Laboratorio di Sistemi Operativi

A.A. 2021-2022

Docente: A. Rossi

Dispense tratte dalle slide del corso di Informatica a cura dello studente **S. Cerrone**

1. Introduzione

1.1 Introduzione a Unix

1965 Bell Labs (con MIT e General Electrics) lavora ad un nuovo sistema operativo: Multics. Principali caratteristiche: multi-utente, multi-processo (time-sharing), con file system multi-livello (gerarchico)

1969 Bell Labs abbandona Multics. Ken Thompson, Dennis Ritchie, Rudd Canaday e Doug McIlroy progettano e implementano la prima versione di Unix insieme ad alcune utility. Il nome Unix è di Brian Kernighan: gioco di parole su Multics.

1970 Prima versione di Unix, per PDP-7

1973 Unix viene riscritto in C (Dennis Ritchie)

1.2 Caratteristiche Unix

- Multi-utente, con sofisticato sistema di protezioni e permessi.
- Multi-processo, con time-sharing
- Filesystem gerarchico con radice unica
- Basato su **kernel**:
    - Il nucleo del SO è l'unica porzione che deve essere adattata all'HW
    - Il nucleo è l'unica porzione che gira in modalità privilegiata (kernel mode)

1.2.1 Kernel

Il **kernel** (nucleo del sistema) gestisce le risorse essenziali di un calcolatore:

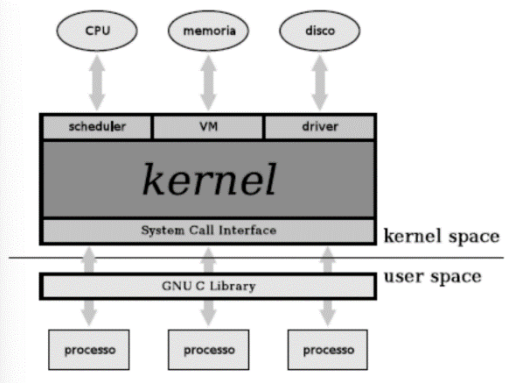
- **CPU**: Lo scheduler stabilisce, ad intervalli fissi e sulla base delle priorità, quale processo deve essere mandato in esecuzione
- **Memoria**: Il kernel gestisce la memoria virtuale (simula uno spazio maggiore di quello che effettivamente ha la memoria centrale), che consente di assegnare a ciascun processo uno spazio di indirizzi virtuale che il kernel, con l'ausilio della unità di gestione della memoria, rimappa sulla memoria fisica (RAM o disco)
- **Periferiche**: Le periferiche vengono viste attraverso un'interfaccia astratta che permette di trattarle come fossero file, secondo il concetto per cui "*everything is a file*" (interfacce di rete escluse)

Tutto il reso, anche l'interazione con l'utente, è ottenuto tramite programmi eseguiti dal kernel, che accedono alle risorse hardware tramite delle richieste a quest'ultimo.

1.2.2 Kernel e System Calls

Le interfacce con cui i programmi possono accedere all'hardware vanno sotto il nome di **system call** (chiamate al sistema): un insieme di funzioni che un programma può chiamare, per le quali viene generata un'interruzione del processo, passando il controllo dal programma al kernel.

Queste chiamate al sistema vengono rimappate in funzioni definite dentro opportune librerie (Libreria Standard del C)



1.2.3 Sistema Multiutente

Il kernel Unix nasce fin dall'inizio come sistema multiutente. Ogni utente ha un nome (*username*), una password e un identificativo numerico (*user id* o *uid*).

Sono previsti meccanismi di permessi e protezioni per impedire che utenti diversi possano danneggiarsi a vicenda o danneggiare il sistema. È presente un utente speciale privilegiato, ***superuser***, il cui username è di norma *root*, ed il cui uid è zero: è l'amministratore del sistema.

1.2 La Shell

La shell (guscio) è un interprete di comandi che si interpone tra l'utente ed il sistema operativo. In sistemi Unix qualsiasi operazione può essere eseguita da una sequenza di comandi shell.

La shell può eseguire uno script oppure interagire in modalità interattiva.
L'utente fornisce al prompt uno dei seguenti comandi:

- nome di un comando built-in
- nome di un file eseguibile
- nome di uno script, cioè file testuale dotato del permesso di esecuzione

1.2.1 I comandi Unix

comando $[argomenti]$

Gli argomenti, separati da almeno un separatore (di default il carattere spazio), possono essere:

- opzioni o flag (cominciano con un trattino "−")
- parametri

L'ordine delle **opzioni** è, in genere, **irrilevante** mentre l'ordine dei **parametri** è, in genere, **rilevante**.
Inoltre, bisogna ricordare che Unix è **CASE SENSITIVE**.

Comando	**Significato**
ls	Visualizza la lista di file nella directory, come il comando dir in DOS
cd *directory*	Cambia directory
passwd	Cambia password
file *filename*	Visualizza il tipo di file o il tipo di file con nome *filename*
cat *textfile*	Riversa il contenuto di *textfile* sullo screen
pwd	Visualizza la directory di lavoro corrente
exit *or* logout	Lascia la sessione
man *command*	Leggi pagine manuale su *command*
info *command*	Leggi pagine info su *command*

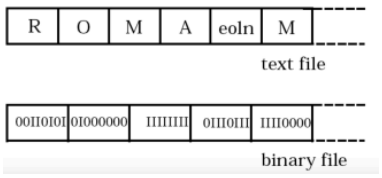
…

1.3 File System

I tipi principali di File sono: file ordinari; directory e file speciali. Il sistema assegna biunivocamente a ciascun file un identificatore numerico, detto **i-number** (*index-number*), che gli permette di rintracciarlo nel file system.

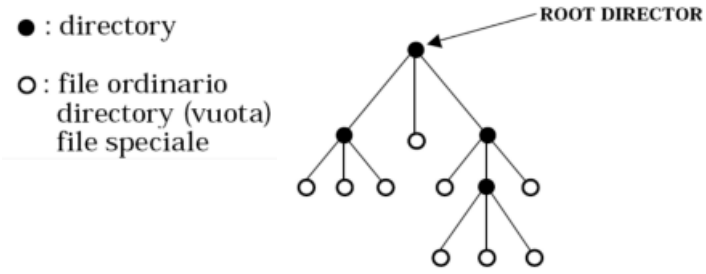
1.3.1 I File Ordinari

Sono sequenze di byte (byte streams) che possono contenere informazioni qualsiasi (dati, programmi sorgente, programmi oggetto, …).
Il sistema non impone alcuna struttura interna.

