Sistemi Operativi

Enrico Favretto

A.A. 2022/2023

Indice

1.1 Lab-01: Bash/Terminale

Terminale: è l'ambiente testuale di interazione con il sistema operativo. Tipicamente usato come applicazione all'interno dell'ambiente grafico ed è possibile avviarne più istanze, è anche disponibile all'avvio.

Shell: All'interno del terminale, l'interazione avviene utilizzando una applicazione specifica in esecuzione al suo interno, comunemente detta Shell. Essa propone un prompt per l'immissione diretta dei comandi da tastiera e fornisce un feedback testuale, è anche possibile eseguire sequenze di comandi preorganizzate contenute in file testuali, chiamati *script* o *batch*. A seconda della modalità alcuni comandi possono avere senso o meno o comportarsi in modo particolare.
L'insieme dei comandi e delle regole di composizione costituisce un linguaggio di programmazione orientato allo scripting.

Bash: Esistono molte shell. La shell **Bash** è una di quelle più utilizzate e molte sono comunque sono molto simili tra di loro, ma hanno sempre qualche differenza.
Tipicamente - almeno in sessioni non grafiche - al login un utent ha associata una shell particolare

POSIX: **P**ortable **O**perating **S**ystem **I**nterface for Uni**x**: è una famiglia di standard IEEE.
Nel caso delle shell in particolare definisce una serie di regole e comportamenti che possono favorire la probabilità
La shell bash soddisfa molti requisiti POSIX, ma presenta alcune differenze ed "estensioni" per agevolare la programmazione.

Comandi Interattivi: La shell attende un input dall'utente e al completamento, di solito premendo INVIO, lo elabora.
Per indicare lattesa mostra allutente un PROMPT (pu essere modificato).
Fondamentalmente si individuano 3 canali:

- Standard input (tipicamente la tastiera), canale 0, detto *stdin*
- Standard output (tipicamente il video), canale 1, detto *stdout*
- Standard error (tipicamente il video), canale 2, detto *stderr*

1.1.1 Struttura comandi

Struttura generale comandi: Solitamente un comando è identificato da una parola chiave cui possono seguire uno o pi "argomenti" opzionali o obbligatori, accompagnati da un valore di riferimento o meno (in questo caso hanno valore di "flag") e di tipo posizionale o nominale. A volte sono ripetibili.

ls -alh /tmp

gli argomenti nominali sono indicati con un trattino cui segue una voce (stringa alfanumerica) e talvolta presentano una doppia modalità di riferimento: breve (tipicamente voce di un singolo carattere) e lunga (tipicamente un termine mnemonico)

ls -h / ls -help

Termini:

- l'esecuzione dei comandi avviene "per riga" (in modo diretto quella che si immette fino a INVIO)
- un "termine" (istruzione, argomento, opzione, etc.) solitamente una stringa alfanumerica senza spazi
- spaziature multiple sono spesso valide come singole e non sono significative se non per separare i termini
- è possibile solitamente usare gli apici singoli/doppi per forzare una sequenza come singolo termine
- gli spazi iniziali e finali di una riga collassano
- le righe vuote sono ignorate

Commenti: È anche possibile utilizzare dei commenti da passare alla shell. L'unico modo formale è l'utilizzo del carattere # per cui esso e tutto ciò che segue fino al termine della riga viene ignorato

ls -la # this part is a comment

Comandi fondamentali: I comandi possono essere "builtin" (funzionalità intrinseche dell'applicazione shell utilizzata) o "esterni" (applicazioni eseguibili che risiedono su disco).

- clear (*clearing the terminal screen in Linux*)
- pwd (*prints the current working directory path, starting from the root(/)*)
- ls (*list files or directories*)
- cd (*list the contents of the current directory*)
- wc (*creates a new file in your current working directory with the name wc with the output of your ls command*)
- date (*displays and sets the system date and time*)
- cat (*allows us to create single or multiple files, view content of a file, concatenate files and redirect output in terminal or files*)
- echo (*displaying lines of text or string which are passed as arguments on the command line*)

- alias/unalias (*is a way to make a complicated command or set of commands simple/remove the definition for each alias name specified*)
- test (*used as part of the conditional execution of shell commands*)
- read (*attempts to read up to count bytes from file descriptor fd into the buffer starting at buf*)
- file (*tells you the type of a file*)
- chown (*changes the user and/or group ownership of each given file*)
- chmod (*used to manage file system access permissions on Unix and Unix-like systems*)
- cp/mv (*copy a file/directory into another directory/moves a file/directory into another directory*)
- help (*listing all possible commands that are pre-installed in Ubuntu*)
- type (*used to describe how its argument would be translated if used as commands*)
- grep (*allows you to find a string in a file or stream*)
- function (*a reusable block of code*)

Canali Input/Output
Ogni comando lavora su un insieme di canali, come standar output o un file descriptor

Ridirizionamento di base
I canali possono anche essere ridirezionati, ad esempio

- ls 1 >/tmp/out.txt 2 >/tmp/err.txt
- ls nonExistingItem 1 > */tmp/err.txt* 2 > &1
- <: $command < file$ invia l'input al comando (file.txt read-only)
  mail -s "subject" $rcpt < context.txt$ (anzichè iterattivo)
- <>: come sopra solo che il file.txt è aperto in read-write
- $source > target : command1 > out.txt2 > err.txt$
  ridireziona source su target (source può essere sottointeso, target può essere un canale)
- > |: si comporta come > ma forza la sovrascrittura anche se bloccata nelle configurazioni
- >>: si comporta come > ma opra un append se la destinazione esiste

- <<: here-document, consente all'utente di specificare un terminatore testuale, dopodiche accetta l'input fino alla ricezione di tale terminatore
- <<<: here-string, consente di fornire input in maniera non iterattiva

Esistono molte varianti e possibilità di combinazione dei vari operatori
Un caso molto usato è la soppressione dell'output (utile per gestire solo side-effects, come il codice di ritorno), esempio:
$typecommand 1 > dev/null 2 > \&1$ (per sapere se command esiste)

1.1.2 Ambiente e variabili

Ambiente e variabili

- La shell può utilizzare variabili per memorizzare e recuperare i valori
- I valori sono generalmente trattati come stringhe o interi: sono presenti anche semplici array
- Il formato del nome è del tipo: $\hat{}[\_[:alpha:]][\_[:alnum:]]\$$
- Per set (impostare) / get (accedere) al valore di una variabile:
  - Il set si effettua con $[export]variabile = valore$, lo scope è generalmente quello del processo attuale: anteponendo export si rende disponibile anche ai processi figli
  - Il get si effettua con $\$variabile$ o $\${variabile}$
- La shell opera in un ambiente in cui ci sono alcuni riferimenti impostabili e utilizzabili attraverso l'uso delle cosiddette "variabili d'ambiente" con cui si intendono generalmente quelle con un significato particolare per la shell stessa, tra le variabili d'ambiente più comuni vi sono:

  $$SHELL, PATH, TERM, PWD, PS1, HOME$$

  Essendo variabili si impostano e usano come le altre

Variabili di sistema
Alcune variabili sono impostate e/o utilizzate direttamente dal sistema in casi particolari.

- SHELL: contiene il riferimento alla shell corrente (path completo)
- PATH: contiente i percorsi in ordine di priorità in cui sono cercati i comandi (separati da :)
- TERM: contiene il tipo di terminale corrente
- PWD: contiene la cartella corrente
- PS1: contiene il prompt e si possono usare marcatori speciali
- HOME: contiene la cartella principale all'utente corrente

1.1.3 Esecuzione comandi

Esecuzione comandi e $PATH
Quando si immette un comando (o una sequenza di comandi) la shell analizza quanto inserito (parsing) e individua le parti che hanno la posizione di comandi da eseguire: se sono interni ne esegue le funzionalit direttamente altrimenti cerca di individuare un corrispondente file eseguibile: questo normalmente cercato nel file-system solo e soltanto nei percorsi definiti dalla variabile PATH a meno che non sia specificato un percorso (relativo o assoluto) nel qual caso viene utilizzato esso direttamente. Dall'ultima osservazione discende che per un'azione abbastanza comune come lanciare un file eseguibile (non installato) nella cartella corrente occorre qualcosa come: ./nomefile

Array

- Definizione: lista=("a" 1 "b" 2 "c" 3) separati da spazi
- Output completo: {$*lista*[@]}
- Accesso singolo: {$*lista*[$x$]}
- Lista indici: {$!*lista*[@]}
- Dimensione: {#*lista*[@]}
- Set elemento: $lista[x] = value$
- Append: $lista+ = (value)$
- Sub-array: $\$lista[@] : s : n$, dall'indice s di lunghezza n

Variabili $$ e $?
Le variabili $$ e $? non possono essere impostate manualmente.

- $$: contiente il PID del processo attuale
- $?: contiente il codice di ritorno dell'ultimo comando eseguito

Esecuzione comandi e Parsing
La riga dei comandi è elaborata con una serie di azioni "in sequenza" e poi rielaborata più volte, tra le azioni ci sono:

- Sostituzioni speciali della shell
- Sostituzione variabili
- Elaborazione subshell

sono svolte con un ordine di priorità e poi l'intera riga è rielaborata

Concatenazioni Comandi
È possibile concatenare più comandi in un'unica riga in vari modi con effetti differenti

- comand1 ; comand2: concatenazione semplice, esecuzione in sequenza
- comand1 && comand2: concatenazione logica (AND), l'esecuzione procede solo se il comando precedente non fallisce (**codice ritorno 0**)
- comand1 —— comand2: concatenazione logica (OR), l'esecuzione procede solo se il comando precedente non fallisce (**codice riotrno NON 0**)
- comand1 — comand2 : concatenazione con piping

Operatori di piping (pipe): — e —&
La concatenazione con gli operatori di piping cattura l'output di un comando e lo passa in input al successivo

- ls — wc -l: cattura solo stdout
- ls —& wc -l: cattura stdout e stderr

Nota: il comando ls ha un comportamento atipico: il suo output di base è differente a seconda che il comando sia diretto al terminale o a un piping

Subshell
È possibile avviare una subshell, ossia un sotto-ambiente in vari modi, in particolare raggruppando i comandi tra parentesi tonde: (... comandi ...)
Spesso si usa il catturare l'output standard della sequenza che viene sostituito letteralmente e rielaborato, ci sono due modi per farlo: $ (... comandi ...) oppure`*comandi*`

Esempio

```
echo "/tmp" > /tmp/tmp.txt ; ls $(cat /tmp/tmp.txt)
```

i comandi sono eseguiti rispettando la sequenza:

1. `echo "/tmp" > /tmp/temp.txt`: crea un file temporaneo con /tmp
2. `ls $(cat /tmp/tmp.txt)`: è prima eseguita la subshell
   - `cat /tmp/tmp.txt` che genera in stdout `"/tmp"` e poi con sostituzione:
   - `ls /tmp` mostra il contenuto della cartella /tmp

1.1.4 Utilizzo

Espansione Aritmetica
La sintassi base per una subshell è da non confondere con l'espansione aritmetica che utilizza le doppie parentesi rotonde, all'interno delle doppie

parentesi rotonde si possono rappresentare varie espressioni matematiche inclusi assegnamenti e confronti

```
((a=7))((a++))((a<10))((a = 3>10?1:0))
```

```
b = $((c+a))
```

Confronti Logici
I costrutti fondamentali per i confronti logici sono il comando `test ...` e i ragruppamenti tra parentesi quadre singole e doppie: `[ ... ]`, `[[ ... ]]` Dove `test ...` e `[ ... ]` sono builtin equivalenti mentre `[[ ... ]]` è una coppia di **shell-keywords**
In tutti i casi il blocco di confronto genera il codice in uscita 0 in caso di successo, un valore differente altrimenti

- Built-in: sono sostanzialmente dei comandi il cui corpo dell'esecuzione è incluso nell'applicazione shell direttamente (non eseguibili esterni) e quindi seguono sostanzialmente le regole generali dei comandi
- Shell-keywords: sono gestite come marcatori speciali così che possono "attivare" regole particolari di parsing.

Un esempio di questo sono gli operatori `< e >` che normalmente valgono come ridirezionamento, ma all'interno di `[[ ... ]]` valgono come operatori relazionali
Tipologia operatori
Le parentesi quadre singole sono POSIX-compilant, mentre le doppie sono un'estensione bash. Nel primo caso gli operatori relazionali tradizionali $(>, <, ...)$ non possono essere usati perchè hanno un altro significato, salvo eventualmente l'utilizzo nelle doppie parentesi quadre, e quindi se ne usano di specifici che hanno però un equivalente più tradizionale nel secondo caso
Gli operatori e le sintassi variano a seconda del tipo di informazioni utilizzate: c'è una distinzione sottile tra confronti per stringhe e per interi. **Confronti tra Interi e Stringhe**

Interi	[...]	[[...]]
$uguale - a$	$-eq$	$==$
$diverso - da$	$-ne$	$!=$
$minore/minore - uguale$	$-lt/-le$	$</<=$
$maggiore/maggiore - uguale$	$-gt/-ge$	$>/>=$

Stringhe	[...]	[[...]]
$uguale - a$	$= o ==$	
$diverso - da$	$!=$	
$minore/minore - uguale$	$/ <$	$<$
$maggiore/maggiore - uguale$	$/ >$	$>$

Per le strighe nell'uguaglianza bisogna lasciare uno spazio prima e dopo
Confronti tra Operatori Unari Esistono alcuni operatori unari ad esempio per verificare se una stringa è vouta, oppure per controllare l'esistenza di file in una cartella

- [[`-f /tmp/prova`]]: è un file?
- [[`-e /tmp/prova`]]: file esiste?
- [[`-d /tmp/prova`]]: è una cartella?

