

# LabSO 2023

Laboratorio Sistemi Operativi - A.A. 2022-2023

---

Michele Grisafi - [michele.grisafi@unitn.it](mailto:michele.grisafi@unitn.it)

# Docker

---

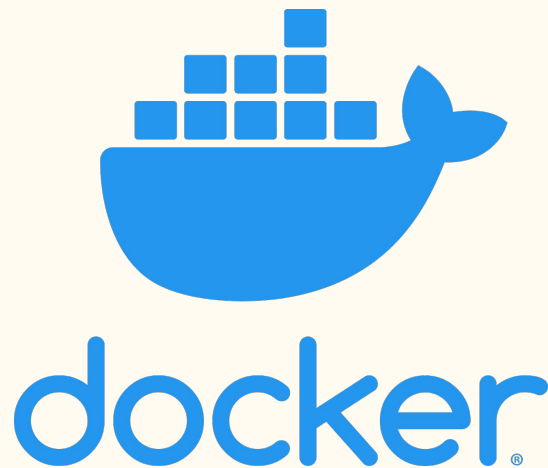
# Docker, cos'è?

Tecnologia di virtualizzazione a livello del sistema operativo che consente la creazione, la gestione e l'esecuzione di **applicazioni** attraverso containers.

I **containers** sono ambienti leggeri, dinamici ed isolati che vengono eseguiti sopra il kernel di Linux.



... e molti altri!



# Docker Container

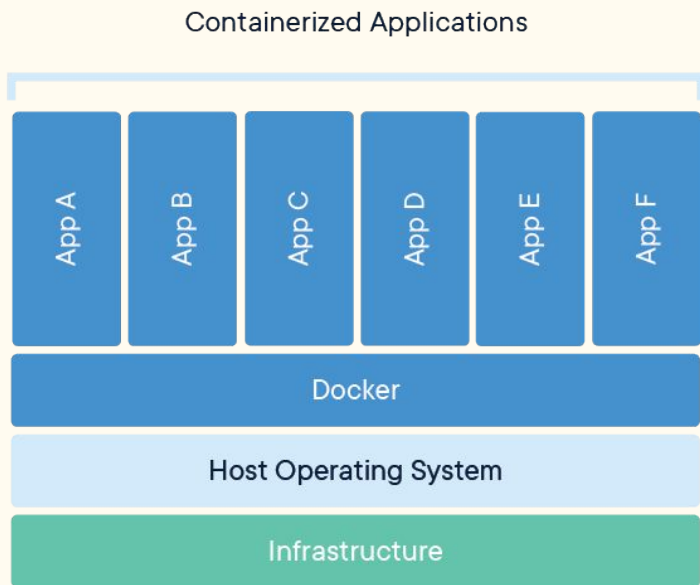
- Virtualizzazione a livello OS
- Containers condividono kernel
- Avvio e creazione in secondi
- Leggere (KB/MB)
- Utilizzo leggero di risorse
- Si distruggono e si rieseguono
- Minore sicurezza