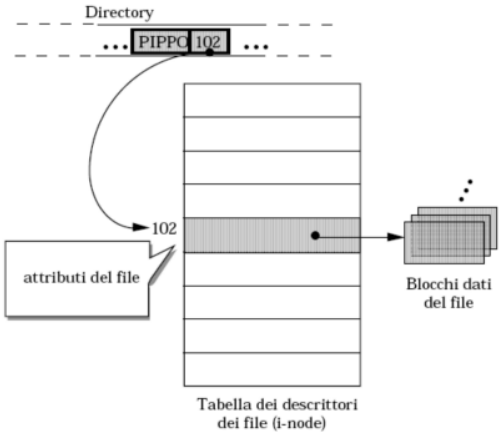


1.3.2 Organizzazione dei File

Per consentire all'utente di rintracciare facilmente i propri file, Unix permette di raggrupparli in cartelle, dette **directories**, organizzate in una (unica) struttura gerarchica:



1.3.3 Implementazione dei File



Attributi di un File

Per ogni file (ordinario, directory, speciale) Unix mantiene le seguenti informazioni nel descrittore dei file:

- **Tipo**: ordinario, directory, speciale?
- **Posizione**: dove si trova?
- **Dimensione**: quanto è grande?
- **Numero di links**: quanti nomi ha?
- **Proprietario**: chi lo possiede?
- **Permessi**: chi può usarlo e come?
- **Creazione**: quando è stato creato?
- **Modifica**: quando è stata l'ultima modifica?
- **Accesso**: quando è stato l'accesso più recente?

1.3.4 Path Assoluti e Relativi

Si può identificare un file tramite il suo **path assoluto**, che descrive il cammino dalla root-directory al file:

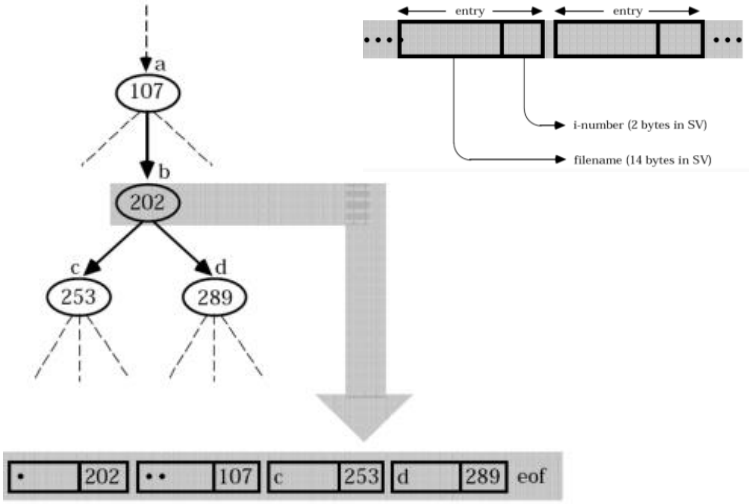
$$\underbrace{/\ dir}_{root}\ /\mathrm{dir}/\ldots/\mathrm{dir}/\mathrm{filename}$$
$$/ \rightarrow separatore$$

Se un path **non** comincia con "/", allora si intende **relativo alla working directory**.

1.3.4 Directories

Sono sequenze di bytes come i file ordinari ma differiscono da quest'ultimi solo perché non possono essere scritte da programmi ordinari.

Il loro contenuto è una serie di **directory entries**: coppie formate da un nome di file e un i-number.



Almeno due entry in ogni directory:

- la directory stessa "."
- la directory padre ".."

Directories:

- /bin (binary) comandi eseguibili
- /dev file speciali (I/O devices)
- /etc file per l'amministrazione del sistema, ad esempio: /etc/passwd
- /lib librerie di programmi
- /tmp area temporanea usata dal sistema
- /home home directories degli utenti
- /usr programmi, librerie, doc. etc… per i programmi user-related

Working Directory. Ogni istanza della shell opera, ad ogni istante, su una directory corrente, o working directory. Subito dopo il login, la working directory è la home directory dell'utente. L'utente può cambiare la working directory con il comando cd (*change directory*)

Home Directory. Ad ogni utente viene assegnata dal system administrator una directory di base (home directory) che ha come nome lo username del proprietario. Ad essa, l'utente potrà aggiungere file (oppure subdir). Per denotare la propria home directory si può usare l'abbreviazione "~".

1.3.5 File Speciali

Un esempio di file speciali sono i devices di I/O. Richieste di lettura/scrittura da/a files speciali causano operazioni di input/output dai/ai devices associati.

Trattamento uniforme di files e devices. I programmi non hanno bisogno di sapere se operano su un file o su un device.

1.4 Protezione di un File

A ciascun file (normale, speciale, directory) sono associati alcuni attributi:

- **Proprietario** (***owner***): l'utente che ha creato il file
- **Gruppo** (***group***): il gruppo a cui il proprietario appartiene
- **Permessi** (***premissions***): il tipo di operazioni che il proprietario, i membri del suo gruppo o gli altri utenti possono compiere sul file

Proprietario, gruppo e permessi iniziali sono assegnati dal sistema al file al momento della sua creazione. Il proprietario può successivamente modificare tali attributi con appositi comandi (chown, chgrp, chmod)

1.4.1 Identificazione Utenti

Un utente si identifica con: Username, Password, ID e gruppo.
Ogni utente viene identificato da uno **user name** assegnato dall'amministratore del sistema. Ad esso corrisponde biunivocamente uno **userid** numerico, assegnato dal sistema. User name e user-id sono **pubblici**.

1.4.2 I gruppi

Ogni utente può far parte di uno o più **gruppi**, definiti dall'amministratore del sistema (senza almeno "sudo" non puoi creare un gruppo).
Ogni gruppo è identificato da un **group name** di al più 8 caratteri, associato biunivocamente a un **group-id** numerico.