N.B.: è possibile utilizzare sia [e sia [[
Negazione Il carattere ! viene usato per negare i confronti

1.1.5 Script bash

Script/Bash
È possibile raccogliere sequenze di comandi in un file di testo che può poi essere eseguito:

- Richiamando il tool "bash" e passando il file come argomento

```
bash ../file.sh
```

- Impostando il bit "x" e specificando il percorso completo, o solo il nome se la cartella è in $PATH

```
chmod +x ../file.sh && ../file.sh
```

Esempio:

- **Script**: "bashpid.sh"

```
echo $BASHPID
echo $( $BASHPID)
```

- **CLI**: `chmod +x ./bashpid.sh; echo $BASHPID; ./bashpid.sh`

Elementi particolari
Le righe vuote e i commenti (#) sono ignorati
La prima riga può essere un **metacommento**, detto hash-bang, she-bang, che identifica un'applicazione a cui passare il file stesso come argomento (tipicamete usato per identificare l'interprete utilizzato)
Sono disponibili anche variabili speciali:

- $: lista completa degli argomenti passati allo Script
- $#: numero di argomenti passati allo script
- $i: i-esimo argomento con i da 0 a n

Costrutti e Funzioni
Esistono costrutti come for-loop, if-else, while-loop che possono anche essere scritti su singola riga.
If-Else:

```
if [ $1 -lt 10 ]; then
     echo less than 10
elif [ $1 -ge 20 ]; then
     echo greater than 20
else
     echo between 10 and 20
fi
```

For-loop:

```
for i in ${!lista[@]}; do
     echo ${lista[$i]}
done
```

While-loop:

```
while [[ $i < 10 ]]; do
     echo $i; (( i++ ))
done
```

Funzioni

- `function_name () {`
  `local var1 = 'C' #variabile con scope locale`
  `echo $1 $2 #argomenti normali della funzione`
  `return 44 #codice di ritorno`
  `}`
- `function function_name () {commands;}`
- `function_name arg1 arg2 #chiamata della funzione con due argomenti`

Esercizi

1. Stampare T o F se il valore di input rappresenta un file o una cartella esistente.

   ```
   [-e $DATA] && echo "T" || echo "F"
   ```

2. Stampare "file", "cartella" o "?" a seconda di cosa l'input rappresenta nel fileStystem.

   ```
   [-f $DATA] && echo "file" || ([-d $DATA] && echo "cartella"
                              || echo "?")
   ```

Concatenazione di if uno dentro l'altro

3. Stampare il risultato di una semplice operazione aritmetica contenuta nel file rappresentato dal valore di input, oppure "?" se non esiste

```
[-f $DATA] && echo $(( $(cat $DATA) )) || echo "?"
```

   Controllo se input è un file e computo l'operazione all'interno di esso, tramite (( )), altrimenti ritorna il ?

4. Scrivere uno script che dato un qualunque numero di argomenti li restituisca in ordine inverso.

5. Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all'output generato da "ls", per semplicità assumere l'assenza di spazi

N.B.: Se eseguiti all'interno di uno script, la variabile DATA deve essere esportata con:

```
export DATA = valore
```

L'utilizzo di BASH, tramite CLI o Script, è basilare per poter interagire attraverso comandi con il file-system, con le risorse di sistema e per poter invocare tools e applicazioni.

1.2 Lab-02: Docker

Docker: Tecnologia di virtualizzazione a livello del sistema Operativo che consente la creazione, gestione e esecuzione di applicazioni attraverso containers
Containers: sono ambieni leggeri, dinamici ed isolati che vengono eseguiti sopra il kernel Linux

Docker Container	**Virtual Machine**
◦ *Virtualizzazione a livello OS*	• *Virtualizzazione a livello HW*
◦ *Containers condividono il kernel*	• *Ogni VM ha il suo OS*
◦ *Avvio e creazione in secondi*	• *Avvio e creazione in minuti*
◦*Leggere (KB/MB)*	• *Pesanti (GB)*
◦*Utilizzo leggero di risorse*	• *Utilizzo intenso di risorse*
◦*Si distruggono e si rieseguono*	• *Si trasferiscono*
◦*Minore sicurezza*	• *Maggiore sicurezza*
	• *Maggiore controllo*
◦*Basati su immagini (già pronte)*	

Compatibilità

- **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 container usando il kernel linux virtualizzato tramite xhyve hypervysor, gestito da una applicazione

Immagine: Un'immagine docker è un insieme di istruzioni per la creazione di un container. Essa consente di ragruppare varie applicazioni ed eseguirle, con una certa configurazione, in maniera più rapida attraverso un container
Container: I containers sono invece gli ambienti virtualizzati gestiti da docker, che possono essere creati, avviati, fermati ed eliminati. I container sono **basati** su una immagine

Gestione dei containers

- `docker run [options] <image>`: crea un nuovo container da un'immagine
- `docker container ls [options]`: mostra i containers attivi (`[-a]` li mostra tutti)
- `docker exec [options] <container> <command>`: esegue il comando al'interno del container
- `docker stats`: mostra le statistiche di utilizzo dei containers

- `docker <command> --help`: aiuto su specifico comando

Parametri opzionali

- `--name <nome>`: assegna un nome specifico al container
- `-d`: detach mode, cioè scollega il container
- `-ti`: esegue il container in modalità interattiva, per collegarsi `docker attach <container>`, scollegarsi `Crtl-P,Crtl-Q`
- `--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, la modalità host non funziona a causa della VM sottostante
- `--privileged`: esegue il container con i privilegi dell'host

Gestione delle immagini
È possibile creare nuove immagini delle docker

- `docker images`: mostra le immagini salvate
- `docker rmi <imageID>`: elimina immagine se non in uso
- `docker search <keyword>`: cerca un'mmagine nella repository di docker
- `docker commit <container> <repository/imageName>`: crea una nuova immagine dai cambiamenti nel container

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.

Esempio:
```
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 pi in linea con la filosofia di docker.

Sintassi 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>`: rimuove il 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 mostrato nel percorso specificato

1.2.1 GCC

GCC
GCC = Gnu Compiler Collection, insiemi di strumenti open-source che costituisce lo standard per la creazione di eseguibili Linux. Supporta diversi linguaggi e consente la modifica dei vari passaggi intermedi per una completa personalizzazione dell'eseguibile
Compilazione
Gli strumenti gcc possono essere chiamati singolarmente:

- `gcc -E <sorgente.c> -o <preProcessed.ii|i>`
- `gcc -S <preProcessed.ii|i> -o <assembly.asm|.s>`
- `gcc -c <assembly.asm|.s> -o <objectFile.obj|.o>`
- `gcc <objectFile.obj|.o> -o <executable.out>`

l'input di ogni comando può essere il file sorgente, e l'ultimo comando è in grado di creare direttamente l'eseguibile, inoltre l'assembly e il codice macchina generato dipendono dall'architettura di destinazione

1.2.2 Make/Makefile

Make
Make tool: è uno strumento della collezione GNU che può essere usato per gestire la compilazione automatica e selettiva di grandi e piccoli progetti. Consente di specificare delle dipendenze tra i vari file, per esempio consentendo solo la compilazione delle librerie in cui i sorgenti sono stati modificati
Può anche essere usato per gestire un deployment di una applicazione, assumendo alcune delle capacità di uno scipt bash
Makefile: Make può eseguire dei makefiles i quali contengono tutte le direttive utili alla compilazione di un'applicazione

```
make -f makefile
```

in alternativa, il comando make senza argomenti processerà il 'makefile' presente nella cartella di lavoro.
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** 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.
- **Prerequisiti**: ovvero degli altri file che devono esistere affinchè le regole di un target vengano eseguite. Un prerequisito pu essere esso stesso un target!

L'esecuzione di un file make inizia specificando uno o pi target `make -f makefile target1 ...` e prosegue a seconda dei vari prerequisiti.

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

Esempio:

```
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 il carattere # fino alla fine della riga non in una ricetta (considerato un commento)
Le ricette/rule devono iniziare con il carattere TAB
Una ricetta che inizia con TAB@ 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.

Target Speciali
Se non viene passato alcun target viene eseguito quello di default: il primo disponibile. Esistono poi dei target con 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, il simbolo % sostituisce una qualunque stringa, in un prerequisito corrisponde alla stringa sostituita nel target
Esempio:

```
all:  ...
      rule

.SECONDARY: target1 ..

.PHONY: target2 ..

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

Variabile utenti e automaitche

- **Utente**: Le variabili utente si definiscono con la sintassi `nome:=valore` oppure `nome=valore` e vengono usate con `$(nome)`, inoltre possono essere sovrascritte da riga di comando cone `make nome=value`
- **Automatiche**: 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 di makefile, esempio SHELL=/bin/bash abilita bash anzichè SH

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

Esempio:

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

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

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
```

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

1.3 Lab-03: Fondamenti di C

C Fundamentals

- **Tipi e Casting**

  C è un linguaggio debolmente e tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti:

  ```
  float a = 3.5;
  int b = (int)a
  ```

  La grandezza delle variabili è dipendente dall' architettura di riferimento i valori massimi per ogni tipo cambiano a seconda se la variabile è signed o unsigned.

$void = 0byte$	$float = 4bytes$
$char = 1byte$	$long = 8bytes$
$short = 2bytes$	$double = 8bytes$
$int = 4bytes$	$longdouble = 16bytes$

- **sizeof**: Si tratta di un operatore che elabora il tipo passato come argomento (tra parentesi) o quello dellespressione e restituisce il numero di bytes occupati in memoria

- **Puntatori**
  C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: * ed &.

  - **Variabili**:
    : ha significati diversi a seconda se usato in una dichiarazione o in una assegnazione
    `int *pointer; // crea un puntatore ad intero`
    `int valore = *(pointer) // ottiene un valore puntato`
    &: ottiene l'indirizzo di memoria in cui è collocata una certa variabile.
    `long whereIsValore = &valore`
  - **Funzioni**: C consente anche di creare dei puntatori a delle funzioni: puntatori che possono contenere l'indirizzo di funzioni differenti
    `float (*punt)(float,float)`
    `ret_type (*ptnName)(argType, argType,....)`

- **main.c**

  A parte in casi particolari l'applicazione deve avere una funzione "main" che è utilizzata come punto di ingresso, il valore di ritorno è un int che rappresenta il codice di uscita dell'applicazione (0 se omesso).

Quando viene invocata riceve in input il numero di argomenti, `int argc`, in cui è incluso il nome dell'eseguibile e la lista degli argomenti, `char * argv[]`, come vettore di stringhe (vettore di vettori di caratteri)
Esempio:

```
gcc main.c -o main.o
./main.o arg1 arg2
```

In questo esempio il valore di argc = 3, e argv è della forma "0–./main.o; 1– arg1; 2– arg2"

- **printf/fprintf**

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Inviano dati sul canale stdout (printf) o su quello specificato (fprintf) secondo il formato indicato. Il formato una stringa contenente contenuti stampabili (testo, a capo,) ed eventuali segnaposto identificabili dal formato generale:

```
%[flags][width][.precision][length]specifier
```

Esempio: %d (intero con segno), %c (carattere), %s (stringa), ...
Ad ogni segnaposto deve corrispondere un ulteriore argomento del tipo corretto

- **Direttive**

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

`#include <lib>`	*copia il contenuto del file lib 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*

Macro possono essere passate a GCC con -D NAME=VALUE
Esempio:

```
#include <stdio.h>
#define ITER 5
#define POW(A) A*A
```

```
int main(int argc, char **argv) {
#ifdef DEBUG
      printf("%d\n, argc");
      printf("%s\n", argv[0]);
#endif
      int res = 1;
      for (int i = 0; i < ITER; i++){
            res *= POW(argc);
      }
      return res;
}
gcc main.c -o main.out -D DEBUG=0
gcc main.c -o main.out -D DEBUG=1
./main.out 1 2 3 4
```
Danno lo stesso risultato

```
#include <stdlib.h>
#include <stdio.h>
#define DIVIDENDO 3
int division(int var1, int var2, int * result){
      *result = var1/var2;
      return 0;
}
int main(int argc, char * argv[]){
      float var1 = atof(argv[1]);
      float result = 0;
      division((int)var1,DIVIDENDO,(int *)&result);
      printf("%d \n",(int)result);
      return 0;
}
```

- **Librerie standard**:

  Librerie possono essere usate attraverso la direttiva #include. Tra le pi importanti vi sono:

  - stdio.h: FILE, EOF, stderr, stdout, stdin, ...
  - stdlib.h: atof(),atoi(),...
  - string.h: memset(), memcpy(), strncat(), ...
  - math.h: sqrt(), sin(), cos(),...
  - unistd.h: read(), write(), fork(), ...
  - fcntl.h: creat(), open(), ...

- **Struct e Unions**: Structs permettono di aggregare diverse variabili, mentre le unions permettono di creare dei tipi generici che possono ospitare uno di vari tipi specificati.

Esempio:

```
struct Books {
      char author[50];
      char title[50];
      int bookID;
} book1, book2;
union Result{
      int intero;
      float decimale;
} result1, result2;
```

- **Typedef**: consente la definizione di nuovi tipi di variabili o funzioni
  Esempio:

```
typedef unsigned int intero;
typedef struct Books{
...
} bookType;
intero var = 22; //= unsigned int var = 22;
bookType book1; //= struct Books book1;
```

- **Exit**: `void exit(int status)`
  Il processo è terminato restituendo il valore status come codice di uscita. Si ottiene lo stesso effetto se all'interno della funzione main si ha return status. La funzione non ha un valore di ritorno proprio perchè non sono eseguite ulteriori istruzioni dopo di essa.
  Il processo chiamante è informato della terminazione tramite un segnale apposito.

