Make

Make este un tool esential de folosit in proiecte mari prin care se pot determina relatiile de dependenta intre fisiere.

Este estential de folosit in proiectele mari pentru ca daca intr-un proiect utilizam mai multe fisiere sursa si unul din ele este modificat, atunci tot procesul de compilare trebuie reluat. Dar in general nu e necesar ca acest proces sa fie reluat in intregime, fiind necesara recompilarea doar a surselor modificate.

Sintaxa este: **make program**.

Prin aceasta sintaxa spunem interpretorului de comenzi ca dorim executarea operatiilor de determinare a dependintelor intre fisiere si dorim compilarea fisierelor necesare pentru crearea executabilului.

“Program” reprezinta ceea ce in continuare vom numi tinta. Fiecare tinta putand avea la randul lor tinte pentru alte dependinte.

Caracteristica esentiala a tool-ului make este ca la fiecare apelare se vor evalua in lant toate dependintele si se vor prelucra sau compila doar cele pentru care s-au constatat modificari.

Insa pentru a putea utiliza tool-ul make avem nevoie sa precizam dependitele prin intermediul unui fisier numit makefile.

Printre fisierele din arhiva, veti gasi:

- un fisier numit **dependency.c** care contine:

```c
#include<stdio.h>

void sayHello()
{
        printf("Hello Wooooorld!\n");
}
```

- un fisier numit **dependency.h** care contine:

```c
void sayHello(void);
```

- un fisier numit **main.c** care contine:

```c
#include "dependency.h"

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

- un fisier numit **makefile** care contine:

```
runme: main.o dependency.o
        gcc -o runme dependency.o main.o

main.o: main.c
        gcc -c main.c

dependency.o: dependency.c
        gcc -c dependency.c
```

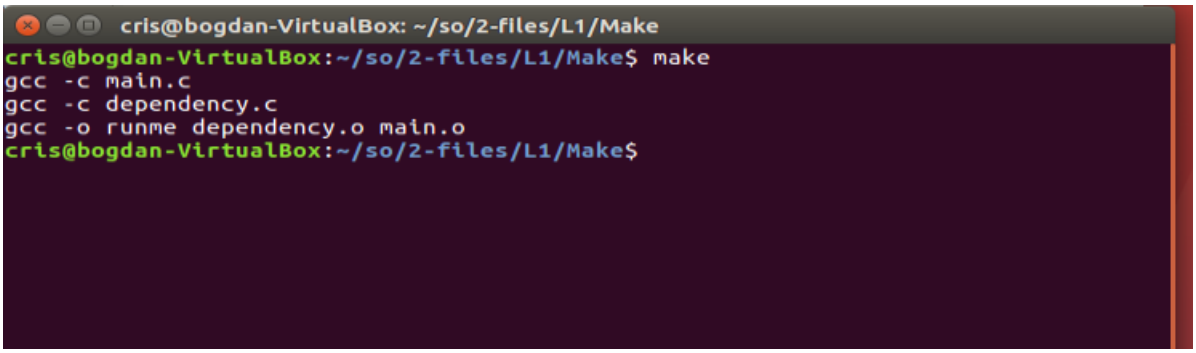
Prima linie, din fiecare set, reprezinta linia de dependinte iar cea de-a doua linie reprezinta linia de comenzi.

Runme, main.o, dependency.o reprezinta tinte.

Liniile de comenzi trebuie sa inceapa cu tab.

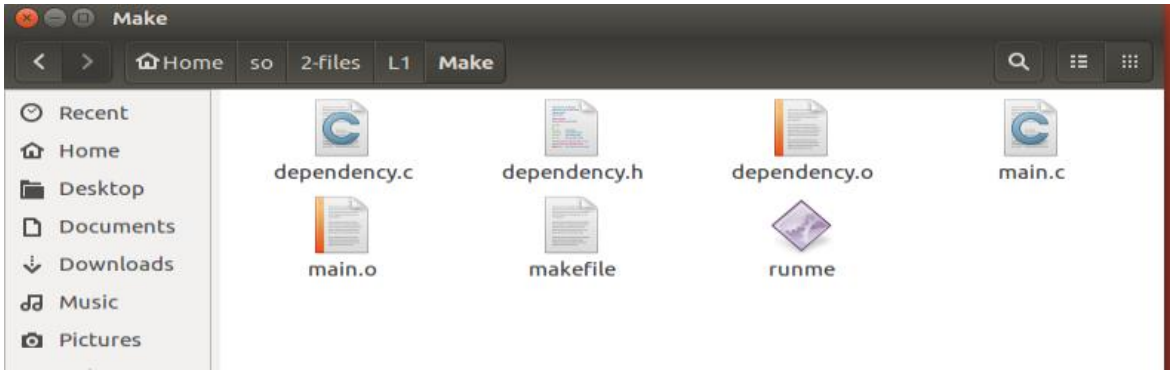
Comentariile sunt delimitate ca si in cazul shell script-urilor de # si se continua pana la sfarsitul liniei.

Daca intr-un folder aveti 1 singur fisier makefile, in terminal se apeleaza simplu comanda make (sau bineinteles se poate si make runme)



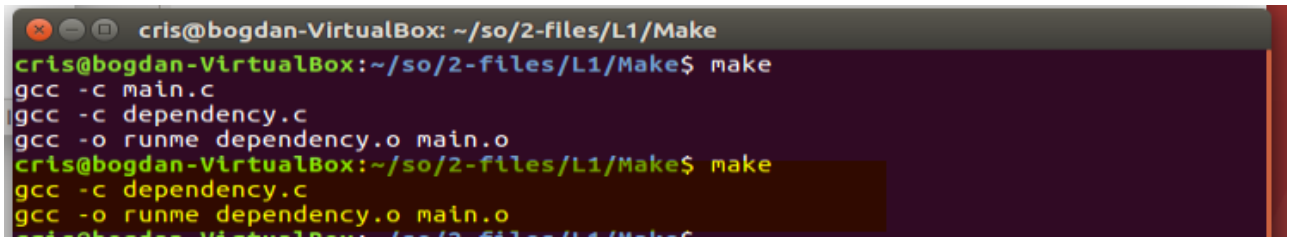
Daca doriti sa aveti mai multe fisiere makefile intr-un director, le numiti cum doriti iar apoi in terminal le veti apela cu make -f a123.

Make parcurge secventa de comenzi, deci pentru **runme** avem nevoie de **main.o && dependency.o**. Asadar tool-ul make, prima data compileaza **main.c** pentru a obtine **main.o** (output file) si mai apoi in mod similar pentru **dependency.c**.



Daca veti modifica fisierul **dependency.c**

o sa vedeti ca la urmatorul apel make, doar 2 linii din 3 vor fi recompilate pentru ca noi nu am adus modificari fisierului main.c.



Nu in ultimul rand putem avea tinte fara dependinte. De exemplu multe fisiere de descriere contin tinta clean destinata eliminarii fisierelor temporare obtinute in urma procesului de compilare:

```
clean :
        /bin/rm –f  *.o
```

Tool-ul make este indicat de folosit in proiectele mari, proiecte care au multe fisiere, deci mai multe dependinte astfel in fisierele de descriere se poate observa repetarea unui anumit numar de elemente.

Pentru simplificarea fisierelor de descriere si pentru a reduce dimensiunea fisierelor, se pot utiliza macrouri care au urmatoarea sintaxa: **nume=sir de caractere**.

Referirea unui macro se face prin **$(nume)** sau **${nume}** sau fara paranteze, daca numele este alcatuit dintr-un caracter.

Make preia linia de definitie macro (care contine =) si asociaza numele din stanga semnului = cu secventa de caractere ce urmeaza in dreapta. Si pentru a face distinctie intre o linie de definitie si una de comenzi, la linia de definitie nu este permis tab-ul.

Ca o alta regula de sintaxa, caracterul \ indica faptul ca linia curenta se continua pe urmatoarea linie (make inlocuieste \ cu spatiu)

Prin conventie macro-urile se scriu cu caractere mari. Deasemenea macrourile se pot utiliza in noi definitii. Si orice definitie de macro trebuie sa fie facuta inainte de linia de dependinte in care apare.

Fisierul **fancymakefile** a fost obtinut prin rescrierea fisierului **makefile** cu macro-uri si contine:

```
CE=c
TARGET = main.o dependency.o

runme: $(TARGET)
        gcc -o runme dependency.o main.o
main.o: main.$(CE)
        gcc -c main.$(CE)
dependency.o: dependency.$(CE)
        gcc -c dependency.$(CE)
```

Un alt mod de definitie de macro ar fi din **linia de comanda:** `make refext DIR=/usr/proj`

Sau pentru ca variabilele interpretorului fac parte din **mediul de lucru**:

```
DIR = /home/student/proiect
export DIR
make  refext
```

Makefile:

```
SRC = ${DIR}/src
refext      :
      cd ${DIR}/src
```

Deoarece un macro poate fi definit in mai multe moduri pot aparea conflicte intre definitii provenite din diverse surse. In ordinea prioritatilor avem:

- definitii introduse din linia de comanda
- definitii din fisierul de descriere
- variabile din mediu interpretorului de comenzi
- definitii interne

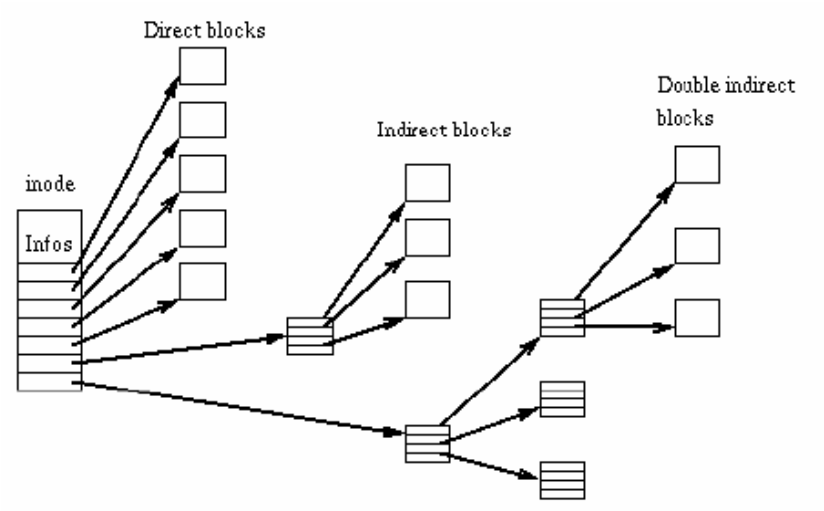
Fisiere – Make & I/O

Fisierele in Linux sunt organizate intr-o structura ierarhica. Fiecare fisier este reprezentat de o structura numita **inode.**

Un inode contine informatii de descriere precum: tipul fisierului, drepturile de acces, proprietarul, dimensiune, data crearii, referinte catre blocurile de date.

Lista de blocuri de pe disc care contine fisierul (adica blocurile de date) se realizeaza printr-un tablou cu 13 intrari. Primele 10 intrari contin direct adresele de bloc pentru primele 10 blocuri ale fisierului. A 11-a coloana e adresa unui bloc rezervat fisierului a carui continut e interpretat ca liste de adrese de blocuri(indirectare simpla)

Intrarea a 12-a contine un bloc al carui continut consta in adrese de blocuri de date (indirectare dubla). Analog intrarea a 13-a produce indirectare tripla.



Accesul la fisiere

Fisierele sunt accesate prin intermediul unor descriptori de fisiere.

Prin conventie descriptorii 0,1,2 au semnificatie speciala:

0 - intrare standard

1 - iesire standard

2 - iesire standard de eroare

Exista 5 apeluri care genereaza descriptori de fisiere: **creat, open, fcntl**, **dup** si **pipe**.

Pe parcursul laboratorului veti intalnii urmatoarele librari:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>

#define BUF_SIZE 8192
```

Functia OPEN

- are nevoie de urmatoarele librari:

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

- Si are semnatura:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Functia open returneaza descriptorul de fisier sau -1 in caz de eroare.

Cel de-al 3-lea argument este folosit doar la crearea de fisiere noi. Apelul cu 2 argumente se foloseste pentru citirea si scrierea fisierelor existente.

Functia open returneaza cel mai mic descriptor de fisier disponibil.