1.4.3 I permessi

Ad un file possono essere attribuiti i seguenti permessi: $\left.\begin{matrix}\mathrm{r}: readable \\ \mathrm{w}: writable \\ \mathrm{x}: executable\end{matrix}\right\} per \begin{cases}proprietario \\ gruppo \\ altri\ utenti\end{cases}$

Esempio:

	$\underbrace{\mathrm{r}\ \mathrm{w}\ \mathrm{x}}_{proprietario}$	$\underbrace{\mathrm{r}\ -\ -}_{gruppo}$	$\underbrace{r\ -\ -}_{altri\ utenti}$
In binario:	1 1 1	1 0 0	1 0 0
In ottale:	7	4	4

Alla creazione di un file, Unix assegna i seguenti permessi:

- Per i files ordinari non eseguibili: rw- rw- rw- 110 110 110 6 6 6
- Per i files ordinari eseguibili e per directories: rwx rwx rwx 111 111 111 7 7 7

2. Shell Bash

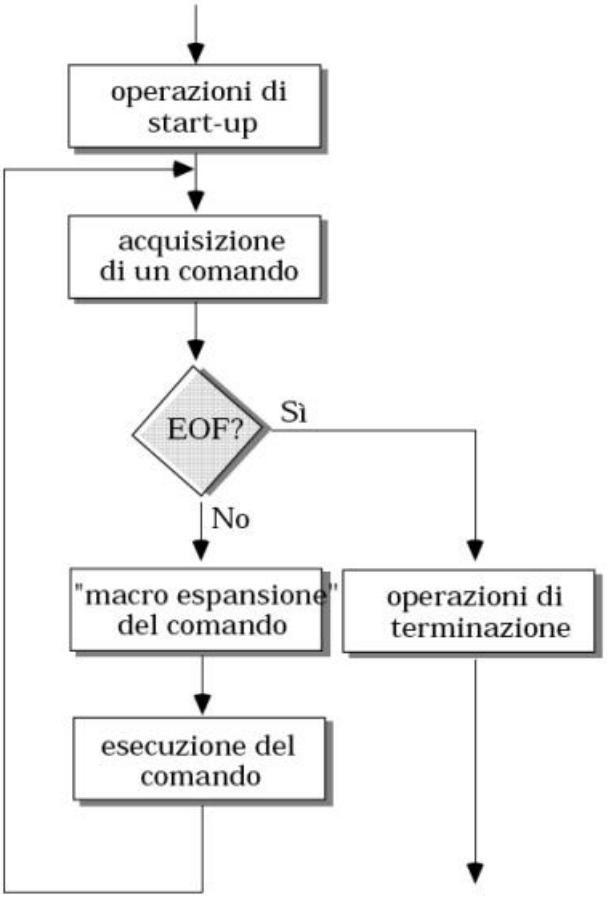
2.1 Shell

Programma che interpreta il linguaggio a linea di comando attraverso il quale l'utente utilizza le risorse del sistema. Permette la gestione di variabili e dispone di costrutti per il controllo del flusso delle operazioni.

Viene generalmente eseguito in **modalità interattiva**, all'atto del login, restando attivo per tutta la durata della sessione di lavoro ed effettuando le seguenti operazioni:

- Gestione del "**main command loop**"
- Analisi sintattica
- Esecuzione di comandi ("**built-in**", file eseguibili) e programmi in linguaggio di shell (**script**)
- Gestione dello **standard I/O** e dello **standard error**
- Gestione dei **processi da terminale**

2.1.1 Ciclo esecuzione Shell



2.1.2 Variabili di shell predefinite

Esistono delle variabili di shell predefinite (variabili di ambiente), che permettono di caratterizzare il comportamento della shell. Per convenzione, il nome di tali variabili è in caratteri **tutti maiuscoli**:

- HOME: argomento di default per il comando cd, inizializzato da login con il path della home directory, letto dal file /etc/passwd
- PATH: il path di ricerca degli eseguibili
- PS1: stringa del prompt, di default "$" per l'utente normale e "#" per il superuser
- HOSTNAME: il nome del computer
- SHELL: la shell corrente
- …

2.1.3 Shell Interattiva

La comunicazione tra utente e shell avviene tramite comandi (built-in oppure file eseguibile) o script (file ASCII presente nel sistema e dotato del permesso di esecuzione).

La sintassi dei comandi è comando $[argomento \ldots]$ dove gli argomenti possono essere:

- **opzioni** o **flag** (-)
- **parametri**

separati da almeno un separatore (di default è il carattere spazio).

Una volta interpretata la prima parola sulla linea di comando, la shell ricerca nel file system un file con il nome uguale a tale prima parola.

La ricerca avviene ordinatamente all'interno delle directory elencate nella variabile d'ambiente PATH

2.1.4 (Ri)definizione di variabili di shell

La shell offre all'utente sia la possibilità di ridefinire alcune variabili d'ambiente, sia di definire delle nuove variabili a proprio piacimento. N.B.: le variabili sono locali alla sessione e muoiono con essa.

Esempio:

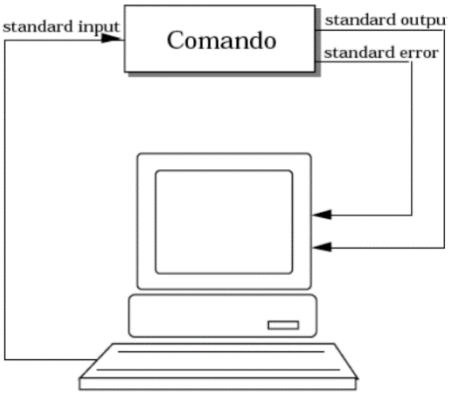
```
~$ frutto=mela
~$ frutto=${frutto}banana
~$ verbo=mangia
~$ nome=Stefania
~$ echo $nome $verbo una $frutto
Stefania mangia una melabanana
~$
```

2.2 File Standard

Normalmente, un programma (comando) opera su più file. In Unix esiste il concetto di **file standard**:

File standard	Che cos'è
standard input	Il file da cui normalmente il programma acquisisce i suoi input
standard output	Il file su cui normalmente un programma produce i suoi output
standard error	Il file su cui normalmente un programma invia i messaggi di errore

2.2.1 Redirezione File



I programmi dispongono di 3 canali di comunicazione:

- Standard input (codice 0), per input (come la tastiera)
- Standard output (codice 1), per output (come lo schermo)
- Standard error (codice 2), per errore

La shell può variare queste associazioni di default **redirigendo** i files standard su qualsiasi file nel sistema.