- **Vettori e Stringhe**

  - **Vettori**:
    I vettori sono sequenze di elementi omogenei (tipicamente liste di dati dello stesso tipo, ad esempio liste di interi o di caratteri), si realizzano con un puntatore al primo elemento della lista.
    Esempio: Con int arr[4] = {2, 0, 2, 1} si dichiara un vettore di 4 interi inizializzandolo: sono riservate 4 aree di memoria consecutive di dimensione pari a quella richiesta per ogni singolo intero (tipicamente 2 bytes, quindi 4*2=8 in tutto)
    Esempio: `char str[7] = {'c', 'i', 'a', 'o', 56,57,0}` : 7*1 = 7 bytes
    str è dunque un puntatore a char (al primo elemento) e si ha che:
    `str[n] corrisponde a *(str+n)`
    e in particolare
    `str[0] corrisponde a *(str+0)=*(str)=*str`
  - **Stringhe**:

Le stringhe in C sono vettori di caratteri, ossia puntatori a sequenze di bytes, la cui terminazione è definita dal valore convenzionale 0 (zero).
Un carattere tra apici singoli equivale all'intero del codice corrispondente.
In particolare un vettore di stringhe è un vettore di vettore di caratteri e dunque:

```
char c; carattere
char * str; vettore di a caratteri o stringa
char **strarr; vettore di vettore di caratteri o vettore
di stringhe
```

Si comprende quindi la segnatura della funzione main con **argv.
C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri.
Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc).
Le stringhe, quando acquisite in input o dichiarate con la sintassi "stringa", terminano con il carattere '\0' e sono dunque di grandezza `str_len+1`
Sebbene ci siano diversi modi per dichiarare ed inizializzare una stringa, questi hanno comportamenti diversi:

```
char string[] = ciao; // writable string in the stack
char * string2 = ciao; // READ-ONLY string
string[2] = a;
string2[2] = a; //Segmentation fault!
```

Dato che le stringhe sono riferite con un puntatore al primo carattere non ha senso fare assegnamenti e confronti diretti, ma si devono usare delle funzioni. La libreria standard string.h ne definisce alcune come ad esempio:

* `char * strcat(char *dest, const char *src)`: aggiunge src in coda a dest
* `char * strchr(const char *str, int c)`: cerca la prima occorrenza di c in str
* `int strcmp(const char *str1, const char *str2)`: confronta str1 con str2
* `size_t strlen(const char *str)`: calcola la lunghezza di str
* `char * strcpy(char *dest, const char *src)`: copia la stringa src in dst
* `char * strncpy(char *dest, const char *src, size_t n)`: copia n caratteri dalla stringa src in dst

1.4 Lab-04: File

File
In unix ci sono due modi per interagre con i file:

1.4.1 Streams

- **Streams**:forniscono strumenti come la formattazione dei dati, bufferizzazione, ecc
  Utilizzando gli streams, un file è descritto da un puntatore a una struttura di tipo FILE (definita in stdio.h). I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volta, ecc.) ed essere interpretati di conseguenza.
  Manipolazione con streams

  `FILE *fopen(const char *filename, const char *mode)`
  Restituisce un FILE pointer per gestire il "filename" nella modalità specificata:

  - r: read
  - w: write
  - r+ read adn write
  - w+: read and write, create or overwrite
  - a: write at the end
  - a+: read and write at the end

  `int fclose(FILE *stream)`: chiusura del file che ritorna un intero con lo stato della chiusura
  Lettura del file:

  - `int fgetc(FILE *stream)`: restituisce un carattere dallo stream
  - `char *fgets(char *str, int n, FILE *strem)`: ritorna una stringa da stream e la salva in str, si ferma quando sono stati letti n-1 caratteri, o viene letto un "\n", oppure raggiunta la fine Inserisce il carattere di terminazione.
  - `int fscanf(FILE *stream, const char *format)`: legge da stream dei dati, salvando ogni dato nelle variabili fornite, seguendo la stringa format
  - `int feof(FILE *strem)`: restituisce 1 se lo stream è arrivato alla fine del file.

  Scrittura del file:

  - `int fputc(int chr, FILE *stream)`: scrive un singolo carattere char su stream

- `int fputs(const char *str, FILE *stream)`: scrive una stringa str su file
- `int fprintf(FILE *stream, const char *format)`:scrive il contenuto di alcune variabili su stream seguendo la stringa format

Flush e Rewind

Seguendo l'immagine, il contenuto di un file viene letto e scritto con degli streams (dei buffer) di dati. Come tali, è comprensibile come queste operazioni non siano immediate: i dati vengono scritti sul buffer e solo successivamente scritti sul file. Il **flush** è l'operazione che trascrive il file dallo stream. Questa operazione avviene quando:

- Il programma termina con un return dal main o con exit().
- `fprintf()` inserisce una nuova riga.
- `int fflush(FILE *stream)` viene invocato.
- `void rewind(FILE *stream)` viene invocato.
- `fclose()` viene invocato

rewind consente inoltre di ripristinare la posizione della testina all'inizio del file.

Esempio 1:

```
#include <stdio.h>
FILE *ptr; //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
     //Read int, word and word
     fscanf(ptr,"%d %s %s", &id, str1, str2);
     printf("%d %s %s\n",id,str1,str2);
}
printf("End of file\n");
fclose(ptr); //Close file
```

Esempio 2:

```
#include <stdio.h>
#define N 10
FILE *ptr;
ptr = fopen("fileToWrite.txt","w+");
fprintf(ptr,"Content to write"); //Write content to file
rewind(ptr); // Reset pointer to begin of file
char chAr[N], inC;
fgets(chAr,N,ptr); // store the next N-1 chars from ptr in
chAr
printf("%d %s",chAr[N-1], chAr);
```

```
do{
      inC = fgetc(ptr); // return next available char or EOF
      printf("%c",inC);
}while(inC != EOF);
printf("\n");
fclose(ptr);
```

1.4.2 File Descriptor

- **File descriptor** interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel. Un file è descritto da un semplice intero (**file descriptor**) che punta alla rispettiva entry nella file table del sistema operativo. I dati possono essere letti e scritti soltanto un buffer alla volta di cui spetta al programmatore stabilire la dimensione.
  Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

  Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel, aprendo il file con la system call open la quale localizza li-node del file e aggiorna la file table del processo.
  A ogni processo è associata una tabella dei file aperti di dimensione limitata, dove ogni elemento della tabella rappresenta un file aperto dal processo ed individuato da un indice intero (il "file descriptor")
  I file descriptor 0, 1 e 2 individuano normalmente standard input (0=stdin), output (1=stdout) ed error (2=stderr) (aperti automaticamente).
  Il kernel gestisce l'accesso ai files attraverso due strutture dati:

  - La **la tabella dei files attivi** che contiene una copia dell'i-node di ogni file aperto (per efficienza)
  - La **tabella dei files aperti** che contiene un elemento per ogni file aperto e non ancora chiuso
    Questo elemento contiene:
    - I/O pointer: posizione corrente nel file
    - i-node pointer: Puntatore a inode corrispondente

    La tabella dei file aperti può avere più elementi corrispondenti allo stesso file

  Apertura e chiusura file
  `int open(const char *pathname, int flags[, mode_t mode]);`

  flags: interi che definiscono l'apertura del file:

  - **O_RDONLY, O_WRONLY, O_RDWR**: almeno uno è obbligatorio

- **O_CREAT, O_EXCL**: crea il file se non esiste, come creat però fallisce se il file esiste già
- **O_APPEND**: apre il file in append
- **O_TRUNC**: cancella il contenuto del file (se usato in scrittura)

mode: interi per i privilegi da assegnare al nuovo file, **S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, S_IRGRP, , S_IROTH**

`int close(int fd)`

Lettura e scrittura file

- `ssize_t read (int fd, void *buf, size_t count);`
  legge dal file e salva nel buffer buf, fino a count bytes di dati dal file associato con il file descriptor fd
- `ssize_t write(int fd, const void *buf, size_t count);`
  scrive sul file associato al file descriptor fd fino a count bytes di dati dal buffer 'buf'
- `off_t lseek(int fd, off_t offset, int whence);`
  riposiziona l'offset del file a seconda dell'argomento offset partendo da una certa posizione whence. Tre posizioni fondamentali: **SEEK_SET** (inizio file), **SEEK_CUR** (posizione attuale), **SEEk_END** (fine file)

Esempio 1:
```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
//Open new file in Read only
int openedFile = open("filename.txt", O_RDONLY);
char content[10];
int canRead;
do{
     //Read 9B to content
     bytesRead = read(openedFile,content,9);
     content[bytesRead]=0;
     printf("%s",content);
} while(bytesRead > 0);
close(openedFile);
```

Esempio 2:
```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
//Open file (create it with user R and W permissions)
int openFile = open("name.txt", O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
```

```
char toWrite[] = "Professor";
write(openFile, "hello world\n", strlen("hello world\n"));
//Write to file
lseek(openFile, 6, SEEK_SET); // move I/O pointer
write(openFile, toWrite, strlen(toWrite)); //Write to file
close(openFile);
```

1.4.3 Canali standard

Canali standard

- I canali standard (in/err/out), sono rappresentati con strutture stream (stdin, stderr, stdout) e macro (STDIN_FILENO, STDERR_FILENO, STDOUT_FILENO)
- La funzione `fileno` restituisce l'indice di uno stream per cui:
  - `fileno(stdin) = STDIN_FILENO` (= 0)
  - `fileno(stdout) = STDOUT_FILENO` (= 1)
  - `fileno(stderr) = STDERR_FILENO` (= 2)
- `isatty(stdin) == 1` con esecuzione interattiva, 0 altrimenti
- `printf("ciao")` ≡ `fprintf(stdout, "ciao")`
- **Esempio**

```
#include <stdio.h>
#include <unistd.h>
void main() {
      printf("stdin:  stdin ->_flags = %hd, STDIN_FILENO =
         %d\n", stdin->_flags, STDIN_FILENO
      );
      printf("stdout:  stdout->_flags = %hd, STDOUT_FILENO =
         %d\n",stdout->_flags, STDOUT_FILENO
      );
      printf("stderr:  stderr->_flags = %hd, STDOUT_FILENO =
         %d\n", stderr->_flags, STDERR_FILENO
      );
}
```

1.4.4 Piping bash

Piping con bash

Normalmente un'applicazione eseguita da ash ha accesso ai canali standard, se le applicazioni sono usate in un'operazione di piping (`ls | wc -l`) allora l'output dell'applicazione sulla sinistra diventa l'input dell'applicazione sulla destra e verranno eseguite **parallelamente**
Esempio:

```
gcc src.c -o pip.out
echo "hi how are you" | ./pip.out
```
src.c
```
#define MAXBUF 10
#include <stdio.h>
#include <string.h>
int main() {
   char buf[MAXBUF];
   fgets(buf, sizeof(buf), stdin); // may truncate!
   printf("%s\n", buf);
   return 0;
}
```

```
ls /tmp | ./inv.out
```
Code:
```
#include <stdio.h>
int main() {
   int c, d;
   // loop into stdin until EOF (as CTRL+D)
   // read from stdin
   while ((c = getchar()) != EOF) {
      d = c;
      if (c >= 'a' && c <= 'z') d -= 32;
      if (c >= 'A' && c <= 'Z') d += 32;
      putchar(d); // write to stdout
   };
   return (0);
}
```

1.5 Lab-05

Architettura

Kernel Unix: il kernel è l'elemento di base di un sistema Unix-like, ovvero il nucleo del sistema operativo. Il kernel è incaricato della gestione delle risorse essenziali (CPU, mem, periferiche)
Ad ogni boot il sistema verifica lo stato delle periferiche, monta la prima partizione (roo file system) in read-only e carica il kernel in memoria.
Il kernel lancia il primo programma (systemd) che a seconda della configurazione voluta, ilizializza il sistema di conseguenza
Il resto delle operazioni, tra cui l'interazione con l'utente, vengono gestite con programmi eseguiti dal kernel
Kernel & Mem Virtuale
I programmi utilizzati dall'utente che vogliono accedere alle periferiche chedono al kernel di farlo per loro
L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione del HW e della MemVirtuale
Ogni programma vede se stesso come **unico processore della CPU** e npn gli è dunque possibile disturbare l'azione degli altri programmi, ⇒ stabilità sistemi Unix
Privilegi
Nei sistemi Unix-like ci sono due livelli di privilegi:

- **User space**: ambiente in cui vengono eseguiti i programmi
- **Kernel space**: ambiente in cui viene eseguito il kernel

1.5.1 System Calls

System calls
Le interfacce con cui i programmi accedono al hw si chiamano **system calls**, cioè chiamate al sistema che il kernel esegue nel **kernel space** retituendo i risultati al programma chiamante nello user space
Le chiamate restituiscono -1 in caso di errore e settano la varaibile globale `errno`. Errori validi sono numeri positivi e seguono lo standard POSIX, che definisce gli alias
Librerie di sistema
Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare le librerie condivise caricate e, fra queste, vi sono tipicamente anche ld-linux.so, e libc.so.

- **ld-linux.so**: quando un programma caricato in memoria, il sistema operativo passa il controllo a ld-linux.so anzich al normale punto di ingresso dellapplicazione. ld-linux trova e carica le librerie richieste, prepara il programma e poi gli passa il controllo.

- **libc.so**: la libreria GNU C solitamente nota come glibc che contiene le funzioni basilari pi comuni.

Esempi di chiamate di sistema

- **time(), ctime()**

```
time_t time( time_t *second )
char * ctime( const time_t *timeSeconds )
```

```
#include <time.h>
#include <stdio.h>
void main(){
   time_t theTime;
   time_t whatTime = time(&theTime); //seconds since 1/1/1970
   //Print date in Www Mmm dd hh:mm:ss yyyy
   printf("Current time = %s= %d\n",
           ctime(&whatTime),theTime);
}
```

- **chdir(), getcwd()**

```
int chdir( const char *path );
char * getcwd( char *buf, size_t sizeBuf );
```

```
#include <unistd.h>
#include <stdio.h>
void main(){
   char s[100];
   getcwd(s,100); // copy path in buffer
   printf("%s\n", s); //Print current working dir
   chdir(".."); //Change working dir
   printf("%s\n", getcwd(NULL,100)); // Allocates buffer
}
```

- **dup(), dup2()**

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

```
#include <unistd.h> <stdio.h> <fcntl.h>
int main(void){
   char buf[51];
   int fd = open("file.txt",O_RDWR); //file exists
   int r = read(fd,buf,50); //Read 50 bytes from fd in buf
   buf[r] = 0; printf("Content:  %s\n",buf);
   int cpy = dup(fd); // Create copy of file descriptor
   // Copy cpy to descriptor 22 (close 22 if opened)
   dup2(cpy,22);
   // Move I/O on all 3 file descriptors!
   lseek(cpy,0,SEEK_SET);
```

```
   // Write starting from 0-pos
   write(22,"This is a fine\n",16);
   close(cpy); //Close ONE file descriptor }
```

- **chmod(), chown()**

```
int chown(const char *pathname, uid_t owner, gid_t group)
int fchown(int fd, uid_t owner, gid_t group)
int chmod(const char *pathname, mode_t mode)
int fchmod(int fd, mode_t mode)
```

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
void main(){
   int fd = open("file",O_RDONLY);
   // Change owner to user:group 1000:1000
   fchown(fd, 1000, 1000);
   // Permission to r/r/r
   chmod("file",S_IRUSR|S_IRGRP|S_IROTH);
}
```

1.5.2 Exec()

Exec() family
La famiglia di funzioni exec ha come scopo finale l'esecuzione di un programma, sostituendo l'immagine del processo corrente con una nuova immagine. **Il PID del processo e la sua file table non cambiano**.
La chiamata di sistema di base è execve(), ma vedremo tutti i suoi alias che differiscono solo per gli argomenti accettati, mantenendo lo stesso identico comportamento.
Ogni alias è composto dalla parola chiave exec seguita dalle seguenti lettere:

- l: accetta una lista di argomenti
- v: accetta un vettore, quindi un solo argomento di diversi argomenti
- p: usa la variabile d'ambiente PATH per cercare il binario
- e: usa un vettore di variabili d'ambiente (es. "name=value")

NB: Ogni vettori di argomenti deve terminare con un elemento NULL

Chiamte funzioni:

- `int execv (const char *path, char *const argv[])`
- `int execvp (const char *file, char *const argv[])`
- `int execve (const char *path, char *const argv[], char *const envp[]);`

- `int execvpe (const char *file, char *const argv[], char *const envp[])`
- `int execl (const char *path, const char * arg0,,argn,NULL)`
- `int execlp (const char *file, const char * arg0,,argn,NULL)`
- `int execle (const char *path, const char * arg0,,argn,NULL, char *const envp[])`
- `int execlpe(const char *file, const char * arg0,,argn,NULL, char *const envp[])`

Esempi:

- **execv()**

  execv1.out

