Operating Systems [2021-2022]

Support doc – C and makefiles

Introduction

The practical assignments of the Operating Systems course must be developed in *C*, a well-known language that is widely used for operating systems programming. *C* provides a straightforward access to system level functions (system calls), allows the manipulation of data at bit level (low level programming), and the implicit usage of pointers to the system's memory. *C* also provides an unmanaged access to the system resources, thus being able to produce highly efficient programs.

This support document is not a C manual! At this time the student already had programming classes that used C. The intention of this document is to highlight some details about C programs that are especially useful to the course of Operating Systems, namely the compilation process, useful compilation flags, the use of `make` and `makefiles` to automate the building process, and specifics of file usage.

Basic C program

As you know, the sources of a C program consist of one or more `.c` files that should have exactly one `main` method. The parameters of the main (`int argc, char **argv`) may be omitted, but they are used to pass arguments to your program upon execution.

prog1.c

```
#include <stdio.h>
#include "prog1.h"

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

   printf("Hello World! ");

   return 0;
}
```

Program compilation

GNU C Compiler will be used for C compilation.

Usage:

```
gcc -Wall file1.c file2.c (...) -o executable
```

Note: Use compile option **–Wall** to enable most of the warnings. Solve every error and warning that is shown by the compiler. Do not ignore warnings! Normally they result from poor and error-prone coding!

Program execution

After the compilation, the program should be in the binary form ready to be executed.

Usage:

```
./executable [argument1] [argument2]
```

Note: when using the `argv` parameter of the main, the `argv[0]` is the name of the program.

Compilation Flags

Often you need to provide the compiler with additional information that it should use during the compilation process. This additional information is provided through compilation flags, which are preceded by the minus signal (' - ') and are included in the compilation command.
The example above uses two of the most frequently used flags: **`-Wall`**, which enables most warnings, and **`-o`** which allows defining the name of the output program. Following, we introduce examples of other flags that you will need to use during the course, and there are several other flags that you should explore when necessary.

Compiling with debug information

In order to debug a C program with **gdb**, the **–g** option must be used during compilation. This option generates debugging information that will be included in the final executable.

E.g.:

```
gcc -Wall -g prog1.c -o prog1
```

Note: Additional information about debugging can be found in the *GNU Debugger tutorial*.

Compiling programs that use threads

In the programs that make use of POSIX threads, it is necessary to compile your program using special flags to the preprocessor and linker. The recommended way to do this is by using `-pthread`, that enables everything necessary with higher portability. In the Linux system used in the course, it corresponds to the use of the following flags: `-lpthread -D_REENTRANT`.

E.g.

```
gcc -Wall -g -pthread prog1.c -o prog1
```

Additional details are provided together with the assignments that involve threads.

Security and errors verification

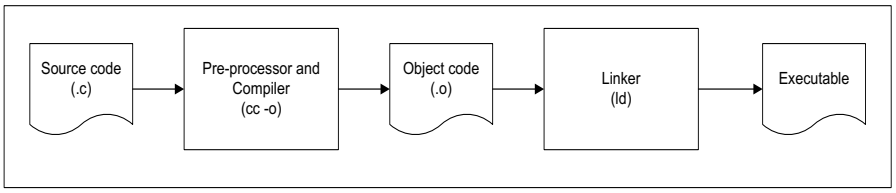
To statically check C programs for security vulnerabilities and programming mistakes, **splint** can be used.

Usage:

```
splint [options] {files}
```

Make and Makefile

Programs are made from separate source files, which correspond to different software modules of the overall program. Each of these files is then compiled into object code, by the compiler, and then linked in one executable, by the linker.