- **Redirezione standard output**

  comando $argomenti \begin{matrix} > \\ >> \end{matrix} file$

  Redirige lo standard output del comando sul $file$:
  - Se $file$ non esiste, viene creato
  - Se $file$ esiste, viene riscritto (>) oppure il nuovo output viene accodato (>>)

  Esempi:
  - ls -a $> listaFile.txt$
  - echo $PATH $>> listaFile.txt$

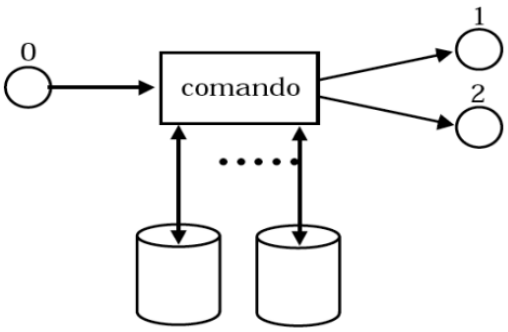
- **Redirezione standard input**
  comando $arg1 \dots argn < file$
  Il file $file$ viene rediretto sullo standard input del comando

- **Redirezione standard error**
- comando $argomenti \begin{matrix} 2> \\ 2>> \end{matrix} file$
  - analogo a $>$ e $>>$
- Esempio:
  - echho "$ciao!$"
    bash: echho: $command\ non\ found$
  - echho "$ciao!$" $2>/\text{dev/null}$

La forma comando $(codiceA)>\&(codiceB)$ redirige il canale A sul canale B
(esempio: comando $> file\ 2>\&1$, in pratica redirigiamo più canali insieme)

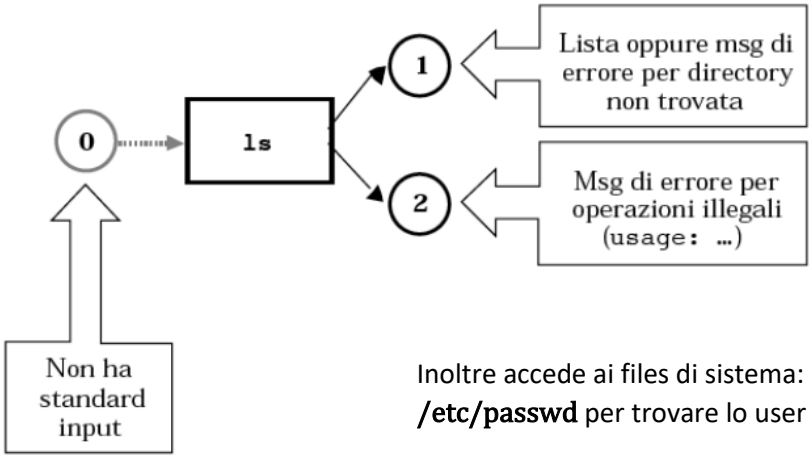
Per redirigere correttamente, è necessario conoscere, di ogni comando:

- come usa lo standard input
- come usa lo standard output
- come usa l'error output
- come usa eventuali altri files



Ricorda: 0 = std input, 1 = std output, 2 = std error

Esempio:



Inoltre accede ai files di sistema:
/etc/passwd per trovare lo user name

2.2.2 Pipe (tubo)

Pipeline di due o più comandi: lo standard output di comando1 funge da input a comando2.

comando1 $[arg \dots]$ | comando2 $[arg \dots] \dots | \dots$

Esempi di comandi concatenabili sono cat, sort, wc: ~$ cat $file$ | sort

La pipe è similare alla redirezione dello standard input ma quest'ultima riceve come argomento un file mentre la pipe un comando.

Esercizi:

1) creare un file che si chiami come l'utente corrente
2) creare un file che si chiami come l'host corrente, e che contenga il nome dell'host corrente

Soluzioni: (1) touch $USER (2) echo $HOSTNAME > ${HOSTNAME}.txt

Comandi concatenabili:

- Solo a **inizio** pipe: echo, ls, e tutti quelli che scrivono su stdout
- Anche al **centro**: wc, sort, uniq, grep, cat, head, tail
    - Se richiamati senza argomenti, leggono da stdin e scrivono su stdout
- Solo a **fine** pipe: less (paginatore interattivo)

Esercizio:

a) Creare una cartella EsercitazioneLSO-1 nella directory di lavoro
b) Creare un file testo chiamato provaFile.txt che contiene username e hostname
c) Creare una variabile che contiene il contenuto di provaFile
d) Aggiungere alla variabile il nome del file
e) Creare un file che contiene il contenuto della variabile

2.3 Command substitution

Il pattern $(comando) viene sostituito con l'output del comando.
Esempi:

- $(ls) equivale a *
- $(echo $ciao$) equivale a ciao
- $(cat $nomefile$) equivale all'intero contenuto del file
- $a = $(ls) assegna ad a l'elenco dei file nella directory corrente
- touch "$(date)" crea un file chiamato come la data attuale

2.4 Metacaratteri

La shell riconosce alcuni caratteri speciali, chiamati **metacaratteri**, che possono comparire nei comandi per definirne il comportamento. Quando l'utente invia un comando, la shell li legge in sequenza alla ricerca di metacaratteri che processa in modo speciale.

Esempio:

```
user> ls *.java

 Albero.java          div.java       ProvaAlbero.java
 AreaTriangolo.java   EasyIn.java    ProvaAlbero1.java
 AreaTriangolo1.java  IntQueue.java
```

Il metacarattere * nel pathname è un'abbreviazione per un nome di file. Il pathname *.$java$ viene espanso dalla shell con tutti i nomi di file che terminano con .$java$. Il comando ls fornisce la lista di tutti i file con tale estensione.

2.4.1 Abbreviazione pathname

I seguenti metacaratteri, detti **wildcard**, sono usati per abbreviare il nome di un file in un path name:

- * stringa di 0 o più caratteri
- ? singolo carattere
- [ ] singolo carattere tra quelli elencati
- { } Stringa tra quelle elencate

Esempi:

```
user> cp /JAVA/Area*.java /JAVA_backup
```

copia tutti i files il cui nome inizia con la stringa `Area` e termina con l'estensione `.java` nella directory `JAVA_backup`.

```
user> ls /dev/tty?
/dev/ttya /dev/ttyb
```

```
user> ls /dev/tty?[234]
/dev/ttyp2 /dev/ttyp4 /dev/ttyq3 /dev/ttyr2 /dev/ttyr4
/dev/ttyp3 /dev/ttyq2 /dev/ttyq4 /dev/ttyr3

user> ls /dev/tty?[2-4]
/dev/ttyp2 /dev/ttyp4 /dev/ttyq3 /dev/ttyr2 /dev/ttyr4
/dev/ttyp3 /dev/ttyq2 /dev/ttyq4 /dev/ttyr3

user> mkdir /user/studenti/rossi/{bin,doc,lib}
crea le directory bin, doc, lib .
```

2.4.2 Quoting

Il meccanismo del **quoting** è utilizzato per inibire l'effetto dei metacaratteri. I metacaratteri a cui è applicato il quoting perdono il loro significato speciale e la shell li tratta come caratteri ordinari.