```
#include <unistd.h>
#include <stdio.h>
void main(){
   char * argv[] = {"par1","par2",NULL};
   execv("./execv2.out",argv); //Replace current process
   printf("This is execv1\n");
}
```

  execv2.out

```
#include <stdio.h>
void main(int argc, char ** argv) {
   printf("This is execv2 with %s and %s\n",argv[0],argv[1]);
}
```

- **execle()**

  execle1.out

```
#include <unistd.h>
#include <stdio.h>
void main(){
   char * env[] = "CIAO=hello world",NULL;
   //Replace proc.
   execle("./execle2.out","par1","par2",NULL,env);
   printf("This is execle1\n");
}
```

  execle2.out

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char ** argv){
   printf("This is execle2 with par:%s and %s.
      CIAO = %s\n",argv[0],argv[1],getenv("CIAO"));
}
```

- **dup2/exec**

  execvDuo.o

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
void main() {
   int outfile = open("/tmp/out.txt", O_RDWR | O_CREAT,
                      S_IRUSR | S_IWUSR);
   dup2(outfile, 1); // copy outfile to FD 1
   char *argv[]={"./time.out",NULL}; // time.out esempi prec
   execvp(argv[0],argv); // Replace current process
}
```

system()

```
int system(const char * str);
#include <stdlib.h> <stdio.h>
#include <sys/wait.h> /* For WEXITSTATUS */
void main(){
   int outcome = system("echo ciao"); // execute command in shell
   printf("Outcome = %d\n",outcome);
   outcome = system("if [[ $PWD < c̈iaoT̈]]; then echo min; fi");
   printf("Outcome = %d\n",outcome);
   outcome = system("notExistingCommand");
   printf("Outcome = %d\n",WEXITSTATUS(outcome));
}
```

1.5.3 Fork()

Forking

Il forking è la "generazione" di nuovi processi (uno alla volta) partendo da uno esistente. La syscall principale per il forking è `fork()`.
Quando un processo attivo invoca questa syscall, il kernel lo clona modificando però alcune informazioni, in particolare quelle che riguardano la sua collocazione nella gerarchia complessiva dei processi.
Il processo che effettua la chiamata è definito "padre/genitore", quello generato è definito "figlio".

Identificativi dei processi

Ad ogni processo è associato un identificativo univoco per istante temporale, sono organizzati gerarchicamente (genitore-figlio) e suddivisi in insiemi principali (sessioni) e secondari (gruppi). Anche gli utenti hanno un loro identificativo e ad ogni processo ne sono abbinati due: quello reale e quello effettivo (di esecuzione)

- PID - Process ID
- PPID - Parent Process ID

- SID - Session ID
- PGID - Process Group ID
- UID/RUID - (Real) User ID
- EUID - Effective User ID

fork: elementi clonati e elementi nuovi
Sono clonati gli elementi principali come il PC (Program Counter), i registri, la tabella dei file (file descriptors) e i dati di processo (variabili). Le meta-informazioni come il "pid" e il "ppid" sono aggiornate (non come in execve()).
L'esecuzione procede per entrambi (quando saranno schedulati) da PC+1 (tipicamente l'istruzione seguente il fork o la valutazione dell'espressione in cui essa utilizzata):

- `fork();`
  `printf("\n");`
  Prossimo step: printf
- `f = fork();`
  `printf("\n")`
  Prossimo step: assegnamento ad f

getpid(), getppid()

- `pid_t getpid()` : restituisce il PID del processo attivo
- `pid_t getppid()` : restituisce il PID del processo genitore

Esempio:
```
#include <stdio.h> <unistd.h> <stdlib.h>
void main(){
   printf("Subshell $$ = ");
   fflush(stdout); // Forza l'output di printf
   system("echo $$"); // subshell
   printf("PID: %dPPID: %d\n",getpid(),getppid());
```
} N.B.: includendo le librerie `<sys/types.h>` e `sys/wait.h` il tipo `pid_t` è un intero che rappresenta un id del processo
fork: valore di ritorno
La funzione restituisce un valore che solitamente è catturato in una variabile (o usato comunque in un'espressione).
Come per tutte le syscall in generale, il valore -1 in caso di errore (in questo caso non ci sarà nessun nuovo processo, ma solo quello che ha invocato la chiamata).
Se ha successo entrambi i processi ricevono un valore di ritorno, ma questo è diverso nei due casi:

- Il processo genitore riceve come valore il nuovo PID del processo figlio

- Il processo figlio riceve come valore 0

fork: relazione tra i processi
I processi genitore-figlio:

- Conoscono reciprocamente il loro PID (ciascuno conosce il proprio tramite getpid(), il figlio conosce quello del genitore con getppid(), il genitore conosce quello del figlio come valore di ritorno di fork())
- Si possono usare altre syscall per semplici interazioni come wait e waitpid
- Eventuali variabili definite prima del fork sono valorizzate allo stesso modo in entrambi: se riferiscono risorse (ad esempio un file descriptor per un file su disco) fanno riferimento esattamente alla stessa risorsa

fork: wait()
`pid_t wait (int *status)`

Attende il cambio di stato di un processo figlio (uno qualsiasi) restituendone il PID e salvando in status lo stato del figlio (se il puntatore non è NULL). Il cambio di stato avviene se il figlio viene terminato o la sua esecuzione interrotta/ripresa a seguito di un segnale
Se non esiste alcun figlio restituisce -1.
Nel nostro caso ci interessa principalmente la terminazione del figlio. Questa system call ci permette di bloccare il processo (anche sincronizzarlo) fino a quando il figlio non ha finito le sue operazioni.
`while(wait(NULL)>0);` questo comando aspetta tutti i figli
Interpretazione stato
Lo stato di ritorno è un numero che comprende pi valori "composti" interpretabili con apposite macro, molte utilizzabili a mo' di funzione (altre come valore) passando lo "stato" ricevuto come risposta come ad esempio:

- `WEXITSTATUS(sts)`: restituisce lo stato vero e proprio (ad esempio il valore usato nella exit).
- `WIFCONTINUED(sts)`: true se il figlio ha ricevuto un segnale SIGCONT.
- `WIFEXITED(sts)`: true se il figlio terminato normalmente.
- `WIFSIGNALED(sts)`: true se il figlio terminato a causa di un segnale non gestito.
- `WIFSTOPPED(sts)`: true se il figlio attualmente in stato di stop.
- `WSTOPSIG(sts)`: numero del segnale che ha causato lo stop del figlio.
- `WTERMSIG(sts)`: numero del segnale che ha causato la terminazione del figlio.

Esempio `#include <stdio.h> <unistd.h> <sys/wait.h>`

```
int main(void) {
   int isChild = !fork();
   if(isChild) {
      sleep(3); return 5;
   }
   int childStatus;    wait(&childStatus);
   printf("Children terminated?  %d\nReturn code:  %d\n",
      WIFEXITED(childStatus),WEXITSTATUS(childStatus));
   return 0; }
```

fork: waitpid()

`pid_t waitpid(pid_t pid, int *status, int options)`

Consente un'attesa selettiva basata su dei parametri. **pid** puè essere:

- -n (aspetta un figlio qualsiasi nel gruppo $|-n|$)
- -1 (aspetta un figlio qualsiasi)
- 0 (aspetta un figlio qualsiasi appartenente allo stesso gruppo)
- n (aspetta il figlio con PID=n)

options sono i seguenti parametri ORed:

- WNOHANG: ritorna immediatamente se nessun figlio è terminato allora non si resta in attesa
- WUNTRACED: ritorna anche se un figlio si è interrotto senza terminare.
- WCONTINUED: ritorna anche se un figlio ha ripreso l'esecuzione.

`wait(st)` è l'equivalente di `waitpid(-1, st, 0)`

fork multiplo: è possibile effettuare più fork all'interno dello stesso programma

Esempio fork & wait

```
#include <stdio.h> <stdlib.h> <unistd.h> <time.h> <sys/wait.h>
int main() {
   int fid=fork(), wid, st, r; // Generate child
   srand(time(NULL)); // Initialise random
   r=rand()%256; // Get random between 0 and 255
   if (fid==0) { //If it is child
      printf("Child...  (%d)", r); fflush(stdout);
      sleep(3); // Pause execution for 3 seconds
      printf(" done!\n");
      exit(r); // Terminate with random signal
   } else { // If it is parent
```

```
      printf("Parent...\n");
      wid=wait(&st); // wait for ONE child to terminate
      printf("...child's id:  %d==%d (st=%d)\n", fid, wid,
            WEXITSTATUS(st));
   }
}
```

Processi zombie e orfani
Normalmente quando un processo termina il suo stato di uscita viene "catturato" dal genitore: alla terminazione il sistema tiene traccia di questo insieme di informazioni (lo stato) fino a che il genitore le utilizza consumandole (con wait o waitpid).
Se il genitore non cattura lo stato d'uscita, i suoi processi figli vengono definiti **zombie** (in realtè non ci sono più , ma esiste un riferimento in sospeso nel sistema).
Se un genitore termina prima del figlio, quest'ultimo viene definito **orfano** e viene adottato dal processo principale (tipicamente systemd con pid pari a 1).
Un processo zombie che diventa anche orfano è poi gestito dal processo che lo adotta (che effettua periodicamente dei wait/waitpid appositamente).
Per ispezionare la lista di processi attivi usare il comando `ps` con le seguenti opzioni:

- a: mostra lo stato (T: stopped, Z: zombie, R: running, etc)
- -H: mostra la gerarchia processi
- -e: mostra l'intera lista dei processi, non solo della sessione corrente
- -f: mostra il PID del padre

1.6 Lab-06: Segnali

Segnali
Ci sono vari eventi che possono avvenire in maniera asincrona al normale flusso di un programma, alcuni dei quali in maniera inaspettata e non predicibile. Per esempio, durante l'esecuzione di un programma ci può essere una richiesta di terminazione o di sospensione da parte di un utente, la terminazione di un processo figlio o un errore generico.
Unix prevede la gestione di questi eventi attraverso i **segnali**: quando il sistema operativo si accorge di un certo evento, genera un segnale da mandare al processo interessato il quale potrà decidere (nella maggior parte dei casi) come comportarsi.
Il numero dei segnali disponibili cambia a seconda del sistema operativo, con Linux che ne definisce 32. Ad ogni segnale corrisponde sia un valore numerico che unetichetta mnemonica (definita nella libraria "signal.h") nel formato `SIGXXX`

$SIGALRM$(*alarm clock*)	$SIGQUIT$(*terminal quit*)
$SIGCHLD$(*child terminated*)	$SIGSTOP$(*stop*)
$SIGCONT$(*continue, if stopped*)	$SIGTERM$(*termination*)
$SIGINT$(*terminal interrupt/CTRL + C*)	$SIGUSR1$(*user signal*)
$SIGKILL$(*kill process*)	$SIGUSR2$(*user signal*)

Per ogni processo, all'interno della process table, vengono mantenute due liste:

- **Pending signals**: segnali emessi dal kernel e che il processo deve ancora gestire.
- **Blocked signals**: segnali che non devono essere comunicati al processo. Chiamata anche con il termine **signal mask**, maschera dei segnali.

Ad ogni schedulazione del processo le due liste vengono controllate per consentire al processo di reagire nella maniera più adeguata

Gestione
I segnali sono anche detti "software interrupts" perchè sono, a tutti gli effetti, delle interruzioni del normale flusso del processo generate dal sistema operativo (invece che dall'hardware, come per gli hardware interrupts).
Come per gli interrupts, il programma può decidere come gestire l'arrivo di un segnale (presente nella lista pending):

- Eseguendo l'azione default
- Ignorandolo (non sempre possibile) $\Rightarrow$ programma prosegue normalmente.
- Eseguendo un handler personalizzato $\Rightarrow$ programma si interrompe.

NB: nella pratica, il programma comunica al kernel come vuole che il segnale venga gestito, ed è poi il kernel che richiamerà la funzione adeguata del programma

Default handler
Ogni segnale ha un suo handler di default che tipicamente può :

- Ignorare il segnale
- Terminare il processo
- Continuare l'esecuzione (se il processo era in stop)
- Stoppare il processo

Ogni processo può sostituire il gestore di default con una funzione "custom" (a parte per SIGKILL e SIGSTOP) e comportarsi di conseguenza. La sostituzione avviene tramite la system call signal() (definita in "signal.h").

signal()
`sighandler_t signal(int signum, sighandler_t handler);`
Imposta un nuovo signal handler handler per il segnale signum. Restituisce il signal handler precedente. Quello nuovo pu essere:

- SIG_DFL: handler di default
- SIG_IGN: ignora il segnale
- `typedef void (*sighandler_t)(int)`: custom handler

Esempio:
```
#include <signal.h> <stdio.h> <stdlib.h>
void main(){
   signal(SIGINT,SIG_IGN); //Ignore signal
   signal(SIGCHLD,SIG_DFL); //Use default handler
}
```

1.6.1 Handler custom

Custom handler
Un custom handler **deve** essere una funzione di tipo void che accetta come argomento un int rappresentante il segnale catturato. Questo consente allo stesso handlers di gestire segnali diversi.
`void myHandler(int par)`
Esempio:
```
#include <signal.h> <stdio.h>
void myHandler(int sigNum){
   if(sigNum == SIGINT) printf("CTRL+C\n");
   else if(sigNum == SIGTSTP) printf("CTRL+Z\n");
```