**NB:** Basati su immagini delle quali se ne trovano tantissime già pronte!

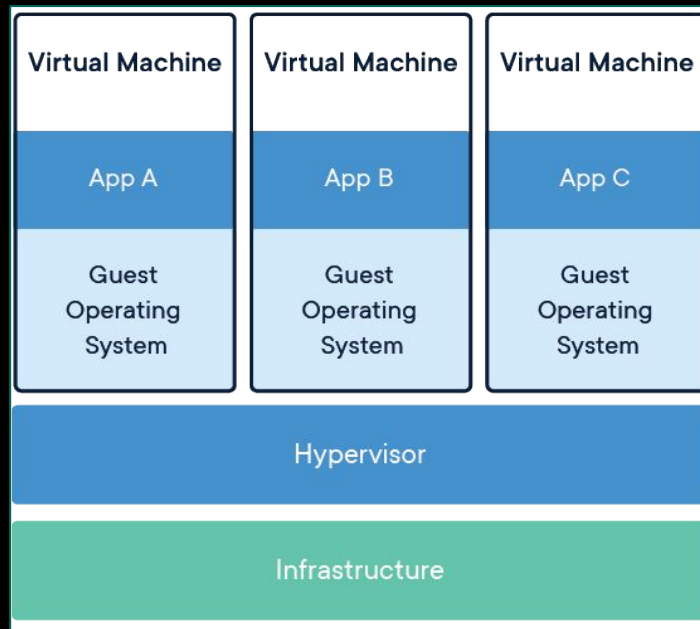
# Virtual Machine

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

# Docker Container

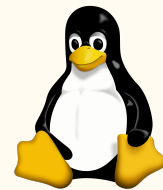


# Virtual Machine



# Compatibilità sui vari OS

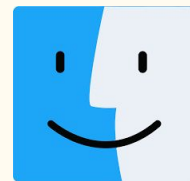
**Linux:** docker gestisce i containers usando il kernel linux nativo.



**Windows:** docker gestisce i containers usando il kernel linux tramite WSL2 (originariamente virtualizzato tramite Hyper-V). Gestito da un'applicazione.



**Mac:** docker gestisce i containers usando il kernel linux virtualizzato tramite xhyve hypervisor. Gestito da un'applicazione.



# Containers e immagini

Un'immagine docker è un insieme di “istruzioni” per la creazione di un container. Essa consente di raggruppare varie applicazioni ed eseguirle, con una certa configurazione, in maniera rapida attraverso un container.

I containers sono invece gli ambienti virtualizzati gestiti da docker, che possono essere creati, avviati, fermati ed eliminati. I container devono essere basati su un'immagine!

(esiste un'opzione di creazione da zero - FROM scratch - che però è raramente usata in pratica)

# Gestione dei containers

`docker run [options] <image>`: crea un nuovo container da un'immagine

`docker container ls [options]`: mostra i containers attivi (`[-a]` tutti)

`docker start/stop <container>`: avvia/ferma l'esecuzione del container

`docker exec [options] <container> <command>`: esegue il comando all'interno del container

`docker stats`: mostra le statistiche di utilizzo dei containers

E la panacea di tutti i dubbi... `docker <command> --help`



# 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 container in modalità interattiva\*
- `--rm`: elimina container all'uscita
- `--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

\*Per collegarsi `docker attach <container>`. Per scollegarsi `Ctrl+P`, `Ctrl+Q`

\*\* la modalità host non funziona su W10 e MacOS a causa della VM sottostante

# Esempi

- Esegui `docker run hello-world`
- Esegui `docker run -d -p 80:80 docker/getting-started` e collegati alla pagina “localhost:80” con un qualunque browser

# Gestione delle immagini

La community di docker offre migliaia di immagini pronte all'uso ma è possibile crearne di nuove.

`docker images`: mostra le immagini salvate

`docker rmi <imageID>`: elimina un'immagine (se non in uso!)

`docker search <keyword>`: cerca un'immagine nella repository di docker

`docker commit <container> <repository/imageName>`: crea una nuova immagine dai cambiamenti nel container

Altrimenti si possono creare nuove immagini con dei dockerfile...

# Dockerfile

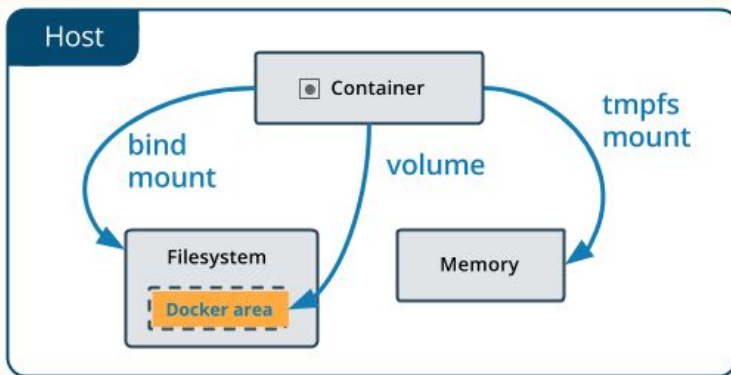
I dockerfile sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scratch'. La creazione a partire da un docker file viene gestita attraverso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

