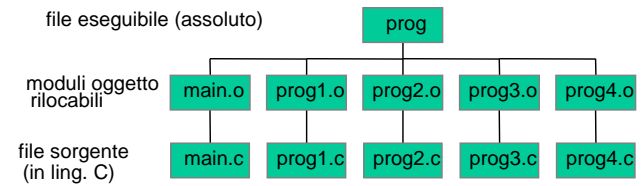


Laboratorio di sistemi operativi – I parte

Lo sviluppo del software

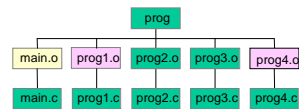
Un progetto



Un progetto (segue)

Supponiamo che:

- **main.o** non esista
- **prog1.o** e **prog4.o** siano antecedenti all'ultima versione di **prog1.c** e **prog4.c**
- **prog2.o** e **prog3.o** siano relativi all'ultima versione di **prog2.c** e **prog3.c**



```
>> cc -c main.c
>> cc -c prog1.c
>> cc -c prog4.c
>> cc -o prog main.o prog1.o prog2.o prog3.o prog4.o
```

Un progetto (segue)

Se i sorgenti sono frequentemente modificati

- controllare se qualche file sorgente è stato modificato
- ricompilare solo gli eventuali file modificati
- ricostruire l'eseguibile **prog**




operazione ripetitiva

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 5

Il comando make

Il comando **make** consente la manutenzione e l'aggiornamento di programmi



- controlla se i file sorgente sono stati ricompilati dopo l'ultima modifica
- compila i file modificati dopo i relativi file oggetto
- ricostruisce la versione aggiornata di prog

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 6

Makefile

Per svolgere le sue funzioni il comando **make** ha bisogno di un file ausiliario

Tale file contiene le regole per la manutenzione del software

Tale file e' di solito chiamato **Makefile**

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 7

Esempio di Makefile

```
# makefile per prog
prog: main.o prog1.o prog2.o prog3.o prog4.o
    cc -o prog main.o prog1.o prog2.o prog3.o prog4.o

main.o: main.c
    cc -c main.c

prog1.o: prog1.c
    cc -c prog1.c

prog2.o: prog2.c
    cc -c prog2.c

prog3.o: prog3.c
    cc -c prog3.c

prog4.o: prog4.c
    cc -c prog4.c
```

commento

regola per la gestione di prog2.o

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 8

makefile

target: cosa bisogna costruire

Da chi dipende target

```
prog: main.o prog1.o prog2.o prog3.o prog4.o
..... cc -o prog main.o prog1.o prog2.o prog3.o prog4.o
```

Come costruire target

Attenzione: TAB

makefile

Una **linea di dipendenza** definisce la **relazione** tra un **file** dipendente o target (a sx del carattere :) e uno o più **file di dipendenza**

Una **linea di comando** o regola, definisce le operazioni che **make** deve **effettuare** per passare da un **file di dipendenza** ad uno **dipendente**

Una linea di comando del **makefile** e' **eseguita** se i relativi **file** di dipendenza sono stati **modificati** più **recentemente** dei file dipendenti associati

make

Digitando **make** si ottiene l'esecuzione delle linee di comando relative al **primo file dipendente**

Se i **file di dipendenza** sono **superati** (oppure *non esistono*), vengono **eseguite** anche le linee di **comando necessarie** al loro aggiornamento (*loro creazione*)

esempio

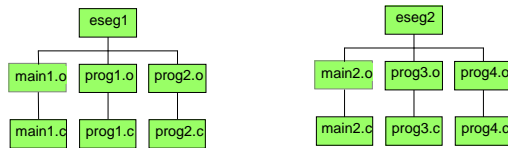
```
>> ls
main.c Makefile prog1.c prog2.c prog3.c prog4.c
>> make
cc -c main.c
cc -c prog1.c
cc -c prog2.c
cc -c prog3.c
cc -c prog4.c
cc -o prog main.o prog1.o prog2.o prog3.o prog4.o
>> ls
main.c main.o Makefile prog1.c prog1.o prog2.c
prog2.o prog3.c prog3.o prog4.c prog4.o
>>
```

esempio (cont.)

```
>> make
prog is up to date
>> rm prog2.o
>> make
cc -c prog2.c
cc -o prog main.o prog1.o prog2.o prog3.o prog4.o
>>
```

problema

2 applicazioni da mantenere



come deve essere fatto il Makefile?