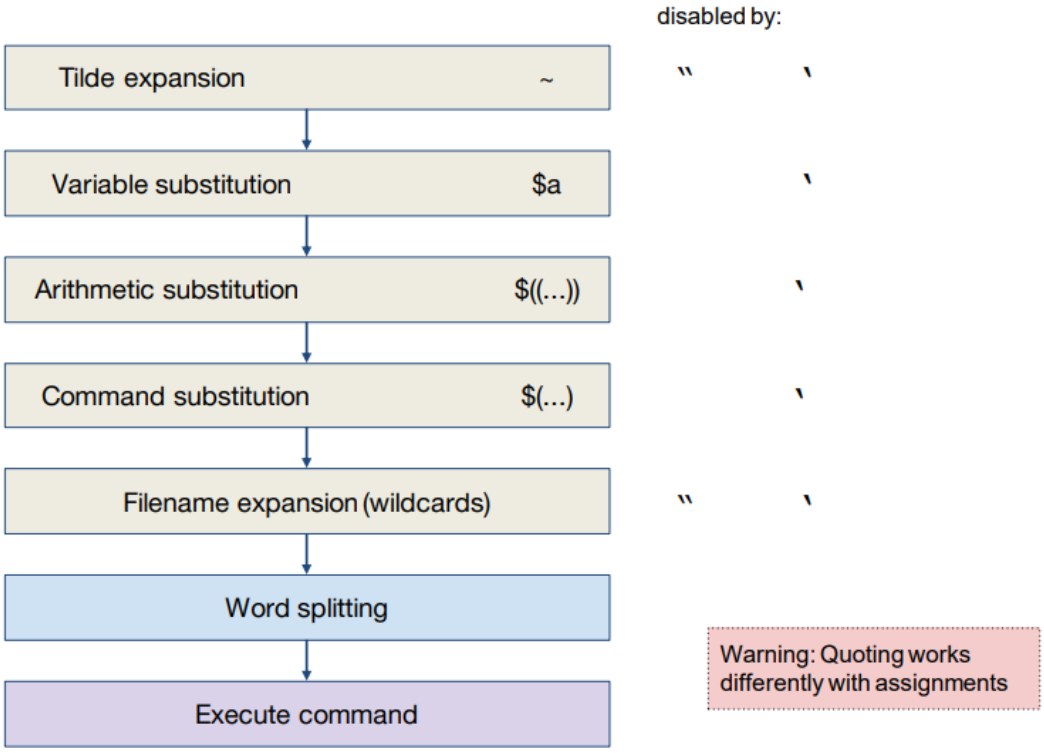
Ci sono tre meccanismi di quoting:

- Il metacarattere di **escape** \ inibisce l'effetto speciale del metacarattere che lo segue:
  ```
  user> cp file file\?
  user> ls file*
  file     file?
  ```
- Tutti i metacaratteri presenti in una stringa racchiusa tra **singoli apici** perdono l'effetto speciale:
  ```
  user> cat 'file*?'
  ...
  ```
- I metacaratteri per l'abbreviazione del pathname presenti in una stringa racchiusa tra **doppi apici** perdono l'effetto speciale (ma non tutti i metacaratteri della shell):
  ```
  user> cat "file*?"
  ```

2.4.3 Metacaratteri di Shell

Simbolo	Significato	Esempio d'uso
>	Redirezione dell'output	ls $>temp$
>>	Redirezione dell'output senza sovrascrizione (append)	ls $>> temp$
<	Redirezione dell'input	wc -l $<text$
< < delim	Redirezione dell'input da linea di comando (here document)	wc -l <<delim
*	Wildcard: stringa di 0 o più caratteri, ad eccezione del punto (.)	ls $*.c$
?	Wildcard: un singolo carattere, ad eccezione del punto (.)	ls $?.c$
[…]	Wildcard: un singolo carattere tra quelli elencati	ls $[a-zA-Z].bak$
{…}	Wildcard: le stringhe specificate all'interno delle parentesi	ls $\{prog, doc\}*.txt$
|	Pipe	ls | more
;	Sequenza di comandi	pwd;ls;cd
||	Esecuzione condizionale. Esegue un comando se il precedente fallisce	cc $prog.c$ || echo $errore$
&&	Esecuzione condizionale. Esegue un comando se il precedente termina con successo.	cc $prog.c$ && $a.out$
(…)	Raggruppamento di comandi	(date;ls;pwd) > $out.txt$
#	Introduce un commento	ls # $lista\ di\ file$
\	Fa in modo che la shell non interpreti in modo speciale il carattere che segue	ls $file.$*
!	Ripetizione di comandi memorizzati nell'history list	! ls

2.4.4 Shell expansions and substitutions



Esercizi:

1) Creare un file che si chiami come l'utente corrente
2) Assegnare alla variabile x l'elenco dei file che cominciano con un punto
3) Scrivere alcune parole nel file nomi.txt. Successivamente, per ogni parola contenuta nel file, creare un file con nome uguale a quella parola

2.4.5 Word splitting

L'ultima fase prima di eseguire un comando consiste nella **suddivisione in parole**.
La variabile IFS (internal field separator) definisce i separatori. Di default: IFS="⟨*space*⟩⟨*tab*⟩⟨*newline*⟩"

Come effetto collaterale, il word splitting sostituisce i *newline* con spazi. Per una migliore comprensione si consiglia di confrontare l'output di ls con quello di echo $(ls) in una directory con molti file.

Esempio:

```
~$ text="1|tom|1982"
~$ IFS='|'
~$ echo $text
1 tom 1982
~$ unset IFS
~$ echo $text
1|tom|1982
```

Esercizio: Usando la virgola come delimitatore, creare un file per ogni elemento separato dal delimitatore usato.

2.5 Esercizi

2.5.1 Esercizi con il comando wc

1) Assegnare alla variabile x il numero di righe di un file a vostra scelta
2) Contare i file della directory corrente che contengono una z nel nome
3) Contare i file nella directory corrente che non contengono una z all'inizio del nome

2.5.2 Esercizi con il comando sort

1) Elencare i file della directory corrente in ordine alfabetico inverso
2) Scrivere nel file "elenco" l'elenco dei file nella directory corrente, in ordine alfabetico

2.5.3 Esercizio con head e tail

Scrivere una combinazione di comandi Unix che consenta di visualizzare:

1) La terza e la quarta riga del file provaFile.txt (soluzione: head -4 $provaFile.txt$ | tail -2)
2) Le penultime 3 righe (quindi senza l'ultima) del file provaFile.txt
3) L'n-esima riga del file provaFile.txt

2.6 Espressioni regolari

Una espressione regolare è un pattern che descrive un insieme di stringhe. L'elemento atomico delle espressioni regolari è il carattere:

- Un carattere è una espressione regolare che descrive se stesso
- L'espressione "a" descrive l'insieme di stringhe $\{a\}$

La maggior parte dei caratteri sono "espressioni regolari" ma possiamo anche inserire i metacaratteri con il carattere di escape "\"

