str. The result of read_line() indicates that whether the line is an integer or not. For example, with the input string below:

```
Hello, world
Operating system
Computer Science and Engineering
123
```

After calling the function, read_line() writes "Hello,world" into str and returns 0. The second calling will write "Operating system" into str and return 0. The third calling will write "Computer Science and Engineering" into str and return 0. The last call will write "123" into str and return 1.

PROBLEM 3. Write main.c to create an executable file named **myfactorial** that reads input from stdin line by line and compute factorial if the line is an integer (each line does not exceed 50 letters). Then print factorial if the line is an integer else print -1. Write a Makefile to compile the program at least two targets:

- all: create **myfactorial** from other files.
- clean: remove all of object files, binaries.

```c
// main.c
#include <stdio.h>
#include "readline.h"
#include "factorial.h"

int main(int argc, char * argv[]) {
        // Implement myfactorial
}

#endif
```

PROBLEM 4 Given a file named "*numbers.txt*" containing multiple lines of text. Each line is a non-negative integer. Write a C program that reads integers listed in this file and stores them in an array (or linked list). The program then uses the $fork()$ system call to create a child process. The parent process will count the numbers of integers in the array that are divisible by 2. The child process will count numbers divisible by 3. Both processes then send their results to the stdout. For examples, if the file "*numbers.txt*" contains the following lines

```
12
3
```

```
4
10
11
```

After executing the program, we must see the following lines on the screen (in any order)

```
3
2
```

PROBLEM 5 A practice with multi-processing programming:

- Firstly, downloading two text files named "movie-100k-split.zip" and extract it. These file contains the 100.000 ratings of 943 users for 1.682 movies in the following format:

```
userID <tab> movieID <tab> rating <tab> timeStamp
userID <tab> movieID <tab> rating <tab> timeStamp
...
```

- Secondly, you should write a program that spawns two child processes, and each of them will read a file and compute the average ratings of movies in the file and write the answer into a result file. So you will have two result files for two input text files.

- Finally, the parent process will wait until its children finish their tasks and it will read the result files to compute the final results and write them to a text file named "finaloutput.txt".

A. Makefile example

```
FC=gfortran
CC=gcc
CP=cp

.PHONY: all clean

OBJS = mylib.o mylib_c.o

# Compiler flags
FFLAGS = -g -traceback -heap-arrays 10 \
          -I. -L/usr/lib64 -lGL -lGLU -lX11 -lXext

CFLAGS = -g -traceback -heap-arrays 10 \
          -I. -lGL -lGLU -lX11 -lXext

MAKEFLAGS = -W -w

PRJ_BINS=hello
PRJ_OBJS = $(addsuffix .o,$(PRJ_BINS))

objects := $(PRJ_OBJS) $(OBJS)

all: myapp

%.o: %.f90
         $(FC) -D_MACHTYPE_LINUX $< -c -o $@

%.o: %.F
         $(FC) -D_MACHTYPE_LINUX $< -c -o $@

%.o: %.c
         $(CC) -D_MACHTYPE_LINUX  $< -c -o $@

myapp: objects
         $(CC) $(CFLAGS) $^ $(objects) -o $@

clean:
         @echo "Cleaning up.."
         rm -f *.o
         rm -f $(PRJ_BINS)
```

B. System Call wait()

This example uses system call to guarantee the order of running processes.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>      /* defines fork(), and pid_t. */

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

  pid_t child_pid;

  /* lets fork off a child process... */
  child_pid = fork();

  /* check what the fork() call actually did */
  if (child_pid == -1) {
        perror("fork");
        exit(1);
  }

  if (child_pid == 0) {
    /* fork() succeeded, we're inside the child process */
    printf("Hello, ");
    fflush(stdout);
  }
  else {
    /* fork() succeeded, we're inside the parent process */
    wait(NULL);          /* wait the child
exit */
    printf("World!\n");
    fflush(stdout);
  }

  return 0;
}
```