possibile soluzione

```

# makefile per prog
eseg1: main1.o prog1.o prog2.o
    cc -o eseg1 main1.o prog1.o prog2.o
main1.o: main1.c
    cc -c main1.c
prog1.o: prog1.c
    cc -c prog1.c
prog2.o: prog2.c
    cc -c prog2.c

eseg2: main2.o prog3.o prog4.o
    cc -o eseg2 main2.o prog3.o prog4.o
main2.o: main2.c
    cc -c main2.c
prog3.o: prog3.c
    cc -c prog3.c
prog4.o: prog4.c
    cc -c prog4.c
  
```

Funziona?

NO

Digitando make si ottiene l'esecuzione delle linee di comando relative al **solo primo file dipendente** (prog1 in questo caso)



eseg2 non viene generato

soluzione

```

# makefile per prog
all: eseg1 eseg2
eseg1: main1.o prog1.o prog2.o
    cc -o eseg1 main1.o prog1.o prog2.o
main1.o: main1.c
    cc -c main1.c
prog1.o: prog1.c
    cc -c prog1.c
prog2.o: prog2.c
    cc -c prog2.c

eseg2: main2.o prog3.o prog4.o
    cc -o eseg2 main2.o prog3.o prog4.o
main2.o: main2.c
    cc -c main2.c
prog3.o: prog3.c
    cc -c prog3.c
prog4.o: prog4.c
    cc -c prog4.c
  
```

il target "all" dipende da eseg1 e eseg2

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 17

possibili usi del make

```
>> make -f makeprog
>> make
>> make prog
```

esegue le regole descritte nel file **makeprog** relative al primo target

esegue le regole in un file di nome **Makefile** (default) relative al primo target

esegue le regole relative al **solo target prog** nel file di nome **Makefile**

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 18

make

Non sempre e' necessario specificare tutte le dipendenze e le regole

Esempio:

```
# makefile di prog
prog: main.o prog1.o prog2.o prog3.o prog4.o
    cc -o prog main.o prog1.o prog2.o prog3.o prog4.o
main.o: main.c
prog1.o: prog1.c
prog2.o: prog2.c
prog3.o: prog3.c
prog4.o: prog4.c
```

Mancano le linee di comando

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 19

makefile

make e' in grado di riconoscere le relazioni esistenti tra alcuni file e di applicare ad essi regole predefinite

make riconosce file con il suffisso **.c** come sorgenti in linguaggio C e fa eseguire, se necessario, la loro **compilazione**

Marco Lapegna - Lab. di Sistemi Operativi I - lo sviluppo del software - 20

makefile parametrici

E' possibile rendere **parametrici** i makefile utilizzando delle variabili chiamate **macro**

```
# makefile di prog
OGGETTI= main.o prog1.o prog2.o prog3.o prog4.o
prog: $(OGGETTI)
    cc -o prog $(OGGETTI)
```

la macro viene definita
la macro viene utilizzata

esempio : di makefile

file makeprova

```
OGGETTI = qdrig.o pippo.o
OGGETTI1 = qdcol.o pluto.o
MAIN = driver.o
MAIN1 = driver_uno.o
#
rows:    $(OGGETTI) $(MAIN)
cc -o rows $(OGGETTI) $(MAIN)
#
columns: $(OGGETTI1) $(MAIN1)
cc -o columns $(OGGETTI1) $(MAIN1)
```

esempio

```
csh>make -f makeprova rows
cc -c qdrig.c
cc -c pippo.c
cc -c driver.c
cc -o rows driver.o qdrig.o pippo.o

csh>make -f makeprova rows
'rows' is up to date.
```

altre opzioni di make

```
csh>make -f makeprova -n
cc -c qdrig.c
cc -c pippo.c
cc -c driver.c
cc -o rows driver.o qdrig.o pippo.o

csh> touch *.c
```

Visualizza quello che farebbe, ma non esegue niente!