2.6.1 Basic Regular Expressions

- . qualunque carattere (1)
- exp* zero o più occorrenze di exp (2)
- ^exp exp all'inizio del rigo (1)
- $exp$$ exp alla fine del rigo (1)
- $[a\text{-}z]$ un carattere nell'intervallo specificato
- $[^a\text{-}z]$ un carattere fuori dall'intervallo
- \<exp exp all'inizio di una parola (1)
- exp\> exp alla fine di una parola (1)
- $exp\{N\}$ exp compare N volte (1)
- $exp\{N,\}$ exp compare almeno N volte (1)
- $exp\{N,M\}$ exp compare almeno N volte e al più M (1)
- $[[:CLASS:]]$ un carattere in $CLASS$ (1)

Note:
(1) è un carattere normale per Bash
(2) ha un significato diverso per Bash

Le classi di caratteri POSIX:

- $[[:alpha:]]$ I caratteri alfabetici
- $[[:alnum:]]$ I caratteri alfanumerici
- $[[:digit:]]$ Le cifre
- $[[:upper:]]$ I caratteri alfabetici maiuscoli
- $[[:lower:]]$ I caratteri alfabetici minuscoli

Esempi:

- a*b zero o più a seguite da una b
- a.*b una a prima di una b
- \<$[[:upper]]$ una parola che inizia con lettera maiuscola
- ^d la lettera d all'inizio del rigo
- ^a*$ un rigo vuoto o composto solo di a
- ^a.*$b$$ un rigo che inizia con a e finisce con b
- \<.- una parola con un trattino al secondo posto

Molti comandi per l'elaborazione di testi di UNIX (ad esempio grep, ed, sed, etc…) consentono la definizione di **espressioni regolari**, ossia di schemi per la ricerca di testo basati sull'impiego di **metacaratteri**:

- Generalmente, i metacaratteri usati da tali comandi non coincidono con i metacaratteri impiegati dalla shell per identificare i nomi dei file
- Molti caratteri che hanno un significato speciale nelle espressioni regolari hanno pure un significato speciale per la shell

ATTENZIONE: non confondere i metacaratteri di shell con quelli che non lo sono; utilizzare gli apici o i doppi apici per racchiudere le espressioni.

La "concatenazione" di espressioni regolari è una espressione regolare:

- Le "stringhe" possono essere costruite dalla "concatenazione" dei caratteri
- Una stringa corrisponde ("match") ad una concatenazione di stringhe se è composta da due sottostringhe che corrispondono, rispettivamente, alle due espressioni regolari
- "ab" corrisponde alla concatenazione di $exp1 =$ "a" ed $exp2 =$ "b"

L'operatore "|" (Esempio $exp3{=}exp1|exp2$): una stringa corrisponde ad $exp3$ se esiste un match con $exp1$ o con $exp2$.

2.6.2 Extended Regular Expessions

- $exp+$ una o più occorrenze di exp (1: è un carattere normale per Bash)
- $exp?$ zero o una occorrenza di exp (2: ha un significato diverso per Bash)
- $exp1|exp2$ $exp1$ oppure $exp2$ (2)
- (exp) equivale a exp, serve a stabilire l'ordine di valutazione

In **grep**, questi simboli vanno preceduti da backslash: "\".
In **egrep** (*extended grep*), si usano direttamente)

Esempi per grep:

- $[[:digit:]]\+$ una sequenza non vuota di cifre
- $\^a\|b$ un rigo che inizia con a oppure contiene b (precedenza)
- $\^\(a\|b\)$ un rigo che inizia con a oppure con b
- $\(\(\.txt\)\|\(\.doc\)\)\>$ una parola che termina con . txt o con . doc
  In egrep diverrebbe: $((\.txt)|(\.doc))\>$

2.6.3 Esercizi con grep

1) Elencare i file con permesso di esecuzione per il proprietario
2) Elencare le directory il cui nome inizia per maiuscola
3) Elencare i file con permesso di esecuzione oppure di scrittura per il gruppo di appartenenza

Soluzioni:

(1) ls -l | grep ^..x
(2) ls -l | grep ^d.....x
(3) ls -l | grep ^d.....w|.....x

3. Processi Bash

3.1 I processi

I processi sono programmi in esecuzione; lo stesso programma può corrispondere a diversi processi (esempio, tanti utenti che usano emacs). Ogni processo può generare nuovi processi (figli).

Ogni processo ha:

- Process identification number (**PID**)
- Parent process identification number (**PPID**)

Tranne il processo `init`, che ha PID=1 e nessun PPID (la radice della gerarchia di processi è il processo init con PID=1. init è il primo processo che parte al boot di sistema).

Un programma singolo, nel momento in cui viene eseguito, è un **processo**. La nascita di un processo, cioè l'avvio di un programma, può avvenire solo tramite una richiesta da parte di un altro processo già esistente.

Si forma quindi una sorta di gerarchia dei processi organizzata ad albero.
Il processo principale (root) che genera tutti gli altri quello dell'eseguibile **init** che a sua volta attivato direttamente dal kernel.

3.1.1 Tabella processi

Il kernel gestisce una tabella dei processi che serve a tenere traccia del loro stato. In particolare, sono registrati i valori seguenti:

- Il nome dell'eseguibile in funzione;
- Gli eventuali argomenti passati all'eseguibile al momento dell'avvio attraverso la riga di comando;
- Il numero di identificazione del processo;
- Il numero di identificazione del processo che ha generato quello a cui si fa riferimento;
- Il nome del dispositivo di comunicazione se il processo controllato da un terminale;
- Il numero di identificazione;

3.1.2 Attributi dei processi

A ogni processo sono associati due utenti:

- **Real user**: utente che ha lanciato il processo
- **Effective user**: utente che determina i diritti del processo

Quando un processo apre un file, vale l'effective user; di solito i due utenti coincidono, se invece un file eseguibile ha il bit "set user ID" impostato, il corrispondente processo ha:

- **Real user**: utente che ha lanciato il processo
- **Effective user**: utente proprietario dell'eseguibile

Ogni processo ha anche due gruppi associati: real group ed effective group. Un programma con "set user ID" è ping (`ls -l /bin/ping`): ping ha bisogno di partire con permessi di amministratore.

3.1.3 Terminazione di un processo

Per arrestare un processo in esecuzione si può utilizzare:

- la sequenza *Ctrl+C* da terminale stesso su cui il processo è in esecuzione
- il comando `kill` seguito dal PID del processo (da qualsiasi terminale)

3.2 Controllo dei processi

Normalmente, la shell aspetta che ogni comando termini (comando in foreground). Con `comando&`, la shell non aspetta (comando in background) ma il comando può comunque scrivere su standard output

3.2.1 Processi in background

I processi in background sono eseguiti in una sottoshell, in parallelo al processo padre (la shell) e non sono controllati da tastiera. I processi in back ground sono quindi utili per eseguire task in parallelo che non richiedono controllo da tastiera.