```
}
signal(SIGINT,myHandler);
signal(SIGTSTP,myHandler);
```

signal() return
signal() restituisce un riferimento allhandler che era precedentemente assegnato al segnale:

- NULL: handler precedente era l'handler di default
- 1: l'handler precedente era SIG_IGN
- $< address >$: l'handler precedente era *(address)

Esempio:
```
#include <signal.h> <stdio.h>
void myHandler(int sigNum){}
int main(){
   printf("DFL: %p\n",signal(SIGINT,SIG_IGN));
   printf("IGN: %p\n",signal(SIGINT,myHandler));
   printf("Custom:%p == %p\n",signal(SIGINT,SIG_DFL),myHandler);
}
```

SIGXXX	description	default
SIGALRM	(alarm clock)	quit
SIGCHLD	(child terminated)	ignore
SIGCONT	(continue, if stopped)	ignore
SIGINT	(terminal interrupt, CTRL + C)	quit
SIGKILL	(kill process)	quit
SIGSYS	(bad argument to syscall)	quit with dump
SIGTERM	(software termination)	quit
SIGUSR1/2	(user signal 1/2)	quit
SIGSTOP	(stopped)	quit
SIGTSTP	(terminal stop, CTRL + Z)	quit

Esempio
```
#include <signal.h> <stdio.h> <unistd.h> <sys/wait.h>
void myHandler(int sigNum){
   printf("Child terminated Received %d\n",sigNum);
}
int main(){
   signal(SIGCHLD,myHandler);
   int child = fork();
   if(!child){
      return 0; //terminate child
   }
   while(wait(NULL)>0);
}
```

Inviare i segnali: kill()
`int kill(pid_t pid, int sig);`
`$ kill -`$< signo > < pid\_t >$
Invia un segnale ad uno o pi processi a seconda dell'argomento pid:

- pid > 0: segnale al processo con PID=pid
- pid $= 0$: segnale ad ogni processo dello stesso gruppo
- pid $=$ -1: segnale ad ogni processo possibile (stesso UID/RUID)
- pid $<$ -1: segnale ad ogni processo del gruppo |pid|

Restituisce 0 se il segnale viene inviato, -1 in caso di errore.
Ogni tipo di segnale pu essere inviato, non deve essere necessariamente un segnale corrispondente ad un evento effettivamente avvenuto
Esempio

```
#include <signal.h> <stdio.h> <stdlib.h> <sys/wait.h> <unistd.h>
void myHandler(int sigNum){
   printf("[%d]ALARM!\n",getpid());
}
int main(void){
   signal(SIGALRM,myHandler);
   int child = fork();
   if (!child) while(1); // block the child
   printf("[%d]sending alarm to %d in 3 s\n",getpid(),child);
   sleep(3);
   kill(child,SIGALRM); // send ALARM, child's handler reacts
   printf("[%d]sending SIGTERM to %d in 3 s\n",getpid(),child);
   sleep(3);
   kill(child,SIGTERM); // send TERM: default is to terminate
   while(wait(NULL)>0);
}
```

kill() da bash
kill è anche un programma in bash che accetta come primo argomento il tipo di segnale (**kill -l** per la lista) e come secondo argomento il PID del processo
Esempio

```
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>
void myHandler(int sigNum){
   printf("[%d]ALARM!\n",getpid());
   exit(0);
}
int main(){
   signal(SIGALRM,myHandler);
   printf("I am %d\n",getpid());
   while(1);
}
```

```
gcc bash.c -o bash.out
$ ./bash.out
# On new window/terminal
$ kill -14 <PID>
```

alarm()

`unsigned int alarm(unsigned int seconds);`

Genera un segnale SIGALRM per il processo corrente dopo un lasso di tempo specificato in secondi. Restituisce i secondi rimanenti all'alarm precedente.

Esempio

```
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>
short cnt = 0;
void myHandler(int sigNum){printf("ALARM!\n"); cnt++;}
int main(){
   signal(SIGALRM,myHandler);
   alarm(0); //Clear any pending alarm
   alarm(5); //Set alarm in 5 seconds
   //Set new alarm (cancelling previous one)
   printf("Seconds remaining to previous alarm %d\n",alarm(2));
   while(cnt<1);
}
```

pause()

`int pause();`

Esempio:

```
#include <signal.h> <unistd.h> <stdio.h>
void myHandler(int sigNum){
   printf("Continue!\n");
}
int main(){
   signal(SIGCONT,myHandler);
   signal(SIGUSR1,myHandler);
   pause();
}
```

Bloccare segnali

Oltre alla lista dei "pending signal" esiste la **lista dei "blocked signals"**, ovvero dei segnali ricevuti dal processo ma volutamente non gestiti. Mentre i segnali ignorati non saranno mai gestiti, i segnali bloccati sono solo **temporaneamente** non gestiti.

Al contrario dei segnali ignorati, un segnale bloccato rimane nello stato pending fino a quando esso non viene gestito oppure il suo handler tramutato in ignore.

Per allertare il kernel dei segnali che devono essere bloccati, un processo può **modificare la propria signal mask**, ovvero una struttura dati mantenuta nel kernel che pu essere alterata con la funzione `sigprocmask()`

Bloccare i segnali: sigset_t
La signal mask viene memorizzata come struttura dati ottimizzata che non può essere modificata direttamente. Invece, può essere **gestita attraverso un sigset_t**, cioè una struttura dati locale contenente un elenco di segnali. Questo insieme può essere modificato con funzioni dedicate e poi può essere utilizzato per modificare la maschera di segnale stessa.

- `int sigemptyset(sigset_t *set); Svuota`
- `int sigfillset(sigset_t *set); Riempie`
- `int sigaddset(sigset_t *set, int signo); Aggiunge singolo`
- `int sigdelset(sigset_t *set, int signo); Rimuove singolo`
- `int sigismember(const sigset_t *set, int signo); Interpella`

NB: la modifica di questa struttura non modifica implicitamente la maschera dei segnali! Le modifiche devono essere salvate con `sigprocmask()`.

sigprocmask()
```
int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oldset);
```
A seconda del valore di **how** e di **set**, la maschera dei segnali del processo viene cambiata. Nello specifico:

- how = SIG_BLOCK: i segnali in set sono aggiunti alla maschera;
- how = SIG_UNBLOCK: i segnali in set sono rimossi dalla maschera;
- how = SIG_SETMASK: set diventa la maschera.

Se oldset non è nullo, in esso verrà salvata la vecchia maschera (anche se set è nullo).

Esempio 1:
```
#include <signal.h>
int main(){
   sigset_t mod,old;
   sigfillset(&mod); // Add all signals to the blocked list
   sigemptyset(&mod); // Remove all signals from blocked list
   sigaddset(&mod,SIGALRM); // Add SIGALRM to blocked list
   sigismember(&mod,SIGALRM); // is SIGALRM in blocked list?
   sigdelset(&mod,SIGALRM); // Remove SIGALRM from blocked list
   // Update the current mask with the signals in mod
   sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Esempio 2:

```
#include <signal.h> <unistd.h> <stdio.h>
sigset_t mod, old;
int i = 0;
void myHandler(int signo){
   printf("signal received\n"); i++; }
int main(){
   printf("my id = %d\n",getpid());
   signal(SIGUSR1,myHandler);
   sigemptyset(&mod); //Initialise set
   sigaddset(&mod,SIGUSR1);
   while(1) if(i==1) sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Verificare pending signals: sigpending()

```
int sigpending(sigset_t *set);
```

Esempio:

```
#include <signal.h> <unistd.h> <stdio.h> <stdlib.h>
sigset_t mod,pen;
void handler(int signo){
   printf("SIGUSR1 received\n");
   sigpending(&pen);
   if(!sigismember(&pen,SIGUSR1))
      printf("SIGUSR1 not pending\n");
   exit(0);
}
int main(){
   signal(SIGUSR1,handler);
   sigemptyset(&mod);
   sigaddset(&mod,SIGUSR1);
   sigprocmask(SIG_BLOCK,&mod,NULL);
   kill(getpid(),SIGUSR1);
   // sent but its blocked...
   sigpending(&pen);
   if(sigismember(&pen,SIGUSR1))
      printf("SIGUSR1 pending\n");
   sigprocmask(SIG_UNBLOCK,&mod,NULL);
   while(1);
}
```

sigaction() system call

```
int sigaction(int signum, const struct sigaction *restrict act,
struct sigaction *restrict oldact);

struct sigaction {
   void (*sa_handler)(int);
```

```
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask; //Signals blocked during handler
   int sa_flags; //modify behaviour of signal
   void (*sa_restorer)(void); //Deprecated, not POSIX
};
```

Esempio:

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>
void handler(int signo){
   printf("signal received\n");
}
int main(){
   struct sigaction sa; //Define sigaction struct
   sa.sa_handler = handler; //Assign handler to struct field
   sigemptyset(&sa.sa_mask); //Define an empty mask
   sigaction(SIGUSR1,&sa,NULL);
   kill(getpid(),SIGUSR1);
}
```

Inviare un payload con un segnale

`int sigqueue(pid_t pid, int sig, const union sigval value);`
Invia un segnale sig al processo identificato da pid, accompagnato da un payload value. Quest'ultimo è rappresentato con la seguente union:

```
union sigval {
   int sival_int;
   void *sival_ptr;
};
```

Essa pu contenere un puntatore qualsiasi o un intero, che verrà ricevuto solo ed esclusivamente se il processo destinatario utilizza sigaction con SA_SIGINFO.

Ricezione:

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>
void handler(int signo, siginfo_t * info, void * empty){
   printf("Received integer\n",info->si_value.sival_int);
}
int main(){
   struct sigaction sa;
   sa.sa_sigaction = handler;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags |= SA_SIGINFO; // Use sa_sigaction
   sigaction(SIGUSR1,&sa,NULL);
   while(1);
}
```

Invio:

```
int main(int argc, char ** argv){
   union sigval value;
   value.sival_int = atoi(argv[2]);
   sigqueue(atoi(argv[1],SIGUSR1,value));
   while(1);
}
```

1.7 Lab-07

Errori in C
Durante l'esecuzione di un programma ci possono essere diversi tipi di errori: system calls che falliscono, divisioni per zero, problemi di memoria ...
Alcuni di questi errori non fatali, come una system call che fallisce, possono essere indagati attraverso la variabile `errno`. Questa variabile globale contiene lultimo codice di errore generato dal sistema.
Per convertire il codice di errore in una stringa comprensibile si può usare la funzione `char *strerror(int errnum)`.
In alternativa, la funzione `void perror(const char *str)` che stampa su **stderr** la stringa passatagli come argomento concatenata, tramite ': ', con strerror(errno).

Esempio:
```
#include <stdio.h> <errno.h> <string.h>
extern int errno; // declare external global variable
int main(void){
   FILE * pf;
   pf = fopen ("nonExistingFile.boh", "rb"); //Try to open file
   if (pf == NULL) { //something went wrong!
      fprintf(stderr, "errno = %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr,"Strerror:  %s\n", strerror(errno));
   } else {
      fclose (pf);
   }
}
```

Esempio:
```
#include <stdio.h> <errno.h> <string.h> <signal.h>
extern int errno; // declare external global variable
int main(void){
   int sys = kill(3443,SIGUSR1); //Send signal to non exist proc
   if (sys == -1) { //something went wrong!
      fprintf(stderr, "errno = %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr,"Strerror:  %s\n", strerror(errno));
   } else {
      printf("Signal sent\n");
   }
}
```

1.7.1 Gruppi

Process groups

All'interno di Unix i processi vengono raggruppati secondi vari criteri, dando vita a sessioni, gruppi e threads
Perchè i gruppi: I process groups consentono una migliore gestione dei segnali e della comunicazione tra i processi.
Un processo, per l'appunto, può :

- Aspettare che tutti i processi figli appartenenti ad un determinato gruppo terminino;

  ```
  waitpid(-33,NULL,0); // Wait for a children in group 33
  ```

- Mandare un segnale a tutti i processi appartenenti ad un determinato gruppo
  ```
  kill(-45,SIGTERM); // Send SIGTERM to all children in grp 45
  ```

L'organizzazione dei processi in gruppi consente di organizzare meglio le comunicazione e di coordinare le operazioni avendo in particolare la possibilità di inviare dei segnali ai gruppi nel loro complesso.

Gruppi in Unix
Mentre, generalmente, una sessione è collegata ad un terminale, i processi vengono raggruppati nel seguente modo:

- In bash, processi concatenati tramite pipes appartengono allo stesso gruppo:
  ```
  cat /tmp/ciao.txt | wc -l | grep '2'
  ```

- Alla loro creazione, i figli di un processo ereditano il gruppo del padre

- Inizialmente, tutti i processi appartengono al gruppo di 'init', ed ogni processo può cambiare il suo gruppo in qualunque momento.

Il processo il cui PID è uguale al proprio **GID** è detto **process group leader**

Group system calls

```
int setpgid(pid_t pid, pid_t pgid); //set GID of proc.  (0=self)
pid_t getpgid(pid_t pid); // get GID of process (0=self)
```

Esempio:
```
#include <stdio.h> <unistd.h> <sys/wait.h>
int main(void){
   int isChild = !fork(); //new child
   printf("PID %d PPID: %d GID %d\n",
           getpid(),getppid(),getpgid(0));
   if(isChild){
   isChild = !fork(); //new child
   if(!isChild) setpgid(0,0); // Become group leader
      sleep(1);
      fork(); //new child
```

```
      printf("PID %d PPID: %d GID %d\n",
              getpid(),getppid(),getpgid(0));
   };
   while(wait(NULL)>0);
}
```

1.7.2 Segnnali ai gruppi

Mandare segnali ai gruppi
Nel prossimo esempio:

1. Processo ancestor crea un figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)
   (b) I 4 processi aspettano fino all'arrivo di un segnale
2. Processo ancestor crea un secondo figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)
   (b) I 4 processi aspettano fino allarrivo di un segnale
3. Processo ancestor manda due segnali diversi ai due gruppi

```
#include <stdio.h><unistd.h><sys/wait.h><signal.h><stdlib.h>
void handler(int signo){
   printf("[%d,%d] sig%d received\n",getpid(),getpgid(0),signo);
   sleep(1); exit(0);
}
int main(void){
   signal(SIGUSR1,handler);
   signal(SIGUSR2,handler);
   int ancestor = getpid(); int group1 = fork(); int group2;
   if(getpid()!=ancestor ){ // First child
      setpgid(0,getpid()); // Become group leader
      fork();fork(); //Generated 3 children in new group
   }else {
      group2 = fork();
      if(getpid()!=ancestor){ // Second child
         setpgid(0,getpid()); // Become group leader
         fork();fork();
      }
   } //Generated 3 children in new group
   if(getpid()==ancestor){
      printf("[%d]Ancestor and I'll send signals\n",getpid());
      sleep(1);
      kill(-group2,SIGUSR2); //Send SIGUSR2 to group2
      kill(-group1,SIGUSR1); //Send SIGUSR1 to group1
```

```
   }else{
      printf("[%d,%d]chld waiting signal\n", getpid(),getpgid(0));
      while(1);
   }
   while(wait(NULL)>0);
   printf("All children terminated\n");
}
```

Wait figli in un gruppo
Nel prossimo esempio:

1. 1. Processo ancestor crea un figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo1)
   (b) I 4 processi aspettano 2 secondi e terminano
2. Processo ancestor crea un secondo figlio
   (a) Il figlio cambia il proprio gruppo e genera 3 figli (Gruppo2)
   (b) I 4 processi aspettano 4 secondi e terminano
3. Processo ancestor aspetta la terminazione dei figli del gruppo1
4. Processo ancestor aspetta la terminazione dei figli del gruppo2