Aggiorna la data di tutti i file .c

esempio

alcuni target potrebbero non avere file da cui dipendono

Esempio:

```
clean:
  rm *.o
```

Il comando

>> **make clean**

cancella tutti i file oggetto

librerie di programmi

principale caratteristica del make:
capacita' di gestire in **maniera automatica**
operazioni complesse e ripetitive



strumento per la **installazione e la**
manutenzione del software in ambiente Unix

cosa e' una libreria di programmi

una **libreria di programmi** e' una **collezione di**
funzioni o sottoprogrammi gia' compilati (moduli
oggetto) da linkare ad una applicazione

di solito **ogni funzione e' in un file**
(ad es. la funzione somma nel file somma.c)



se si cambia una funzione **il make ricompilera'**
solo la funzione modificata

osservazione

Attorno ad una libreria di programmi ruotano
tre figure differenti:

- chi **sviluppa** la libreria
- chi **installa** la libreria sul sistema
- chi **utilizza** la libreria

ognuna di queste figure eseguirà
operazioni differenti

esempio

Si vuole creare, installare e utilizzare una
libreria composta da **4 funzioni**:

- **somma**
- **differenza**
- **moltiplicazione**
- **divisione**

Fase 1: creazione dei sorgenti

file somma.c

```
float somma(float a, float b) {
    float c;
    c = a + b;
    return c;
}
```

analogamente per differenza,
moltiplicazione e divisione

Fase 1: creazione dell'header file

oltre ai 4 file sorgente e' necessario creare un **header file** con le dichiarazioni delle funzioni che sara' incluso nel programma dell'utente finale

file operazioni.h

```
float somma(float, float);
float differenza(float, float);
float moltiplicazione(float, float);
float divisione(float, float);
```

Fase 1:



i 5 file vengono depositati
in un **"software repository"**
(il piu' famoso e':
www.netlib.org)

get get
get get



chi vuole installare la libreria
deve **scaricarsi i 5 file**
tramite una sessione ftp

e se la libreria e' composta da **500 file**?

E' possibile **"in scatolare"** tutti i file
in un **unico file**?

Fase 1: il comando tar

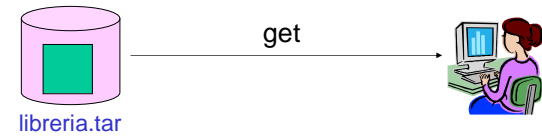
Il comando **tar** racchiude i file di una directory in un **unico archivio** rispettando la loro collocazioni in eventuali sottodirectory

```
>> ls
include src
>> ls src
differenza.c divisione.c moltiplicazione.c somma.c
>> ls include
operazioni.h
>> tar -cvf libreria.tar *.*
include/operazioni.h
src/differenza.c
src/divisione.c
src/moltiplicazione.c
src/somma.c
```

Fase 1: principali opzioni di tar

- -c crea un archivio
- -f definisce il nome dell'archivio
- -v modalita' verbose
- -d cancella un file dall'archivio
- -t stampa il contenuto dell'archivio
- -x estrae il contenuto dell'archivio

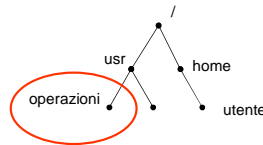
Fase 1:



Tutta la libreria si trova
in un **unico file archivio** !!!