3.3 Jobs e Processi

Non si deve confondere un job di shell con un processo. Un comando impartito attraverso una shell può generare più di un processo, per esempio quando viene avviato un programma o uno script che avvia a sua volta diversi programmi, oppure quando si realizzano dei condotti.

Un job di shell rappresenta tutti i processi che vengono generati da un comando impartito tramite la shell stessa.

3.3.1 Controllo dei Job

Un job si può sospendere e poi rimandare in esecuzione:

```
user> cat >temp    # job in foreground

   Ctrl-z   # sospende il job

  [1]+ Stopped

user> jobs
  [1]+ Stopped     cat >temp

user> fg    # fa il resume del job in foreground

   Ctrl-z   # sospende il job

user> bg    # fa il resume del job in background

user> kill %1   # termina il job 1
  [1]+ Terminated
```

3.4 Monitoraggio Memoria

Il comando top fornisce informazioni sulla memoria utilizzata dai processi, che vengono aggiornate ad intervalli di qualche secondo. I processi sono elencati secondo la quantità di tempo di CPU utilizzata.

```
user> top
load averages: 0.68,  0.39, 0.27 14:34:55
245 processes: 235  sleeping, 9 zombie, 1 on cpu
CPU states: 91.9% idle, 5.8% user, 2.4% kernel, 0.0% iowait, 0.0% swap
Memory: 768M real, 17M free, 937M swap in use, 759M swap free
```

Legenda: la prima riga indica il carico del sistema nell'ultimo minuto, negli ultimi 5 minuti, negli ultimi 15 minuti, rispettivamente; il carico è espresso come numero di processori necessari per far girare tutti i processi a velocità massima; alla fine della prima riga c'è l'ora; la seconda contiene numero e stato dei processi nel sistema; la terza l'utilizzo della PU; la quarta informazioni sulla memoria; le restanti righe contengono informazioni sui processi (THR=thread, RES=resident)

3.4.1 Esercizi

1) Stampate il seguente stato interno del processo bash più recente:
   - PID del processo
   - Dimensione della memoria residente
   - Wait channel

- Tempo di CPU utilizzato
- Stato del processo
- Nome del comando

2) Individuate tutti i processi bloccati (stato S) ed in esecuzione (stato R) sulla vostra macchina

Soluzioni:

(1) ps -p $(pgrep -n bash) -o *pid, rss, wchan, cputime, s, comm*

(2) ps -e -l

4. Script

4.1 Script di shell BASH

Uno script di shell BASH è un file di testo che inizia con **#!/ bin/bash** e che ha il permesso di esecuzione. Il resto del file contiene comandi di shell. Non c'è differenza tra quello che si può scrivere al prompt e quello che si può scrivere in uno script.

Esempio di Script:

1) Con un editor di testi (nano, pico, kate, emacs, vim) creare un file con il seguente contenuto:
   ```
   #!/bin/bash
   echo "Hello world!"
   ls
   ```
2) Salvarlo col nome "mio_script"
3) Dargli permessi di esecuzione
4) Eseguirlo digitando "./mio_script"

Perché #!/ bin/bash?

- I primi due caratteri dicono a bash che il file è uno script
- Il resto dice a bash qual è l'interprete per questo script
- **Risultato**: viene invocato l'interprete passandogli come argomento il nome dello script
  Provare con: #!/ bin/echo e con #!/ bin/cat

La Bash è la shell (ovvero l'interfaccia testuale) più diffusa e utilizzata in ambiente Linux, ma non è l'unica (volendo al si può cambiare).

4.1.1 Variabili definite

- $0 Il nome dello script stesso ($argv[0]$)
- $1 … $9 I primi 9 argomenti dello script; chiamerò lo script con $my\_script\ arg1 \dots arg9$
- $# Numero di parametri ricevuti ($argc$)
- $* Tutti i parametri in una stringa singola
- $@ Tutti i parametri in stringhe separate
- $! Process ID (PID) del processo corrente
- $? Exit status dell'ultimo comando eseguito
- $$ Il Process ID dello script corrente
- $USER L'username dell'utente che ha eseguito lo script
- $HOSTNAME L'hostname della macchina che sta eseguendo lo script
- $SECONDS Il numero di secondi passati dall'inizio dello script
- $RANDOM Restituisce un numero casuale
- $LINENO Restituisce il numero della linea corrente

4.1.2 Redirezione standards I/O ed Error

- **STDIN** /proc/<PID>/fd/0 /dev/stdin /proc/self/fd/0
- **STDOUT** /proc/<PID>/fd/1 /dev/stdout /proc/self/fd/1
- **STDERR** /proc/<PID>/fd/2 /dev/stderr /proc/self/fd/2

4.1.3 Exit

Ogni comando restituisce un intero detto $exit\ status$ al chiamante. In C, è l'intero restituito dalla funzione main. Di norma: $0 =$ terminazione regolare mentre un valore diverso da zero $=$ terminazione irregolare.

La variabile di shell "$?" contiene l'exit status dell'ultimo comando eseguito (per mostrarlo: echo $?).

4.1.4 Esercizi

- Scrivere uno script "*eccho*" che prende un argomento e lo stampa due volte
- Scrivere uno script "*bis*" che prende un comando come argomento e lo esegue due volte
- Scrivere uno script che cerca di creare un file all'interno della directory etc (non dovrebbe avere i permessi per poterlo fare) e va a stampare l'errore su schermo.

4.2 Operatori su comandi

In generale è possibile concatenare o effettuare logiche tra i comandi:

- cmd1; cmd2: esegue cmd1 seguito da cmd2
- cmd1 && cmd2 esegue cmd1; poi esegue cmd2 se cmd1 è terminato con successo
- cmd1 | | cmd2 esegue cmd1; poi esegue cmd2 se cmd1 è terminato con errore

In tutti e tre i casi, l'exit status complessivo è quello dell'ultimo comando eseguito.

4.2.1 Il comando if

Il comando if ha la seguente struttura:

```
if comando
then
    lista comandi
[elif comando
  lista comandi]
[else
  lista comandi]
fi
```

N.B: per mettere if e then sulla stessa linea bisogna usare ";"

4.2.2 Espressioni condizionali

Il comando test exp valuta exp come espressione condizionale; cioè, termina con exit status 0 se exp è vera. test exp si può abbreviare con [exp] (gli spazi tra le quadre ed exp sono obbligatori).