```
#include <stdio.h><unistd.h><sys/wait.h>
int main(void){
   int group1 = fork();
   int group2;
   if(group1 == 0){ // First child
      setpgid(0,getpid()); // Become group leader
      fork(); fork(); //Generated 4 children in new group
      sleep(2); return; //Wait 2 sec and exit
   }else{
      group2 = fork();
      if(group2 == 0){
         setpgid(0,getpid()); // Become group leader
         fork(); fork(); //Generated 4 children
         sleep(4); return; //Wait 4 sec and exit
      }
   }
   sleep(1); //make sure the children changed their group
   while(waitpid(-group1,NULL,0)>0);
   printf("Children in %d terminated\n",group1);
   while(waitpid(-group2,NULL,0)>0);
   printf("Children in %d terminated\n",group2);
}
```

1.8 Lab-08: Pipe

1.8.1 Anonime

Pipe anonime

Piping
Il piping connette l'output (stdout e stderr) di un comando all'input (stdin) di un altro comando, consentendo dunque la comunicazione tra i due.

Esempio:
`ls . | sort -R #stout → stdin`
`ls nonExistingDir |& wc #stdout e stderr → stdin`
`cat /etc/passwd | wc | less #out → in, out→ in`

I processi sono eseguiti in concorrenza utilizzando un buffer:

- Se pieno lo scrittore (left) si sospende fino ad avere spazio libero
- Se vuoto il lettore si sospende fino ad avere i dati

Esempio:

```
./output.out | ./input.out
```
output.out
```
#include <stdio.h> <unistd.h>
int main(){
   for (int i = 0; i<3; i++) {
      sleep(2);
      fprintf(stdout,"Written in buffer");
      fflush(stdout);
   };
};
```
input.out
```
#include <stdio.h> <unistd.h>
int main() {
   char msg[50]; int n=3;
   while((n--)>0){
      int c = read(0,msg,49);
      if (c>0) {
         msg[c]=0;
         fprintf(stdout,"Read:  '%s' (%d)\n",msg,c);
      };
   };
};
```

Le pipe anonime, come quelle usate su shell, 'uniscono' due processi aventi un antenato comune (oppure tra padre-figlio). Il collegamento è **unidirezionale** ed avviene utilizzando un buffer di dimensione finita.

Per interagire con il buffer (la pipe) si usano **due** file descriptors: uno per il lato in scrittura ed uno per il lato in lettura. Visto che i processi figli ereditano i file descriptors, questo consente la comunicazione tra i processi.

Creazione pipe

```
int pipe(int pipefd[2]); //fd[0] lettura, fd[1] scrittura
```

```
#include <stdio.h> <unistd.h>
int main(){
   int fd[2], cnt = 0;
   while(pipe(fd) == 0){ //Create unnamed pipe using 2 FD
      cnt++;
      printf("%d,%d,",fd[0],fd[1]);
   }
   printf("\nOpened %d pipes, then error\n",cnt);
   int op = open("/tmp/tmp.txt",O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
   printf("File opened with fd %d\n",op);
}
```

Lettura pipe

```
int read(int fd[0],char * data, int num)
```

La lettura della pipe tramite il comando read restituisce valori differenti a seconda della situazione:

- In caso di successo, read() restituisce il numero di bytes effettivamente letti
- Se il lato di scrittura è stato chiuso (da ogni processo), con il buffer vuoto restituisce 0, altrimenti restituisce il numero di bytes letti.
- Se il buffer è vuoto ma il lato di scrittura è ancora aperto (in qualche processo) il processo si sospende fino alla disponibilit dei dati o alla chiusura
- Se si provano a leggere più bytes (num) di quelli disponibili, vengono recuperati solo quelli presenti

Esempio:

```
#include <stdio.h> <unistd.h>
int main(void){
   int fd[2]; char buf[50];
   int esito = pipe(fd); //Create unnamed pipe
   if(esito == 0){
      write(fd[1],"writing",8); // Writes to pipe
      int r = read(fd[0],&buf,50); //Read from pipe
      printf("Last read %d.  Received:  '%s'\n",r,buf);
      // close(fd[1]); // hangs when commented
      r = read(fd[0],&buf,50); //Read from pipe
```

```
      printf("Last read %d.  Received:  '%s'\n",r,buf);
   }
}
```

Scrittura pipe

```
int write(int fd[1],char * data, int num)
```

La scrittura della pipe tramite il comando write restituisce il numero di bytes scritti. Tuttavia, se il lato in lettura è stato chiuso viene inviato un segnale `SIGPIPE` allo scrittore (default handler quit) e la chiamata restituisce -1.
In caso di scrittura, se vengono scritti **meno** bytes di quelli che ci possono stare (`PIPE_BUF`) la scrittura è "atomica" (tutto assieme), in caso contrario non c'è garanzia di atomicità e la scrittura sarà bloccata (in attesa che il buffer venga svuotato) o fallirà se il flag `O_NONBLOCK` viene usato.
Per modificare le proprietà di una pipe, possiamo usare la system call **fnctl**, la quale manipola i file descriptors.

```
int fcntl(int fd, F_SETFL, O_NONBLOCK)
```

Esempio:

```
#include <unistd.h> <stdio.h> <signal.h> <errno.h> <stdlib.h>
void handler(int signo){
   printf("SIGPIPE received\n"); perror("Error in handler");
}
int main(void){
   signal(SIGPIPE,handler);
   int fd[2]; char buf[50];
   int esito = pipe(fd); //Create unnamed pipe
   close(fd[0]); //Close read side
   printf("Attempting write\n");
   int numOfWritten = write(fd[1],"writing",8);
   printf("I've written %d bytes\n",numOfWritten);
   perror("Error after write");
}
```

Comunicazione unidirezionale
Un tipico esempio di comunicazione unidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 crea una pipe()
- P1 esegue un fork() e crea P2
- P1 chiude il lato lettura: close(fd[0])
- P2 chiude il lato scrittura: close(fd[1])

- P1 e P2 chiudono l'altro fd appena finiscono di comunicare.

```
#include <stdio.h> <unistd.h> <sys/wait.h>
int main(){
   int fd[2]; char buf[50];
   pipe(fd); //Create unnamed pipe
   int p2 = !fork();
   if(p2){
      close(fd[1]);
      int r = read(fd[0],&buf,50); //Read from pipe
      close(fd[0]); printf("Buf:  '%s'\n",buf);
   }else{
      close(fd[0]);
      write(fd[1],"writing",8); // Write to pipe
      close(fd[1]);
   }
   while(wait(NULL)>0);
}
```

Comunicazione bidirezionale
Un tipico esempio di comunicazione bidirezionale tra un processo scrittore P1 ed un processo lettore P2 è il seguente:

- P1 crea due pipe(), pipe1 e pipe2
- P1 esegue un fork() e crea P2
- P1 chiude il lato lettura di pipe1 ed il lato scrittura di pipe2
- P2 chiude il lato scrittura di pipe1 ed il lato lettura di pipe2
- P1 e P2 chiudono gli altri fd appena finiscono di comunicare.

Esempio:

```
#include <stdio.h> <unistd.h> <sys/wait.h>
#define READ 0, WRITE 1
int main(){
   int pipe1[2]
   int pipe2[2];
   char buf[50];
   pipe(pipe1);
   pipe(pipe2); //Create two unnamed pipe
   int p2 = !fork();
   if(p2){
      close(pipe1[WRITE]); close(pipe2[READ]);
      int r = read(pipe1[READ],&buf,50); //Read from pipe
      close(pipe1[READ]); printf("P2 received:  '%s'\n",buf);
      write(pipe2[WRITE],"Msg from p2",12); // Writes to pipe
```

```
      close(pipe2[WRITE]);
   }else{
      close(pipe1[READ]); close(pipe2[1]);
      write(pipe1[WRITE],"Msg from p1",12); // Writes to pipe
      close(pipe1[WRITE]);
      int r = read(pipe2[READ],&buf,50); //Read from pipe
      close(pipe2[READ]); printf("P1 received:  '%s'\n",buf);
   }
   while(wait(NULL)>0);
}
```

Gestire la comunicazione
Per gestire comunicazioni complesse c'è bisogno di definire un "protocollo". Esempio:

- Messaggi di lunghezza fissa (magari inviata prima del messaggio)
- Marcatore di fine messaggio (per esempio con carattere NULL o newline)

Più in generale occorre definire la sequenza di messaggi attesi.

Esempio: redirige lo stdout di cmd1 sullo stdin di cmd2

```
#include <stdio.h> <unistd.h>
#define READ 0, WRITE 1
int main (int argc, char *argv[]) {
   int fd[2];
   pipe(fd); // Create an unnamed pipe
   if (fork() != 0) { // Parent, writer
      close(fd[READ]); // Close unused end
      dup2(fd[WRITE], 1); // Duplicate used end to stdout
      close(fd[WRITE]); // Close original used end
      execlp(argv[1], argv[1], NULL); // Execute writer program
      perror("error"); // Should never execute
   } else { // Child, reader
      close(fd[WRITE]); // Close unused end
      dup2(fd[READ], 0); // Duplicate used end to stdin
      close(fd[READ]); // Close original used end
      execlp(argv[2], argv[2], NULL); // Execute reader program
      perror("error"); // Should never execute
   }
}
```

1.8.2 FIFO/con nome

Pipe con nome/FIFO
Le pipe con nome, o **FIFO**, sono delle pipe che corrispondono a dei file speciali nel filesystem grazie ai quali i processi, senza vincoli di gerarchia, possono comunicare.

Un processo può accedere ad una di queste pipe se ha i permessi sul file corrispondente ed è vincolato, ovviamente, dall'esistenza del file stesso.
Le FIFO sono interpretate come dei file, perciò si possono usare le funzioni di scrittura/lettura dei file. Restano però delle pipe, con i loro vincoli e le loro capacit.
NB: non sono dei file effettivi, quindi **lseek non funziona**, il loro contenuto è sempre vuoto e non vi ci si può scrivere con un editor!
Normalmente aprire una FIFO blocca il processo finchè anche l'altro lato non è stato aperto. Le differenze tra pipe anonime e FIFO sono solo nella loro creazione e gestione.

Creare una FIFO

`int mkfifo(const char *pathname, mode_t mode);`
Crea una FIFO (un file speciale), solo se non esiste già , con il nome `pathname`.
Il parametro mode può definire i privilegi del file nella solita maniera.

Esempio:

```
#include <sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h>
int main(void){
   char * fifoName = "/tmp/fifo1";
   mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it !exist
   perror("Created?");
}
```

Comunicazione bloccata

```
#include <sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h>
int main(void){
   char * fifoName = "/tmp/fifo1";
   mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it !exist
   perror("Created?");
   if (fork() == 0){ //Child
      open(fifoName,O_RDONLY); //Open READ side of pipe...stuck!
      printf("Open read\n");
   }else{
      sleep(3);
      open(fifoName,O_WRONLY); //Open WRITE side of pipe
      printf("Open write\n");
   }
}
```

Writer

```
#include<sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h><string.h>
int main (int argc, char *argv[]) {
   int fd; char * fifoName = "/tmp/fifo1";
   char str1[80], * str2 = "Im a writer";
   mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if it !exist
   fd = open(fifoName, O_WRONLY); // Open FIFO for write only
```

```
   write(fd, str2, strlen(str2)+1); // write and close
   close(fd);
   fd = open(fifoName, O_RDONLY); // Open FIFO for Read only
   read(fd, str1, sizeof(str1)); // Read from FIFO
   printf("Reader is writing:  %s\n", str1);
   close(fd); }
```

Reader

```
#include<sys/stat.h><sys/types.h><unistd.h><fcntl.h><stdio.h><string.h>
int main (int argc, char *argv[]) {
   int fd; char * fifoName = "/tmp/fifo1";
   mkfifo(fifoName,S_IRUSR|S_IWUSR); //Create pipe if !exist
   char str1[80], * str2 = "I'm a reader";
   fd = open(fifoName , O_RDONLY); // Open FIFO for read only
   read(fd, str1, 80); // read from FIFO and close it
   close(fd);
   printf("Writer is writing:  %s\n", str1);
   fd = open(fifoName , O_WRONLY); // Open FIFO for write only
   write(fd, str2, strlen(str2)+1); // Write and close
   close(fd);
}
```

Comunicazione sul terminale

È possibile usare le FIFO da terminale, leggendo e scrivendo dati tramite gli operatori di redirezione.