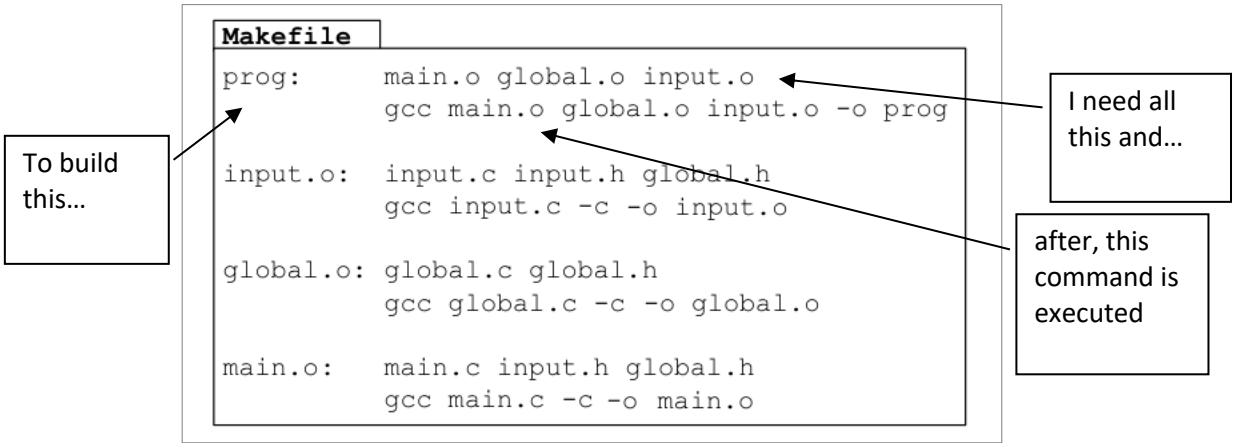


The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. Special files (*makefiles*) describe the relationships among files in the program, and state the commands for updating each file. Only needed files (the ones that were modified and their dependencies) are compiled each time.

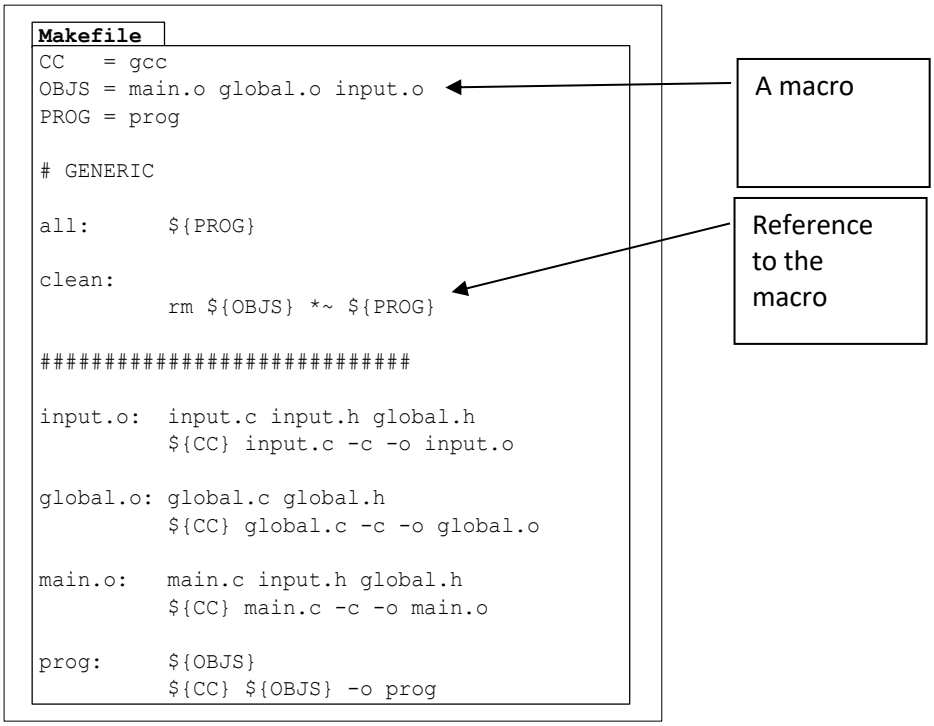
Syntax: `make [-f makefile] [target ...]`

If a *makefile* is not specified, make expects a file called "Makefile" or "makefile" to be used. If a target is not specified, the first in the file is chosen.

