Lezione 2 - 15/3/21 (Docker, gcc, make, C)

Docker vs VM

Docker

- Virtualizzazione a livello OS
- Containers condividono Kernel
- Avvio e creazione in secondi
- Leggere (KB/MB)
- Si distruggono e si rieseguono
- Utilizzo leggero di risorse
- Minore sicurezza

VM

- Virtualizzazione a livello HW
- Ogni VM ha il suo OS
- Avvio e creazione in minuti
- Pesanti (GB)
- Si trasferiscono
- Utilizzo intenso di risorse
- Maggiore sicurezza
- Maggiore controllo

Comandi base Docker

`docker "comando"`

- **version** → Mostra la versione di docker

- **run <image>** → Crea un immagine pronta all'esecuzione
- **start/stop <container>** → Avvia o ferma l'esecuzione di un container
- **exec [options] <container> <command>** → Esegue il comando nel container

Parametri 'run' opzionali

- **--name <nome>** → Assegna un nome specifico al container
- **-d (detach mode)** → Scollega il container (ed il suo input/output) dalla console
- **-ti** → Esegue un container in modalità interattiva
- **--rm** → Elimina container all'uscita

Dockerfile

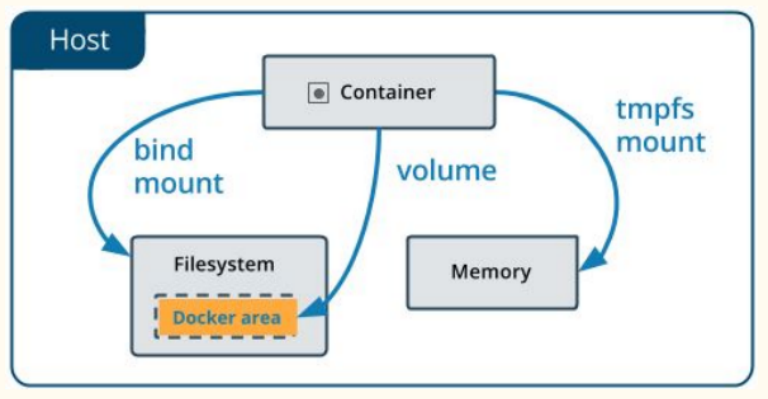
- **--hostname <nome>** → Imposta l'hostname nel container
- **--workdir <path>** → Imposta la cartella di lavoro nel container
- **--network host** → Collega il container alla rete locale
- **--privileged** → Esegue il container con i privilegi dell'host

I dockerfile sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scratch'. La creazione a partire da un dockerfile viene gestita attraverso del caching che ne permette la ricompilazione rapida in caso di piccoli cambiamenti.

`docker buld -t ubuntu/custom - < dockerfile`

Gestione dei volumi

Docker salva i file persistenti su *bind mount* o su dei *volumi*. sebbene i **bind mount** siano strettamente collegati con il filesystem dell'host OS, consentendo dunque una facile comunicazione con il containers, i **volumi** sono ormai lo standard in quanto indipendenti, facili da gestire e più in linea con la filosofia di docker.



Sintassi dei comandi

`docker volume create <volumeName>` → Crea un nuovo volume

`docker volume ls` → Mostra i volumi esistenti

`docker volume inspect <volumeName>` → Esamina un volume

`docker volume rm <volumeName>` → Rimuovi volume

`docekr run -v <volume>:<path/in/container> <image>` → Crea un nuovo container con il **volume** specificato montato nel percorso specificato

`docekr run -v <pathHost>:<path/in/container> <image>` → Crea un nuovo container con un **bind mount** specificato montato nel percorso specificato

Il nostro ambiente - Ubuntu 20.04

```
docker run -ti --rm --name="labOS" --privileged \
-v /:/host -v "$(pwd):/home/labOS" \
--hostname "labOS" --workdir /home/labOS \
ubuntu:20.04 /bin/bash
```

Poi:

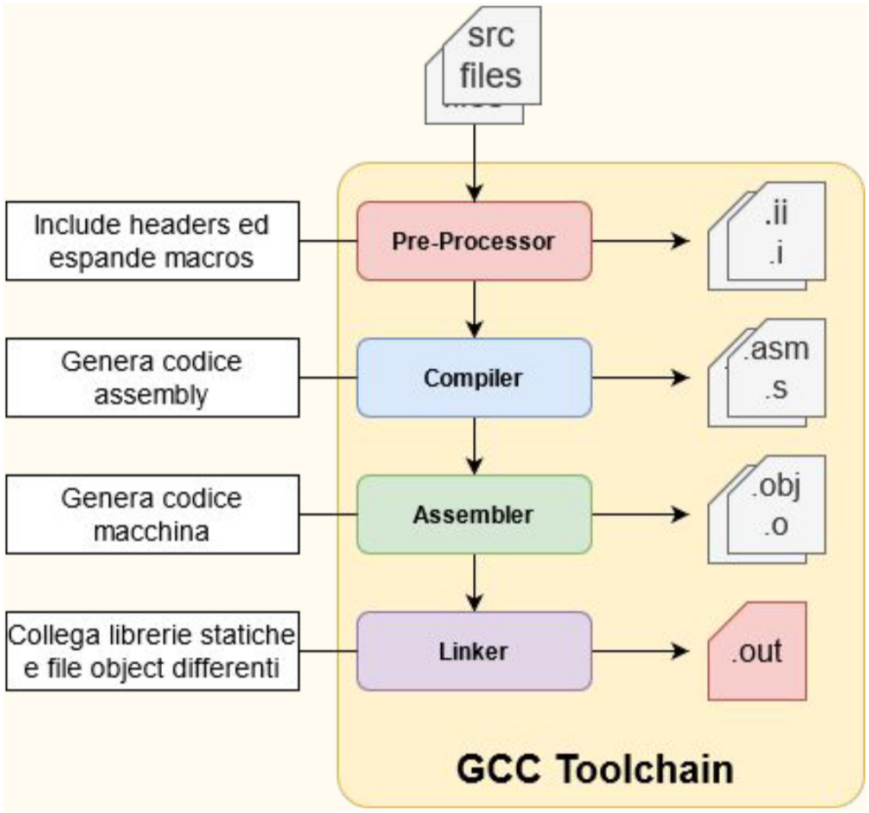
```
apt-get update && apt-get install -y nano build-essential
```

Infine:

```
docker start labOS
```

GCC (Gnu Compiler Collection)

E' un compilatore per vari linguaggi ma noi lo useremo per C.



Compilazione

Posso eseguire la compilazione di un "file.c" passo per passo:

`gcc -E <sorgente.c> -o <preProcessed.ii | .i>`

`gcc -S <preprocessed.i | .ii> -o <assembly.asm | .s>`

`gcc -c <assenbly.asm | .s> -o <objectFile.obj | .o>`

`gcc <objectFile.obj | .o> -o <executable.out>`

N.B.

L'input di ogni comando può essere il file sorgente, e l'ultimo comando è in grado di creare direttamente l'eseguibile.

N.B.

L'assembly ed il codice macchina generato dipendono dall'architettura di destinazione.

Make

Make tool

E' uno strumento della collezione GNU che può essere usato per gestire la compilazione automatica e selettiva di grandi e piccoli progetti. Make consente di specificare delle dipendenze tra vari file in modo da consentire la compilazione solo di certi file modificati o dai quali dipendono. (Assume alcune capacità di uno script bash).

Makefile

Make può eseguire dei makefiles che contengono tutte le direttive utili alla compilazione di un'applicazione o allo svolgimento di un task.

`makefile -f makefile`

Se scrivo il comando soprastante senza argomenti in automatico cercherà i file nominati: "GNUmakefile", "makefile" e "Makefile" in questo ordine; in caso non li trova da un errore.

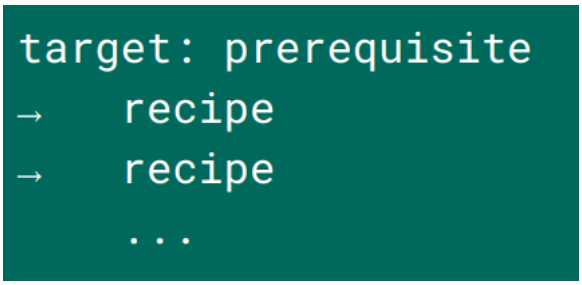
Posso includere dei makefile in un makefile principale per gestire meglio grandi progetti.

Target, prerequisite and recipes

Una **ricetta** è una lista di comandi bash che vengono eseguiti indipendentemente dal resto del makefile e ogni **ricetta** viene eseguita su una shell differente.

I **target** sono generalmente dei files generati da uno specifico insieme di regole.

Ogni target può specificare dei **prerequisiti** ovvero degli altri file che devono esistere affinchè le regole di un target vengano eseguite. Un prerequisito può essere esso stesso un target!



L'esecuzione di un file make inizia specificando uno o più target:

`make -f makefile target1 ...`

e prosegue a seconda dei vari prerequisiti

Regolette:

- Righe vuote vengono ignorate
- I commenti si fanno con il carattere '#' (e si concludono in caso con "\#")
- Le ricette devono partire con un carattere di "tab" (Non spazi)
- Una ricetta che parte con '@' oltre che al "tab" non viene visualizzata altrimenti i comandi vengono mostrati ed eseguiti
- Una riga con un solo "tab" è una ricetta vuota

Target speciali

Il target di default eseguito quando non ne viene passato alcuno è il primo disponibile.

```
all: ...
  rule

.SECONDARY: target1 ..

.PHONY: target1 …

%.s: %.c
  #prova.s: prova.c
  #src/h.s: src/h.c
```

.INTERMEDIATE e **.SECONDARY:** hanno come prerequisiti i target "intermedi". Nel primo caso sono poi rimossi nel secondo sono mantenuti a fine esecuzione.