```
echo "message for pipe" > /path/nameOfPipe
cat /path/nameOfPipe
```

NB: non si possono scrivere dati usando editor di testo! Una volta consumati i dati, questi non sono più presenti sulla fifo.

Pipe anonime vs FIFO

	pipe	**FIFO**
Rappresentazione	Buffer	Buffer
Rifermento	anonimo	File
Accesso	2 FileDescriptor	1 FileDescriptor
Persistenza	Eliminazione alla terminazione dei processi	Esistenza finchè c'è il file
Vincoli di accesso	antenato comune	Permessi su file
Creazione	pipe()	mkfifo()
MaxBytes per atomicità	PIPE_BUF = 4096	PIPE_BUF = 4096

Conlusioni

PIPE e FIFO sono sistemi di comunicazione tra processi ("parenti", tipicamente padre-figlio, nel primo caso e in generale nel secondo caso) che consentono **scambi** di informazioni (**messaggi**) e **sincronizzazione** (grazie al fatto di poter essere "bloccanti").

1.9 Lab-09: Message Queues

Message Queues
Una coda di messaggi, message queue, è una lista concatenata memorizzata all'interno del kernel ed identificata con un ID (un intero positivo univoco), chiamato **queue identifier**.
Questo ID viene condiviso tra i processi interessati, e viene generato attraverso una chiave univoca.
Una coda deve essere innanzitutto generata in maniera analoga ad una FIFO, impostando dei permessi. Ad una coda esistente si possono aggiungere o recuperare messaggi tipicamente in modalità "autosincrona": la lettura attende la presenza di un messaggio, la scrittura attende che via sia spazio disponibile. Questi comportamenti possono però essere configurati.

Quanto trattiamo di message queue, abbiamo due identificativi:

- **Key**: intero che identifica un insieme di risorse condivisibili nel kernel, come semafori, memoria condivisa e code. Questa chiave univoca deve essere nota a pi processi, e viene usata per ottenere il queue identifier.
- **Queue identifier**: id univoco della coda, generato dal kernel ed associato ad una specifica key. Questo ID viene usato per interagire con la coda.

1.9.1 Creazione

Creazione

```
int msgget(key_t key, int msgflg)
```

Restituisce l'**identificativo** di una coda basandosi sulla chiave **key** e sui flags:

- `IPC_CREAT`: cra una cosa se non esiste già , altrimenti restituisce l'identificativo di quella presente
- `IPC_EXCL`: usato in coppia con CREAT, fallisce se la cosa esiste già
- `0xxx`: permessi pr accedere alla coda, analogo a quella del file system. utilizzabili anche "S_IRUSR"

Ottenere una chiave univoca

```
key_t ftok(const char *path, int id)
```

Restituisce una chiave basandosi sul path (una cartella o un file), esistente ed accessibile nel file-system, e sull'id numerico. La chiave dovrebbe essere univoca e sempre la stessa per ogni coppia $\langle path, id \rangle$ in ogni istante sullo stesso sistema.
Un metodo d'uso, per evitare possibili conflitti, potrebbe essere generare un path (es. un file) temporaneo univoco, usarlo, eventualmente rimuoverlo, ed usare l'id per per rappresentare diverse categorie di code, a mo' di indice

```
#include <sys/ipc.h>
key_t queue1Key = ftok("/tmp/unique", 1);
key_t queue2Key = ftok("/tmp/unique", 2);
```

Esempio creazione:

```
<sys/types.h><sys/ipc.h> <sys/msg.h><stdio.h><fcntl.h>
void main(){
   remove("/tmp/unique"); //Remove file
   key_t queue1Key = ftok("/tmp/unique",1); //Get unique key, fail
   creat("/tmp/unique", 0777); //Create file
   queue1Key = ftok("/tmp/unique", 1); //Get unique key → ok
   int queueId = msgget(queue1Key, 0777 | IPC_CREAT); //crt q, ok
   queueId = msgget(queue1Key, 0777); //Get q → ok
   msgctl(queue1Key,IPC_RMID,NULL); //Remove nonexist q → fail
   msgctl(queueId,IPC_RMID,NULL); //Remove q → ok
   queueId = msgget(queue1Key, 0777); //Get nonexist q → fail
   queueId = msgget(queue1Key, 0777 | IPC_CREAT); //Create q → ok
   queueId = msgget(queue1Key, 0777 | IPC_CREAT); //Get q → ok
   /* Create already existing queue -> fail */
   queueId = msgget(queue1Key , 0777 | IPC_CREAT | IPC_EXCL);
}
```

NB: Le queue sono persistenti

1.9.2 Comunicazione

Comunicazione

```
struct msg_buffer{
   long mtype;
   char mtext[100];
} message;
```

Ogni messaggio inserito nella coda ha:

- Un tipo, categoria, rappresentato da un intero maggiore di 0
- Una grandezza non negativa
- Un payload, un insieme di dati (bytes) di lunghezza corretta

Al contrario delle FIFO, i messaggi in una coda possono essere recuperati anche sulla base del tipo e non solo del loro ordine "assoluto" di arrivo. Così come i files, le code sono delle strutture persistenti che continuano ad esistere, assieme ai messaggi in esse salvati, anche alla terminazione del processo che le ha create. L'**eliminazione** della coda deve essere **esplicita**.
Il payload del messaggio non deve essere necessariamente un campo testuale: può essere una qualsiasi struttura dati. Infatti, un messaggio può anche essere senza payload.

Invio messaggi

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int
msgflg);
```

Aggiunge una **copia** del messaggio puntato da **msgp**, con un payload di dimensione **msgsz**, alla coda identificata da msquid. Il messaggio viene inserito immediatamente se c'è abbastanza spazio disponibile, altrimenti la chiamata si blocca fino a che abbastanza spazio diventa disponibile. Se msgflg IPC_NOWAIT allora la chiamata fallisce in assenza di spazio.
NB: msgsz è la **grandezza del payload** del messaggio, non del messaggio intero (che contiene anche il tipo)! Per esempio, sizeof(msgp.mtext)

Ricezione

- `ssize_t msgrcv(int msqid,void *msgp,size_t msgsz, long msgtyp,int msgflg)`

  Rimuove un messaggio dalla coda msqid e lo salva nel buffer msgp. msgsz specifica la lunghezza massima del payload del messaggio (per esempio mtext della struttura msgp).
  Se il payload ha una lunghezza maggiore e msgflg MSG_NOERROR allora il payload viene troncato (viene persa la parte in eccesso), se MSG_NOERROR non è specificato allora il payload non viene eliminato e la chiamata fallisce.

  Se non sono presenti messaggi, la chiamata si blocca in loro attesa. Il flag IPC_NOWAIT fa fallire la syscall se non sono presenti messaggi

- `ssize_t msgrcv(int msqid,void *msgp,size_t msgsz, long msgtyp,int msgflg)`

  A seconda di msgtyp viene recuperato il messaggio:

  - $msgtyp = 0$: primo messaggio della coda (FIFO)
  - $msgtyp > 0$: primo messaggio di tipo msgtyp, o primo messaggio di tipo diverso da msgtyp se MSG_EXCEPT è impostato come flag
  - $msgtyp < 0$: primo messaggio il cui tipo T è $min(T \leq |msgtyp|)$

Esempi Comunicazione

1. ```
   #include
   <sys/types.h><sys/ipc.h><sys/msg.h><string.h><stdio.h>
   struct msg_buffer{
      long mtype;
      char mtext[100];
   } msgp, msgp2; //Two different message buffers
   int main(void){
   ```
   ```

```
   msgp.mtype = 20;
   strcpy(msgp.mtext,"This is a message");
   key_t q1key = ftok("/tmp/unique", 1);
   int qId = msgget(q1key , 0777 | IPC_CREAT | IPC_EXCL);
   int esito = msgsnd(qId , &msgp, sizeof(msgp.mtext),0);
   esito = msgrcv(qId , &msgp2, sizeof(msgp2.mtext),20,0);
   printf("Received %s\n",msgp2.mtext);
}
```

2. ```
   #include <sys/types.h> <sys/ipc.h> <sys/msg.h> <string.h>
   typedef struct book{
      char title[10];
      char description[200];
      short chapters;
   } Book;
   struct msg_buffer{
      long mtype;
      Book mtext;
   } msgp_snd, msgp_rcv; //Two different message buffers
   int main(void){
      msgp_snd.mtype = 20;
      strcpy(msgp_snd.mtext.title,"Title");
      strcpy(msgp_snd.mtext.description,
              "This is a description");
      msgp_snd.mtext.chapters = 17;
      key_t queue1Key = ftok("/tmp/unique", 1);
      int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
      int esito = msgsnd(queueId, &msgp_snd,
                   sizeof(msgp_snd.mtext),0);
      esito = msgrcv(queueId, &msgp_rcv,
                   sizeof(msgp_rcv.mtext),20,0);
      printf("Received:  %s %s %d\n",msgp_rcv.mtext.title,
              msgp_rcv.mtext.description,
              msgp_rcv.mtext.chapters);
   }
   ```

3. ```
   #include <sys/types.h> <sys/ipc.h> <sys/msg.h> <string.h>
   struct msg_buffer{
      long mtype;
      char mtext[100];
   } msgp_snd,msgp_rcv; //Two different message buffers
   int main(int argc, char ** argv){
      int to_fetch = atoi(argv[1]); //Input to decide which msg
   to get
      key_t queue1Key = ftok("/tmp/unique", 1);
      int queueId = msgget(queue1Key , 0777 | IPC_CREAT);
      msgp_snd.mtype = 20;
   ```

```
   strcpy(msgp_snd.mtext,"A message of type 20");
   int esito = msgsnd(queueId , &msgp_snd,
sizeof(msgp.mtext),0);
   msgp_snd.mtype = 10; //Re-use the same message
   strcpy(msgp_snd.mtext,"Another message of type 10");
   esito = msgsnd(queueId , &msgp_snd, sizeof(msgp.mtext),0);
   esito = msgrcv(queueId , &msgp_rcv,
sizeof(msgp_rcv.mtext),to_fetch,0);
   printf("Received:  %s %s %d\n",msgp_rcv.mtext.title,
           msgp_rcv.mtext.description,
           msgp_rcv.mtext.chapters);
}
```

1.9.3 Modifica coda

Modificare la coda

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Modifica la coda identificata da msqid secondo i comandi cmd, riempiendo buf con informazioni sulla coda (ad esempio timestamp dell'ultima scrittura, dell'ultima lettura, numero messaggi nella coda, etc). Valori di cmd possono essere:

- IPC_STAT: recupera informazioni da kernel
- IPC_SET: imposta alcuni parametri a seconda di buf
- IPC_RMID: rimuove immediatamente la coda
- IPC_INFO: recupera informazioni generali sui limiti delle code nel sistema
- MSG_INFO: come IPC_INFO ma con informazioni differenti
- MSG_STAT: come IPC_STAT ma con informazioni differenti

Struttura MSQID_DS

```
struct msqid_ds {
   struct ipc_perm msg_perm; /* Ownership and permissions */
   time_t msg_stime; /* Time of last msgsnd(2) */
   time_t msg_rtime; /* Time of last msgrcv(2) */
   time_t msg_ctime; //Time of creation or last modifi by msgctl
   unsigned long msg_cbytes; /* # of bytes in queue */
   msgqnum_t msg_qnum; /* # of messages in queue */
   msglen_t msg_qbytes; /* Maximum # of bytes in queue */
   pid_t msg_lspid; /* PID of last msgsnd(2) */
   pid_t msg_lrpid; /* PID of last msgrcv(2) */
};
```

Struttura IPC_PERM

```
struct ipc_perm {
   key_t __key; /* Key supplied to msgget(2) */
   uid_t uid; /* Effective UID of owner */
   gid_t gid; /* Effective GID of owner */
   uid_t cuid; /* Effective UID of creator */
   gid_t cgid; /* Effective GID of creator */
   unsigned short mode; /* Permissions */
   unsigned short __seq; /* Sequence number */
};
```

Esempio modifica

```
int main(void){
   struct msqid_ds mod;
   int esito = open("/tmp/unique",O_CREAT,0777);
   key_t queue1Key = ftok("/tmp/unique", 1);
   int queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU );
   msgctl(queueId,IPC_RMID,NULL);
   queueId = msgget(queue1Key , IPC_CREAT | S_IRWXU );
   esito = msgctl(queueId,IPC_STAT,&mod); //Get info on queue
   printf("Current permission on queue:  %d\n",mod.msg_perm.mode);
   mod.msg_perm.mode = 0000;
   esito = msgctl(queueId,IPC_SET,&mod); //Modify queue
   printf("Current permission on queue:
           %d\n\n",mod.msg_perm.mode);
}
```

NB: Le code sono un comodo metodo di comunicazione per l'invio e la ricezione di informazioni anche "complesse" tra generici

1.10 Lab-10: Thread

I **thread** sono singole sequenze di esecuzione all'interno di un processo, aventi alcune delle proprietà dei processi. I threads non sono indipendenti tra loro e condividono il codice, i dati e le risorse del sistema assegnate al processo di appartenenza. Come ogni singolo processo, i threads hanno alcuni elementi indipendenti, come lo stack, il PC ed i registri del sistema.
La creazione di threads consente un parallelismo delle operazioni in maniera rapida e semplificata. Context switch tra threads è rapido, così come la loro creazione e terminazione. Inoltre, la comunicazione tra threads è molto veloce.
Per la compilazione è necessario aggiungere il flag -pthread, ad esempio: gcc -o program main.c -pthread

IN C
In C i thread corrispondono a delle funzioni eseguite in parallelo al codice principale. Ogni thread è **identificato da un ID** e può essere gestito come un processo figlio, con funzioni che attendono la sua terminazione. Sebbene da un punto di vista l'esecuzione di diversi thread sia sempre parallela, a causa delle politiche di scheduling delle CPU, l'esecuzione pu essere parallela solo se la CPU la supporta, ossia se dispone di pi core.

1.10.1 Creazione

Creazione

```
int pthread_create(
pthread_t *restrict thread, /* Thread ID */
const pthread_attr_t *restrict attr, /* Attributes */
void *(*start_routine)(void *), /* Function to be executed */
void *restrict arg /* Parameters to above function */
);
```

Quando si crea un nuovo thread, è necessario fornire un puntatore pthread_t, che verr riempito con il nuovo ID generato. attr consente di modificare il comportamento dei thread, mentre start_routine serve a definire quale funzione deve essere eseguita dal thread. arg è un puntatore void che può essere utilizzato per passare qualsiasi argomento sia richiesto.
NB: void* il tipo di variabile più grande e può quindi essere usato per puntare a qualsiasi struttura di dati

Esempio

```
#include <stdio.h> <pthread.h> <unistd.h>
void * my_fun(void * param){
   printf("This is a thread that received %d\n", *(int *)param);
   return (void *)3;
}
int main(void){
```

```
   pthread_t t_id;
   int arg=10;
// We need to cast the augment to a void *.  We are passing the
address of the variable!
   pthread_create(&t_id, NULL, my_fun, (void *)&arg);
   printf("Executed thread with id %ld\n",t_id);
   sleep(3);
}
```
Le modifiche al valore di `arg` vengono viste anche dal thread

All'interno del kernel:

```
#include <pthread.h>
void * my_fun(void * param){
   while(1);
}
int main(void){
   pthread_t t_id;
   pthread_create(&t_id, NULL, my_fun, NULL);
   while(1);
}
```

1.10.2 Terminazione

Terminazione
Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione void pthread_exit(void * retval); dal thread, specificando un puntatore da restituire.
- Effettuando un return dalla funzione associata al thread, specificando un valore di ritorno.
- Cancellando il thread da un altro thread.
- Qualche thread chiama exit(), o il thread che esegue main() ritorna dallo stesso, terminando cos tutti i threads.

1.10.3 Cancellazione

Cancellazione

```
int pthread_cancel(pthread_t thread);
```

Invia una richiesta di cancellazione al thread specificato, il quale reagir (come e quando) a seconda di due suoi attributi: cancel state e cancel type. Il cancel state di un thread definisce se il thread deve terminare o meno quando una richiesta di cancellazione viene ricevuta. Il cancel type di un thread definisce come il thread deve terminare. Sia lo stato che il tipo possono essere definiti alla creazione del thread o durante la sua esecuzione, allinterno del thread stesso.

1.10.4 Cambiare il thread cancel state

Cambiare il thread cancel state

```
int pthread_setcancelstate(int state, int *oldstate);
```

Modifica il cancel state del thread in esecuzione. Mentre oldstate viene riempito con lo stato precedente, state pu contenere una delle seguenti macro:

- PTHREAD_CANCEL_ENABLE: ogni richiesta di cancellazione viene gestita a seconda del type del thread. Questa la modalit default.
- PTHREAD_CANCEL_DISABLE: ogni richiesta di cancellazione aspetter fino a che il cancel state del thread non diventa PTHREAD_CANCEL_ENABLE.