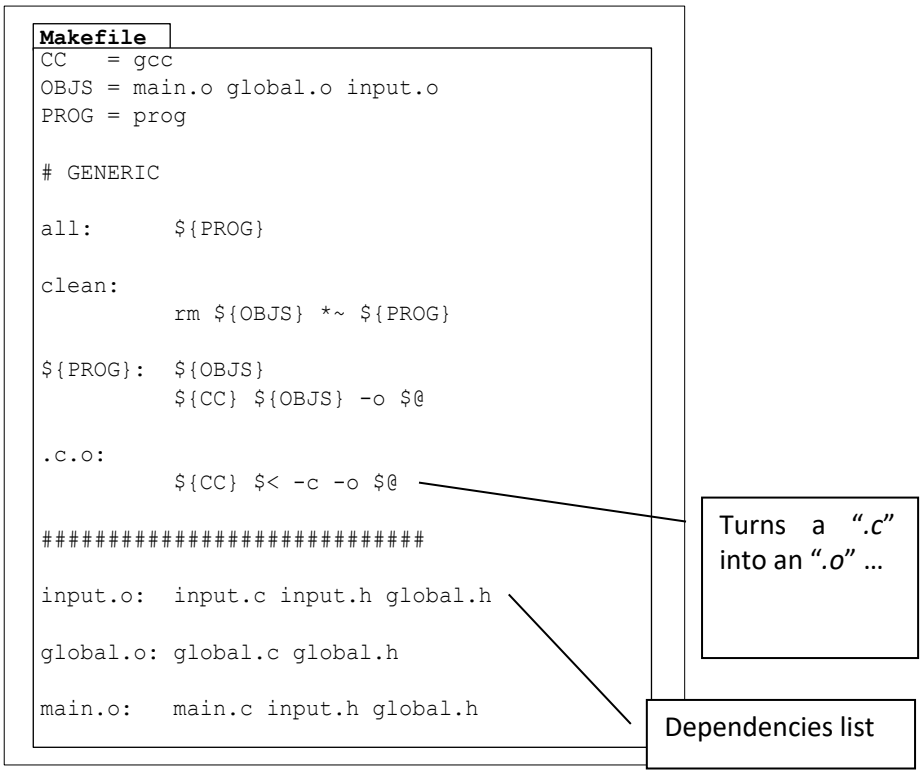
Note:
To compile use the command: `gcc -Wall {source_files} -o {executable file}`
To compile without linking use: `gcc -Wall {source_file} -c [-o {object file}]`
If `-o` is not specified `{source_file}.o` is assumed



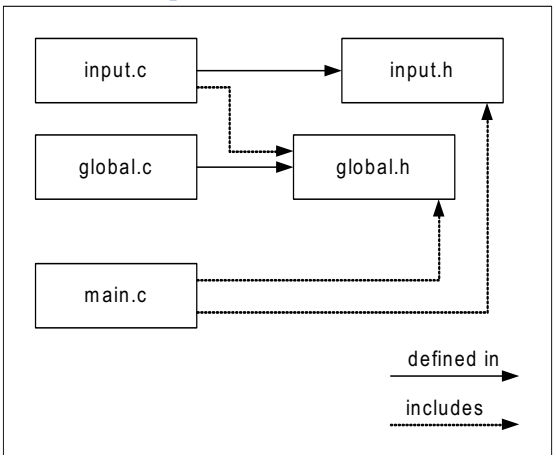
- Put a **TAB** between targets (e.g.: "**prog:**") and the files on which they depend (e.g.: "**main.o global.o input.o**").
- Put a **TAB** before the commands to execute.
- Leave a **blank line** between targets.



- Special macros (see manual):
  - ***$@*** *Refers to the current target name*
  - ***$<*** *Represents the name of the component that is being used to make the target*



Make Example



input.c
```
#include "global.h"
#include "input.h"

void menu() {
     ...
     error = NO_SCREEN;
}
```

input.h
```
#ifndef _INPUT_H
#define _INPUT_H

extern void menu();

#endif
```

global.c
```
#include "global.h"

int error = NO_ERROR;
```

global.h
```
#ifndef _GLOBAL_H
#define _GLOBAL_H

#define NO_ERROR    0
#define NO_SCREEN   1

extern int error;

#endif
```

main.c
```
#include "global.h"
#include "input.h"

void main() {
     menu();
}
```

```
$gcc input.c –c –o input.o
$gcc global.c –c –o global.o
$gcc main.c –c –o main.o
$gcc input.c global.c main.c -o prog
```

- Each module is defined in a *.c* file.
- Each function that is to be accessed externally has its interface in the corresponding *.h* file.
- Global variables are declared in *.c* files. Their existence is declared in *.h* files by using "extern".
- By using *#ifndef* / *#define* , duplication of definitions is avoided during compilation.