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install build-essential nano -y
RUN mkdir /home/labOS
CMD cd /home/labOS && bash
```

```
docker build -t labos/ubuntu - < 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 volume

`docker volume rm <volumeName>`: rimuovi volume

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

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

# Il nostro ambiente

```
docker run -ti --rm --name="lab0S" --privileged \
-v /:/host -v "$(pwd):/home/lab0S" \
--hostname "lab0S" --workdir /home/lab0S \
ubuntu:22.04 /bin/bash
```

Ed eseguire:

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

Quando il container è pronto si può fare il commit per salvare le modifiche in una nuova immagine (es.: `docker commit ..... localhost.ext/unitn:labso2023`)

**NB:** se non aggiungete il flag `--rm`, ogni volta che uscite il container verrà fermato e potrà essere riavviato con `docker start lab0S`

# Il nostro ambiente... oppure

Usare il dockerfile in slide #12 per creare un'immagine del laboratorio:

```
docker build -t labOS/ubuntu - < dockerfile
```

E poi è possibile eseguire un container basato sulla nuova immagine 'labOS/ubuntu':  
usare il comando

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



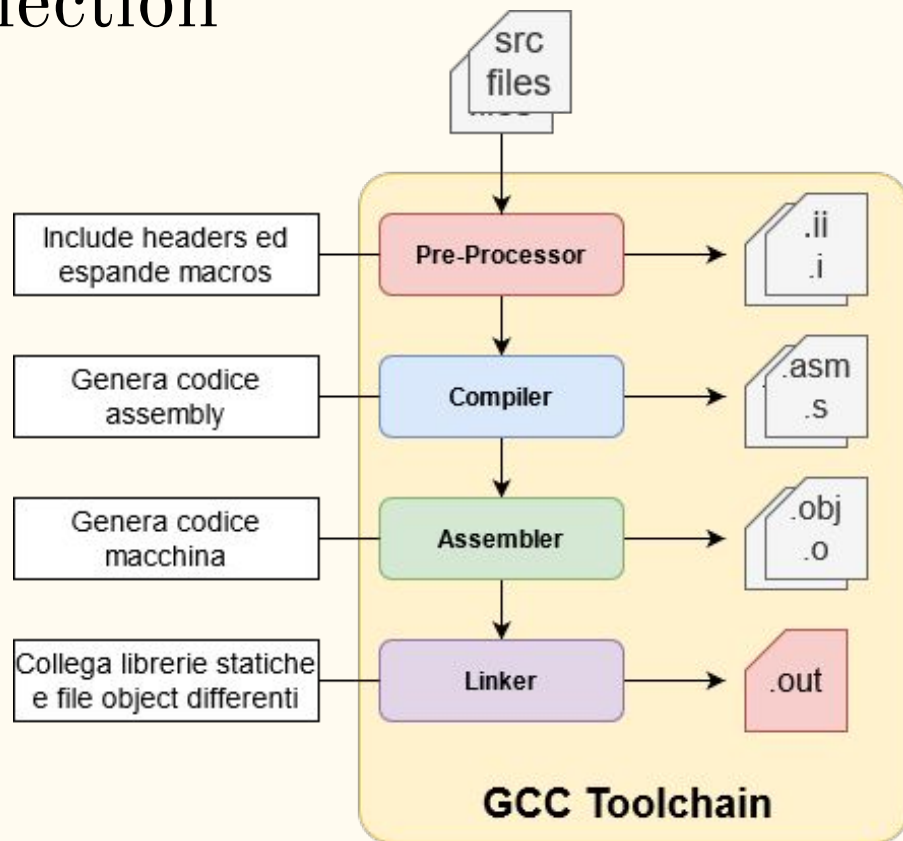
gcc

—

# GCC = Gnu Compiler Collection

Insieme di strumenti open-source che costituisce lo standard per la creazione di eseguibili su Linux.