1.10.5 Cambiare il thread cancel type

```
int pthread_setcanceltype(int type, int *oldtype);
```
Modifica il cancel type del thread in esecuzione. Mentre oldtype viene riempito con il type precedente, type pu contenere una delle seguenti macro:

- PTHREAD_CANCEL_DEFERRED: la terminazione aspetta lesecuzione di un cancellation point. Questa la modalit default.
- PTHREAD_CANCEL_ASYNCHRONOUS: la terminazione avviene appena la richiesta viene ricevuta.

Cancellation points sono delle specifiche funzioni definite nella libreria pthread.h (list). Di solito sono system calls.

Esempio 1

```
#include <stdio.h> <pthread.h> <unistd.h>
int i = 1;
void * my_fun(void * param){
   if(i--)
   pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL); //Change
mode
   printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
   printf("Thread %ld finished\n",*(pthread_t *)param);
}
int main(void){
   pthread_t t_id1, t_id2;
   pthread_create(&t_id1, NULL, my_fun, (void *)&t_id1); sleep(1);
//Create
   pthread_cancel(t_id1); //Cancel
   printf("Sent cancellation request for thread %ld\n",t_id1);
   pthread_create(&t_id2, NULL, my_fun, (void *)&t_id2); sleep(1);
//Create
```

```
   pthread_cancel(t_id2); //Cancel
   printf("Sent cancellation request for thread %ld\n",t_id2);
   sleep(5); printf("Terminating program\n");
}
```

Esempio 2

```
#include <stdio.h> <pthread.h> <unistd.h> <string.h>
int tmp = 0;
void * my_fun(void * param){
   pthread_setcanceltype(*(int *)param,NULL); // Change type
   for (long unsigned i = 0; i < (0x9FFF0000); i++); //just wait
   tmp++;
   open("/tmp/tmp",O_RDONLY); //Cancellation point!
}
int main(int argc, char ** argv){ //call program with async or
defer
   pthread_t t_id1; int arg;
   if(!strcmp(argv[1],"async")) arg = PTHREAD_CANCEL_ASYNCHRONOUS;
   else if(!strcmp(argv[1],"defer")) arg =
PTHREAD_CANCEL_DEFERRED;
   pthread_create(&t_id1, NULL, my_fun, (void *)&arg); sleep(1);
//Create
   pthread_cancel(t_id1); sleep(5); //Cancel
   printf("Tmp %d\n",tmp);
}
```

1.10.6 Aspettare un thread

Aspettare un thread: join
Un processo (thread) che avvia un nuovo thread pu aspettare la sua terminazione mediante la funzione:

```
int pthread_join(pthread_t thread, void ** retval);
```

Questa funzione ritorna quando il thread identificato da thread termina, o subito se il thread è gi terminato. Se il valore di ritorno del thread non nullo (parametro di pthread_exit() o di return), esso viene salvato nella variabile puntata da retval. Se il thread era stato cancellato, retval riempito con PTHREAD_CANCELED.
Solo se il thread è joinable pu essere aspettato!
Un thread pu essere aspettato da al massimo un thread

1.10.7 Detach state di un thread

I thread sono creati per impostazione predefinita nello stato joinable, che consente a un altro thread di attendere la loro terminazione tramite il

comando pthread_join(). I thread joinable non rilasciano le loro risorse alla terminazione, ma quando un thread li aspetta (salvando lo stato di uscita, come i sottoprocessi), o alla terminazione del processo stesso. Al contrario, i thread detached rilasciano le loro risorse immediatamente al termine, ma non permettono ad altri processi di aspettarli. NB: un thread detached non pu diventare joinable durante la sua esecuzione, mentre è possibile il contrario.

Cambiare il detach state

```
int pthread_detach(pthread_t thread);
```

Questo comando pu essere eseguito da un thread qualunque e cambia il detach state di thread da joinable a detached. **Ricorda che una volta cambiato lo stato non si può invertire**

Esempi join 1

```
#include <stdio.h> <pthread.h> <unistd.h>
void * my_fun(void * param){
   printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
   char * str = "Returned string";
   pthread_exit((void *)str); //or return (void *) str;
}
int main(void){
   pthread_t t_id;
   void * retFromThread; //This must be a pointer to void!
   pthread_create(&t_id, NULL, my_fun, (void *)&t_id); //Create
   pthread_join(t_id,&retFromThread); // wait thread
   // We must cast the returned value!
   printf("Thread %ld returned '%s'\n",t_id,(char
*)retFromThread);
}
```

Esempi join 2

```
#include <stdio.h> <pthread.h> <unistd.h>
void * my_fun(void *param){
   printf("This is a thread that received %d\n", *(int *)param);
   return (void *)3;
}
int main(void){
   pthread_t t_id;
   int arg=10, retval;
   pthread_create(&t_id, NULL, my_fun, (void *)&arg);
   printf("Executed thread with id %ld\n",t_id);
   sleep(3);
   pthread_join(t_id, (void **)&retval); //A pointer to a void
pointer
   printf("retval=%d\n", retval);
}
```

Esempio join 3

```
#include <stdio.h> <pthread.h> <unistd.h>
void * my_fun(void *param){
   printf("This is a thread that received %d\n", *(int *)param);
   int i = *(int *)param * 2; //Local variable ceases to exist!
   return (void *)&i;
}
int main(void){
   pthread_t t_id;
   int arg=10, retval;
   pthread_create(&t_id, NULL, my_fun, (void *)&arg);
   printf("Executed thread with id %ld\n",t_id);
   sleep(3);
   pthread_join(t_id, (void **)&retval); //A pointer to a void
pointer
   printf("retval=%d\n", retval);
}
```

1.10.8 Attributi thread

Ogni thread viene creato con gli attributi specificati nella struttura `pthread_attr_t`. Questa struttura, analogamente a quella utilizzata per gestire le maschere di segnale, un oggetto utilizzato solo quando viene creato un thread ed quindi indipendente da esso (se cambia, gli attributi del thread non cambiano).
La struttura deve essere inizializzata, il che imposta tutti gli attributi al loro valore predefinito. Una volta utilizzata e non pi necessaria, la struct deve essere distrutta.
I vari attributi della struttura possono e devono essere modificati individualmente con alcune funzioni dedicate.

- `int pthread_attr_init(pthread_attr_t *attr)` Inizializza la struttura con tutti gli attributi default.
- `int pthread_attr_destroy(pthread_attr_t *attr)` Distrugge la struttura.
- `int pthread_attr_setxxxx(pthread_attr_t *attr, params);` Imposta un certo attributo ad un certo valore.
- int pthread_attr_getxxxx(const pthread_attr_t *attr, params); Ottiene un certo attributo

Esempio attributi

```
#include <stdio.h> <pthread.h> <unistd.h>
void * my_fun(void *param){
   printf("This is a thread that received %d\n", *(int
*)param);return (void*)3;
```

```
}
int main(void){
   pthread_t t_id; pthread_attr_t attr;
   int arg=10, detachSTate;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
//Set detached
   pthread_attr_getdetachstate(&attr,&detachSTate); //Get detach
state
   if(detachSTate == PTHREAD_CREATE_DETACHED)
printf("Detached\n");
   pthread_create(&t_id, &attr, my_fun, (void *)&arg);
   printf("Executed thread with id %ld\n",t_id);
   pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
//Inneffective
   sleep(3); pthread_attr_destroy(&attr);
   int esito = pthread_join(t_id, (void **)&detachSTate);
   printf("Esito '%d' is different 0\n", esito);
}
```

1.10.9 Thread e Segnali

Quando viene inviato un segnale ad un processo non si pu sapere quale thread andr a gestirlo. Per evitare problemi o comportamenti inattesi, importante gestire correttamente le maschere dei segnali dei singoli thread con
`pthread_attr_setsigmask_np(pthread_attr_t *attr,const sigset_t *sigmask)`
la quale usa *sigmask per impostare la maschera dei segnali nella struttura *attr.
poi possibile usare funzioni come `sigwait()` e `sigwaitinfo()` per gestire lattesa di un segnale. Infine, possibile inviare un segnale ad un thread specifico usando
`int pthread_kill(pthread_t thread, int sig);`

1.10.10 Mutex

Problema della sincronizzazione
Quando eseguiamo un programma con pi thread essi condividono alcune risorse, tra le quali le variabili globali. Se entrambi i thread accedono ad una sezione di codice condivisa ed hanno la necessit di accedervi in maniera esclusiva allora dobbiamo instaurare una sincronizzazione. I risultati, altrimenti, potrebbero essere inaspettati.

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>
pthread_t tid[2];
```

```
int counter = 0;
void *thr1(void *arg){
   counter = 1;
   printf("Thread 1 has started with counter %d\n",counter);
   for (long unsigned i = 0; i < (0x00FF0000); i++); //wait some
cycles
   counter += 1;
   printf("Thread 1 expects 2 and has:  %d\n", counter);
}
void *thr2(void *arg){
   counter = 10;
   printf("Thread 2 has started with counter %d\n",counter);
   for (long unsigned i = 0; i < (0xFFF0000); i++); //wait some
cycles
   counter += 1;
   printf("Thread 2 expects 11 and has:  %d\n", counter);
}
void main(void){
   pthread_create(&(tid[0]), NULL, thr1,NULL);
   pthread_create(&(tid[1]), NULL, thr2,NULL);
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
}
```

I mutex sono dei semafori imposti ai thread. Essi possono proteggere una determinata sezione di codice, consentendo ad un thread di accedervi in maniera esclusiva fino allo sblocco del semaforo. Ogni thread che vorr accedere alla stessa sezione di codice dovr aspettare che il semaforo sia sbloccato, andando in sleep fino alla sua prossima schedulazione.
I mutex vanno inizializzati e poi assegnati ad una determinata sezione di codice. Il blocco e sblocco manuale.
NB: i mutex non regolano laccesso alla memoria o alle variabile, ma solo a porzioni di codice.

1.10.11 Creazione Eliminazione Mutex

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr)
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

1.10.12 Bloccaggio e sbloccaggio

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Dopo essere stato creato, un mutex deve essere bloccato per essere efficace.
Non appena un thread lo blocca, un altro thread deve attendere che venga sbloccato prima di procedere al suo blocco.
Quando si richiama il blocco, un thread attende che il mutex sia libero e poi lo blocca

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>
pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;
void* thr1(void* arg){
   pthread_mutex_lock(&lock);
   counter = 1;
   printf("Thread 1 has started with counter %d\n",counter);
   for (long unsigned i = 0; i < (0x00FF0000); i++);
   counter += 1;
   pthread_mutex_unlock(&lock);
   printf("Thread 1 expects 2 and has:  %d\n", counter);
}
void * thr2(void* arg){
   pthread_mutex_lock(&lock);
   counter = 10;
   printf("Thread 2 has started with counter %d\n",counter);
   for (long unsigned i = 0; i < (0xFFF0000); i++);
   counter += 1;
   pthread_mutex_unlock(&lock);
   printf("Thread 2 expects 11 and has:  %d\n", counter);
}
int main(void){
   pthread_mutex_init(&lock, NULL);
   pthread_create(&(tid[0]), NULL, thr1,NULL);
   pthread_create(&(tid[1]), NULL, thr2,NULL);
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
   pthread_mutex_destroy(&lock);
}
```

I thread sono una sorta di processi leggeri che permettono di eseguire funzioni in concorrenza in modo pi semplice rispetto alla generazioni di processi veri e propri (forking).
I MUTEX sono un metodo semplice ma efficace per eseguire sezioni critiche in processi multithread.
importante limitare al massimo la sezione critica utilizzando lock/unlock per la porzione di codice pi piccola possibile, e solo se assolutamente necessario (per esempio quando possono capitare accessi concorrenti).

1.11 Script

1.11.1 lez-01

1.0) Esempio prova

```
 nargs=$#
while [[ $1 != "" ]]; do
     echo "ARG=$1"
     shift
done
```

1.1) Scrivere uno script che dato un qualunque numero di argomenti li restituisca in output in ordine inverso.

```
lista=()
while [[ $1 != "" ]]; do
     lista=( "$1" "$lista[@]" )
     shift
done
echo "$lista[@]"
```

1.2) Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all'output generato da "ls" (che si pu usare ma senza opzioni). Per semplicità , assumere che tutti i file e le cartelle non abbiano spazi nel nome.

```
dati=($( ls ))
lista=()
for i in $!dati[@]; do
     lista=( "$dati[$i]" "$lista[@]" )
done
echo $( ls )
echo "$lista[@]"
```

1.11.2 lez-02

2.1) 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.

```
SHELL := /bin/bash
FOLDER := /tmp

help:
     @echo "make -f makefile_backup backup FOLDER=<path>"

backup:
     @echo "Backup of folder $(FOLDER) as $(FOLDER).bak..." ;
```

```
	sleep 2s
	@[[ -d $(FOLDER).bak ]] && echo "?Error" || cp -rp $(FOLDER)
$(FOLDER).bak

restore:  $(FOLDER).bak
	@echo "Restore of folder $(FOLDER) from $(FOLDER).bak..." ;
	sleep 2s
	@[[ -d $(FOLDER) ]] && echo "?Error" || cp -rp $(FOLDER).bak
$(FOLDER)

.PHONY: help backup restore
```

1.11.3 lez-03

3.1) Scrivere un'applicazione che data una stringa come argomento ne stampa a video la lunghezza, ad esempio: `./lengthof "Questa frase ha 28 caratteri"`
deve restituire a video il numero 28

```
#include <stdio.h>
int main(int argc, char **argv){
	int code = 0;
	int len = 0;
	char *p;
	printf("%d\n", argc);
	if(argc != 2) {
		fprintf(stderr,"Usage:  %s <stringa>\n", argv[0]);
		code = 2;
	}else {
		p = argv[1]; //Copy pointer to first argument
		while (*p != 0 ){ //Check if character is termination
			p++; //Move to next char
			len++; //Increase length count
		}
		printf("%d\n", len);
	}
	return code;
}
```

3.2) Scrivere un'applicazione che definisce una lista di argomenti validi e legge quelli passati alla chiamata verificandoli e memorizzando le opzioni corrette, restituendo un errore in caso di un'opzione non valida.
Non ho voglia di copiarlo

3.3) Realizzare funzioni per stringhe `char *stringrev(char * str)` (inverte ordine caratteri) e `int stringpos(char * str, char chr)` (cerca chr in str e restituisce la posizione)
(Non completo, controllare stringrev, da segmentation fault)

1.11.4 lez-04