Makefile
```
CC    = gcc
OBJS = main.o global.o input.o
PROG = prog

# GENERIC

all:       ${PROG}

clean:
           rm ${OBJS} *~ ${PROG}

${PROG}:   ${OBJS}
           ${CC} ${OBJS} -o $@

.c.o:
           ${CC} $< -c -o $@

############################

input.o:   input.c input.h global.h

global.o: global.c global.h

main.o:    main.c input.h global.h
```

- You only have to change the shadow parts to compile different projects, or to update your files during the same project!
- The project becomes up-to-date on every call to Make.

Exercise

Consider a program that is implemented in the following source files:

```
main.c   main.h  semlib.c semlib.h drone.c  drone.h
```

the compiled executable should be named `program` and uses all the sources in the list of files. Furthermore, the `main.c` uses functions of both `semlib.c` and `drone.c`. You should use the flags `-Wall` for everything and `-pthread` is required by `main.c`.

1) Create a basic Makefile that compiles the sources to the executable `program`, without taking advantage of macros.

Solution:

```
program:    semlib.o drone.o main.o
            gcc -Wall -pthread semlib.o drone.o main.o -o program

semlib:     semlib.h semlib.c
            gcc -Wall semlib.c -c -o semlib.o

drone:      drone.h drone.c
            gcc -Wall drone.c -c -o drone.o

main:       semlib.h drone.h main.h main.c
            gcc -Wall -pthread main.c -c -o main.o
```

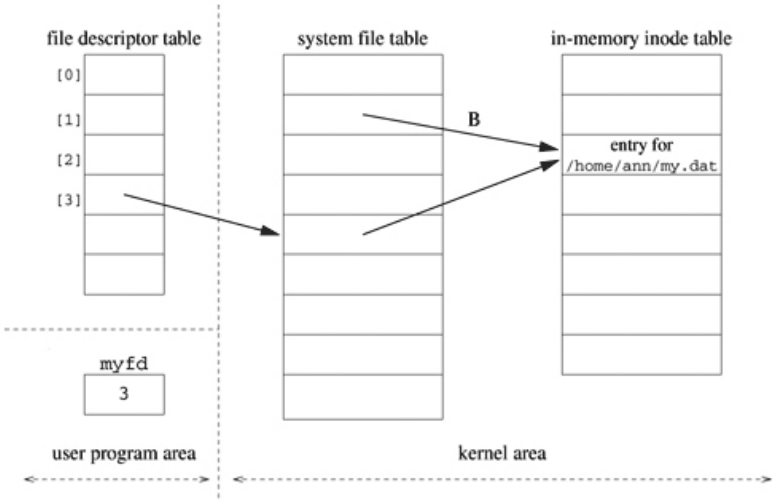
2) Looking at the solution presented above, it is easy to understand that the makefile is not very maintainable. Thus, improve your makefile to take full advantage of macros and special macros.

Solution:

```
CC      = gcc
FLAGS   = -Wall
PROG    = program
OBJS    = semlib.o drone.o main.o

all:        ${PROG}

clean:
            rm ${OBJS} *~ ${PROG}

${PROG}:    ${OBJS}
            ${CC} ${FLAGS} ${OBJS} -lm -o $@

.c.o:
            ${CC} ${FLAGS} $< -c -o $@

##########################

semlib.o:   semlib.h semlib.c

drone.o:    drone.h drone.c

main.o:     semlib.h drone.h main.h main.c

program:    semlib.o drone.o main.o
```