Argumentul **flangs** se formeaza printr-un SAU pe biti intre constantele:

O_RDONLY - deschide fisierul pentru citire
O_WRONLY - deschide fisierul pentru scriere
O_RDWR - deschide fisierul pentru citire si scriere
Argumente definite in antet fcntl.h

Alte constance ce pot fi folosite pentru argumentul flags, insa sunt optionale sunt:

O_APPEND - scriere la sfarsitul fisierului
O_CREAT - daca un fisier nu exista, il creaza
O_TRUNC - daca un fisier exista va sterge continutul

Argumentul al 3-lea, **mod**, poate fi o combinatie de SAU pe biti intre urmatoarele constante simbolice:

S_IRUSR, S_IWUSR, S_IXUSR User: read, write, execute
S_IRGRP, S_IWGRP, S_IXGRP Group: read, write, execute
S_IROTH, S_IWOTH, S_IXOTH Other: read, write, execute
Sau sub forma octala pot fi :

```
U  G  O
   0  6  4  4
      |
   0 1 1 0
     R W X
```

```c
void main()
{
        int fileDes;
        fileDes = open("sursa.txt", O_RDONLY);
        close(fileDes);
}
```

Functia CREAT are urmatoarea semnatura si are nevoie de aceasi librari ca si open:

```
int creat( const char *path, mode_t mod);
```

Asemenea functiei open, functia creat returneaza descriptorul de fisier sau -1 in caz de eroare. Apelul acestei functii este echivalent cu:

```
open( path, O_WRONLY | O_CREAT | O_TRUNC, mod);
```

Path, respectiv **mod** ca si in cazul functiei open, reprezinta numele fisierului si drepturile de acces.

Observatie: referitoare la argumentul O_TRUNC care implica stergerea continutului deja existent din fisier. Daca fisierul exista, argumentul al 2-lea, mod, este ignorat deoarece in acest caz, nu se modifica drepturile de acces.

Functia READ are urmatoarea semnatura:

#include <unistd.h>

*ssize_t read(int fd, void *buf, size_t noct);*

Functia Read este utilizata pentru a citi un numar de octeti dintr-un fisier de la pozitia curenta. Functia Read returneaza **numarul de octeti cititi** efectiv, 0 in caz de **EOF** (end of file) sau **-1** in caz de eroare.

Se citesc **noct** octeti din fisierul deschis referit de **fd** (file descriptor) si ii depune in tamponul referit de catre **buf**

```
void main()
{
        int f;
        int cititi=0;
        char bufArray[10];

        f = open("sursa.txt", O_RDONLY);
        while(cititi = read(f, &bufArray, 4))
        {
                bufArray[cititi]='\0';
                printf("%s", bufArray);
        }
        close(f);
}
```

Functia WRITE are urmatoarea semnatura:

#include <unistd.h>

*ssize_t write(int fd, const void *buf, size_t noct);*

Functia write este utilizata pentru a scrie un numar de octeti intr-un fisier. Returneaza **numarul de octeti scrisi** sau **-1** in caz de eroare.

```
void main()
{
        int f;
        f = creat("destinatie.txt", S_IREAD|S_IWRITE);
        write(f, "alt random text", 12);
        close(f);
}
```

Rezultatul e ca in terminal se afiseaza : “alt random t” (doar 12 caractere).

Functia CLOSE are urmatoarea semnatura:

int close(int fd);

Functia close se utilizeaza pentru a disponibiliza descriptorul atasat unui fisier in vederea unei noi utilizari. Returneaza **0** in caz de succes sau **-1** in caz de eroare.

Functia LSEEK are urmatoarea semnatura:

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

off_t lseek( int fd, off_t offset, int interp);
```

Utilizam aceasta functie daca dorim sa pozitionam pointerul de fisier relativ sau absolut pentru un apel read sau write. Functia lseek returneaza un deplasament de fisier sau -1 in caz de eroare.

interp poate fi una din urmatoarele constante:

- **SEEK_SET** - pozitionare fata de inceput
- **SEEK_CUR** - pozitionare relativa fata de pozitia curenta
- **SEEK_END** - pozitionare fata de sfarsitul fisierului

```
void main()
{
        int f;
        int cititi=0;
        char bufArray[10];

        f = open("sursa.txt", O_RDONLY);
        lseek(f,5,SEEK_SET);
        if (read(f, &bufArray, 4)>0)
        {
                printf("Next 4 from the beginning + 5 offset: %s\n", bufArray);
        }
        lseek(f,3,SEEK_CUR);
        if (read(f, &bufArray, 4)>0)
        {
                printf("Next 4 after current position + 3 offset: %s\n", bufArray);
        }

        lseek(f,-12,SEEK_END);
        if (read(f, &bufArray, 4)>0)
        {
                printf("Next 4 after EOF -12 : %s\n", bufArray);
        }
        close(f);
}
```

```
Next 4 from the beginning + 5 offset: m te
Next 4 after current position + 3 offset: anot
Next 4 after EOF -12 : rand
```

```
random text
another random line
```