GCC supporta diversi linguaggi, tra cui C, e consente la modifica dei vari passaggi intermedi per una completa personalizzazione dell'eseguibile.



# La compilazione

Gli strumenti GCC possono essere chiamati singolarmente:

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

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

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

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

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

**NB:** l'assembly ed il codice macchina generato dipendono dall'architettura di destinazione

# Un esempio

1. Provate a compilare una semplicissima applicazione, invocando ogni step singolarmente osservandone l'output.
2. Provate ad aggiungere `#include <stdio.h>` ad inizio file e ripetete il tutto

```
1 //main.c
2 void main(){
3     return;
4 }
```

```
1 //main.c
2 #include <stdio.h>
3 void main(){
4     return;
5 }
```

# Make

---

# Make tool

Il Make tool è 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 i vari file, per esempio consentendo solo la compilazione di librerie i cui sorgenti sono stati modificati.

Make può anche essere usato per gestire il deployment di un'applicazione, assumendo alcune delle capacità di uno script bash.

# Makefile

Make può eseguire dei makefiles i quali contengono tutte le direttive utili alla compilazione di un'applicazione (o allo svolgimento di un altro task).

```
make -f makefile
```

In alternativa, il comando **make** senza argomenti processerà il file '**makefile**' presente nella cartella di lavoro (nell'ordine cerca: GNUmakefile, makefile e Makefile)

Makefiles secondari possono essere inclusi nel makefile principale con delle direttive specifiche, consentendo una gestione più articolata di grandi progetti.

# Target, prerequisite and recipes

Una *ricetta o regola* è una lista di comandi bash che vengono eseguiti indipendentemente dal resto del makefile.

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 affinché 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.

```
target: prerequisite
→  recipe/rule
→  recipe/rule
...
```

```
target1: target2 target3
        rule (3)
        rule (4)
        ...

target2: target3
        rule (1)

target3:
        rule (2)
```



# Sintassi

Un makefile è un file di testo “plain” in cui righe vuote e parti di testo dal carattere “#” fino alla fine della riga non in una ricetta (considerato un commento: sempre che non sia usato l’escaping con “\#” o che compaia dentro una stringa con ' o ") sono ignorati.

Le ricette DEVONO iniziare con un carattere di TAB (**non spazi**).

Una ricetta che (a parte il TAB) inizia con @ non viene visualizzata in output, altrimenti i comandi sono visualizzati e poi eseguiti.

Una riga con un singolo TAB è una ricetta vuota.

Esistono costrutti più complessi per necessità particolari (ad esempio costrutti condizionali)

# Target speciali

Se non viene passato alcun target, viene eseguito quello di default: il primo disponibile. Esistono poi dei target che hanno un significato e comportamento speciale.

**.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.

```
target: prerequisite
→    rule
→    rule
    ...
```

```
all: ...
    rule

.SECONDARY: target1 ..

.PHONY: target2 ...

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

# 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**.

Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente. La variabile **SHELL** può essere usata per specificare la shell di riferimento per l'esecuzione del makefile. Per esempio, **SHELL=/bin/bash** abilita bash anziché SH.

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

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

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

# Funzioni speciali

`$(eval ...)`: consente di creare nuove regole make dinamiche.

`$(shell ...)`: cattura l'output di un comando shell.

`$(wildcard *)`: restituisce un elenco di file che corrispondono alla stringa specificata.

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

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

# Make file - 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
```

# Esercizio per casa

Creare un makefile con una regola `help` di default che mostri una nota informativa, una regola `backup` che crei un backup di una cartella appendendo “.bak” al nome e una `restore` che ripristini il contenuto originale. Per definire la cartella sorgente passarne il nome come variabile, ad esempio:

```
make -f mf-backup FOLDER=...
```

(la variabile `FOLDER` è disponibile dentro il makefile)

# CONCLUSIONI

Docker, GCC e make possono essere utilizzati per la gestione delle varie applicazioni in C. Ognuno di questi strumenti non è indispensabile ma permette di creare un flusso di lavoro coerente e strutturato.