Files in C and Unix

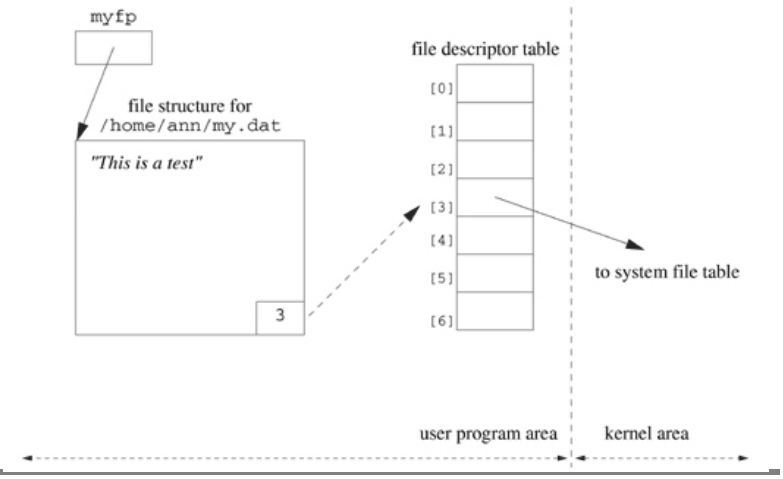
Files are designated within C programs either by file pointers or by file descriptors. The standard I/O library functions for ISO C such as `fopen`, `fscanf`, `fprintf`, `fread`, `fwrite` and `fclose`, use file pointers. The UNIX I/O functions like `open`, `read`, `write`, `close` and `ioctl` use file descriptors. File pointers and file descriptors provide logical designations called handles for performing device-independent input and output.



A *file descriptor* is an index to a table of descriptors that exists in the process user area. The program accesses it through functions that use the file descriptor which takes the form of an integer value (e.g. `int fp = open("myfile.c", O_RDONLY);`). The `open` function creates an entry in the file descriptor table which points to an entry in the *system file table*. The *system file table*, which is shared by all the processes in the system, has an entry for each open file together with its offset (position of read or write), an indication of the access mode (i.e., read, write or read-write) and a count of the number of file descriptor table entries pointing to it. Each element in the table references an entry in the *in-memory inode table*.

Several *system file table* entries may correspond to the same entry in *the in-memory inode table*, as different processes may be reading the same file. In this situation, each process has its own offset into the file and reads the entire file independently of the other process. The writes are also independent of each other. Each user can write over what the other user has written because of the separate file offsets for each process.

A *file pointer* points to a data structure called a FILE structure in the user area of the process. The FILE structure contains a buffer and a file descriptor value. The file descriptor value is the index of the entry in the *file descriptor table* that is actually used to output the file to disk. In some sense the file pointer is a handle to a handle.



When the program calls `fprintf(myfp, "This is a test");`, the message "This is a test" is not actually written to disk, but instead written to a buffer in the FILE structure. When the buffer fills, the I/O subsystem calls `write()` with the file descriptor, and only then the data is written

in the disk. To avoid losing data if a crash occurs before the writing, or just to avoid the effects of buffering, `fflush()` may be used. Calling this function forces whatever has been buffered in the to be written to the disk, even if the buffer is not yet full.

The following example uses the standard C library (stdio) to write/read to/from files.

```
#include <stdio.h>

int write_to_file()
{
   int account;     /* account number */
   char name[ 30 ]; /* account name */
   double balance;  /* account balance */

   FILE *cfPtr;     /* cfPtr = clients.dat file pointer */

   /* fopen opens file. Exit program if unable to create file  */
   if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
      printf( "File could not be opened\n" );
   } /* end if */
   else {
      printf( "Enter the account, name, and balance.\n" );
      printf( "Enter EOF to end input.\n" );
      printf( "? " );
      scanf( "%d%s%lf", &account, name, &balance );
      /* write account, name and balance into file with fprintf */
      while ( !feof( stdin ) ) {
         fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
         printf( "? " );
         scanf( "%d%s%lf", &account, name, &balance );
      } /* end while */
      fclose( cfPtr ); /* fclose closes file */
   } /* end else */
   return 0; /* indicates successful termination */
}

int read_from_file()
{
   int account;     /* account number */
   char name[ 30 ]; /* account name */
   double balance;  /* account balance */

   FILE *cfPtr;     /* cfPtr = clients.dat file pointer */

   /* fopen opens file; exits program if file cannot be opened */
   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
      printf( "File could not be opened\n" );
   } /* end if */
   else { /* read account, name and balance from file */
      printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
      fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
      /* while not end of file */
      while ( !feof( cfPtr ) ) {
         printf( "%-10d%-13s%7.2f\n", account, name, balance );
         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
      } /* end while */
      fclose( cfPtr ); /* fclose closes the file */
   } /* end else */
   return 0; /* indicates successful termination */
}
```