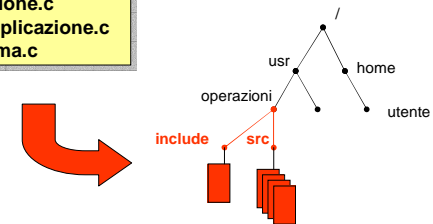
Fase 2: installazione

L'amministratore decide
dove installare la libreria



Fase 2: apertura dell'archivio

```
>> cd /usr/operazioni
>> tar -xvf libreria.tar
include/operazioni.h
src/differenza.c
src/divisione.c
src/moltiplicazione.c
src/somma.c
```



Fase 2: installazione

L'amministratore del sistema (o chi ha sviluppato la libreria) scrivono un opportuno **makefile**

Makefile: prima versione

```
all: somma.o differenza.o moltiplicazione.o divisione.o
somma.o: somma.c
    cc -c somma.c
differenza.o: differenza.c
    cc -c differenza.c
moltiplicazione.o: moltiplicazione.c
    cc -c moltiplicazione.c
divisione.o: divisione.c
    cc -c divisione.c
```

Fase 2: osservazione

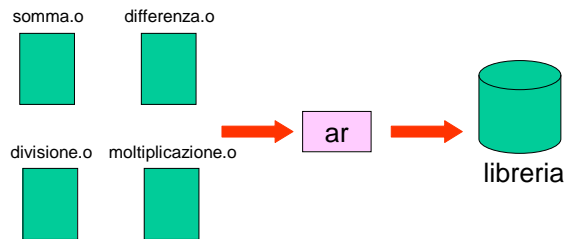
un utente che deve usare la libreria deve linkare al proprio programma **tutti i file della libreria**

Problema:

e' possibile fare riferimento all'insieme delle funzioni come **un'unica libreria** di programmi?

Fase 2: il comando ar

il comando **ar** raggruppa un insieme di file in **un'unica libreria di programmi**



Fase 2: il comando ar

```
>> ar [opz] archivio file1 file2 file3 ...
```

Alcune opzioni

- -r crea l'archivio (eventualmente rimpiazza i file)
- -t stampa il contenuto dell'archivio
- -d cancella un file dall'archivio
- -v modalita' "verbose"

Fase 3: uso della libreria

programma per il calcolo di:

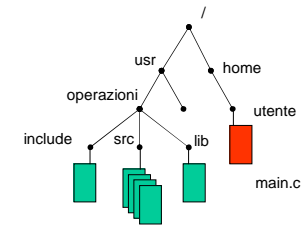
$$R = ((a+b)*(a-b))/2$$

```
#include <operazioni.h>
main () {
    float s, d, m, r ;
    scanf (&a, &b);
    s = somma(a, b);
    d = differenza (a, b);
    m = moltiplicazione (s, d);
    r = divisione (m, 2);
    printf("risultato = %f\n", r);
}
```

Fase 3: compilazione

```
>> cc -o eseg main.c liboperazioni.a
```

Attenzione:
operazioni.h e liboperazioni.a non si trovano nella directory corrente



Fase 3: problema:

di solito le librerie sono installate dall'amministratore di sistema in modo da poter essere **utilizzate da tutti gli utenti**

come fare **riferimento** ad una libreria che **non si trova nella directory corrente?**

come fare **riferimento** ad un header file che **non si trova nella directory corrente?**

Fase 3: soluzione

opzioni al compilatore cc:

- -L dichiara le directory dove **cercare le librerie**
- -I dichiara le directory dove **cercare gli header file**



```
>> cc -o eseg main.c -I/usr/operazioni/include
-L/usr/operazioni/lib liboperazioni.a
>>
```

Fase 3: ulteriore opzione a cc

Se il nome del file che contiene la libreria :

- comincia con **lib**
- ha estensione **.a**

e' possibile usare l'opzione **-l** al compilatore

esempio: **liboperazioni.a**

```
>> cc -o eseg main.c -I/usr/operazioni/include  
-L/usr/operazioni/lib -loperazioni  
>>
```