.PHONY: ha come prerequisiti i target che non corrispondono a dei files o comunque da eseguire "sempre" senza verificare l'eventuale file omonimo.

In un target, **'%'** sostituisce qualunque stringa. In un prerequisito corrisponde alla stringa sostituita nel target.

Variabili utente e automatiche

Le variabili utente si definiscono con la sintassi "nome:=valore" o "nome=valore" e vengono usate con "**$(nome)**". Inoltre, possono essere sovrascritte da riga di comando con make nome=value.

```
target: pre1 pre2 pre3
  echo $@ is 'target'
  echo $^ is 'pre1 pre2 pre3'
  echo $< is 'pre1'
```

Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente.

```
ONCE:=hello $(LATER)
EVERY=hello $(LATER)
LATER=world

target1:
  echo $(ONCE) # 'hello'
  echo $(EVERY) # 'hello world'
```

Con "=" alla seconda riga metto un riferimento alla variabile così se la cambio di conseguenza cambia l'assegnazione. Nella prima riga invece con ":=" metto dentro il valore della variabile in quel punto che era nullo dunque se cambia la variabile non cambierà il valore dove la sto assegnando.

Funzioni speciali

$(eval ...) → Consente di creare nuove regole make dinamiche.

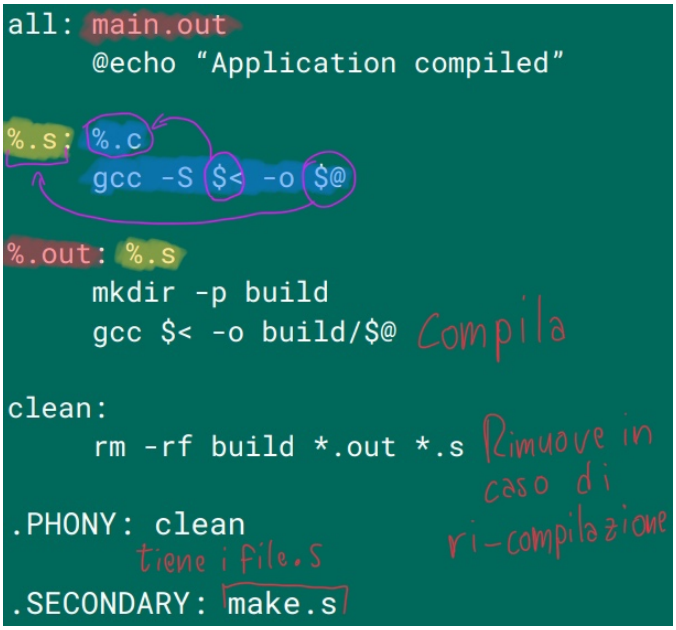
```
LATER=hello
PWD=$(shell pwd)
OBJ_FILES:=$(wildcard *.o)

target1:
  echo $(LATER) #hello
  $(eval LATER+= world)
  echo $(LATER) #hello world
```

$(shell ...) → Cattura l'output di un comando shell.

$(wildcard ...) → Restituisce un elenco di file che corrispondono alla stringa specificata.

Esempio:

```
all: main.out
     @echo "Application compiled"

%.s: %.c
     gcc -S $< -o $@

%.out: %.s
     mkdir -p build
     gcc $< -o build/$@

clean:
     rm -rf build *.out *.s

.PHONY: clean

.SECONDARY: make.s
```



https://pages.di.unipi.it/gadducci/PR1L-13/exeC/make.pdf

Linguaggio C

Direttive

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con '#' e può essere di vari tipi:

- **#include <lib>** → copia il contenuto del file *lib* (cercando nelle cartelle delle librerie) nel file corrente.
- **#include "lib"** → come sopra ma cerca prima nella cartella corrente.
- **#define VAR VAL** →crea una costante **VAR** con il contenuto **VAL,** e sostituisce ogni occorrenza di **VAR** con **VAL**.
- **#define MUL(A,B) A*B** → dichiara una funzione con parametri **A** e **B.** (Queste funzioni hanno una sintassi limitata!).
- **#ifdef, #ifndef, #if, #else, #endif** → rende l'inclusione di parte di codice dipendente da una condizione. (I primi 3 vanno seguiti da "**#endif**").

Main.c

`int main(int argc, char * argv[])`

argc → Contiene il numero di argomenti passati (+1 per il nome del programma).

****argv** → Contiene gli argomenti passati (al primo posto c'è il nome del "file.c").