Operatori ammessi (per ulteriori informazioni usare "man $test$"):

- su stringhe: $==, != ,$ -z
- su interi: -lt.-le,-eq,-ne,-ge,-gt
- operatori unari su nomi di file: -e,-f,-r,-w,-x

Esempi:

```
if [ -z "$1" ]
then
        echo "Questo script richiede un argomento."
        exit 1
fi
```

```
if [ $# -lt 4 ]
then
        echo "Questo script richiede 4 argomenti."
        exit 1
elif [ ! -e "$1" ]
then
        echo "Il file $1 non esiste."
        exit 1
fi
```

4.2.3 Sostituzioni

$\$((exp))$ valuta exp come espressione aritmetica e il comando $\$((exp))$ viene sostituito dalla shell con il valore di exp. N.B.: solo aritmetica su numeri **interi**.
Se ad esempio abbiamo una variabile $a = 7$ allora $\$((\$a + 1))$ viene sostituita con 8 dalla shell.

Per le sostituzioni aritmetiche abbiamo i seguenti operatori:

- aritmetici: $+, -, /, *, \%$

- elevamento a potenza: **
- bit-a-bit: <<, >>, &, |, ~
- booleani: <, <=, ==, ! =, >, >=, &&, ||, !

4.2.4 Ciclo while

Ripete la lista di comandi fintantoché il comando viene eseguito con successo (come in C)

```
while comando
do
    sequenza
comandi
done
```

Esempio:

```
i=0
while [ $i -lt 10 ]
do
    i=$(( $i+1 ))
done
```

Esercizio: Si realizzi uno script "scriviNumeri.sh" che scrive a video i numeri da 0 a N dove N è passato come input.

Soluzione:

```
#!/bin/bash
idx=0
while [ $idx -lt $1 ]
do
    echo $idx
    idx=$((idx+1))
done
```

4.2.3 Ciclo for

```
for var in lista valori
do
    sequenza comandi
done
```

lista valori è come una lista di argomenti passata a un comando

Ad esempio for a in "uno" "due" "tre" farà tre iterazioni con il valore $a = uno, due, tre$ rispettivamente. Si noti che scrivere for a in "$uno\ due\ tre$" farà un'unica iterazione con $a = uno\ due\ tre$ poiché la *lista valori* verrà giustamente considerata come un unico argomento (essendo un'unica stringa).

4.2.4 Il Case

```
case stringa in
  stringa caso 1) lista di comandi 1 ;;
  stringa caso 2) lista di comandi 2 ;;
 ...
esac
```

Se stringa è uguale a stringa caso 1, allora viene eseguita la lista di comandi 1 ed esce dal costrutto; altrimenti lista di comandi 1 non viene eseguita e passa ad elaborare in modo analogo il caso successivo. Poiché * rappresenta una stringa qualunque, essa può essere utilizzata per rappresentare "tutti gli altri casi".

4.2.5 Until

Il ciclo until esegue la lista di comandi finché la condizione è **falsa**

```
until condition;
do
    comandi
done
```

Alcuni test relativi alle proprietà del file:

- -e file esiste
- -d file directory
- -f il file esiste ed è regolare

Esercizio:
Si realizzi uno script che chiameremo "scriviNumeri.sh" che scrive a video i numeri da 20 a 10.
Soluzione a pagina seguente.

```
#!/bin/bash
idx=20
until [ $idx -lt 10 ]
do
        echo $idx
        idx=$((idx-1))
done
```

4.3 Script interattivi

È possibile creare degli script interattivi grazie all'uso del comando read, questo comando attende l'inserimento di una linea di caratteri da parte dell'utente e assegna la stringa corrispondente ad una variabile di shell.

ESEMPIO: Script "pappagallo"

```
#!/bin/bash
echo "Dimmi qualcosa:"
read cosa
echo "Ti faccio l'eco: $cosa"
```

Utilizzo in shell:

```
~$ ./pappagallo
Dimmi qualcosa:
qualcosa
Ti faccio l'eco: qualcosa
```

4.4 Sed e Awk

sed: editor non interattivo di file di testo
awk: linguaggio per l'elaborazione di modelli orientato ai campi

Condividono una sintassi d'invocazione simile:

- Fanno uso delle espressioni regolari
- Leggono l'input, in modo predefinito, dallo sdin
- Inviano i risultati allo stdout
- Le loro capacità combinate danno agli script di shell parte della potenza di Perl

5. Sed

5.1 Stream EDitor

Sed è un editor di linea che non richiede l'interazione con l'utente e consente di effettuare in modo non interattivo operazioni di sostituzione, cancellazione linee, aggiunta linee e rimpiazzo linee.

Sed può filtrare l'input che riceve da un file o una pipe ma **non** modifica l'input e **non** definisce un output. L'output viene inviato allo standard output e può essere rediretto.

La sintassi è la seguente: sed [-an] command [$file$ …]

- sed legge i file specificati, oppure lo standard input se non specificati i file;
- le linee dell'input sono copiate in un buffer temporaneo chiamato *pattern space*
- modifica l'input come specificato da una lista di comandi:
- l'input è quindi scritto sullo standard output.

Nota: di default ogni linea è replicata sullo standard output dopo l'applicazione dei comandi. Per eliminare questo bisogna usare l'opzione -n.

5.2 Comandi sed

Alcuni comandi:

- a\ "append" di testo al di sotto della riga corrente
- c\ modifica il testo della riga corrente
- d cancella testo
- i\ inserisci testo al di sopra della riga corrente
- p stampa testo
- r legge un file
- s cerca e modifica testo

Operatore	**Nome**	**Effetto**
[$indirizzo$]/p	print	Visualizza l'$indirizzo$ specificato
[$indirizzo$]/d	delete	Cancella l'$indirizzo$ specificato
s/modello1/modello2/	substitute	Sostituisce in ogni riga la prima occorrenza della stringa modello1 con la stringa modello2
[$indirizzo$]/s/modello1/modello2/	substitute	Sostituisce, in tutte le righe specificate in $indirizzo$, la prima occorrenza della stringa modello1 con la stringa modello2
[$indirizzo$]/y/modello1/modello2/	Transform	Sostituisce tutti i caratteri della stringa modello1 con i corrispondenti caratteri della stringa modello2, in tutte le righe specificate da *indirizzo* (equivalente di tr)
g	global	Agisce su tutte le verifiche d'occorrenza di ogni riga di input controllata

Sed permette anche i commenti con il comando "#" (per informazioni sugli altri comandi usare man sed)

La forma del comando sed è la seguente: $[indirizzo]$ function $[argumenti]$

Sed è una macchina a registri:

1) copia ciclicamente una linea di input in un pattern space,
2) applica tutti i comandi con address selezionati dal pattern space,
3) copia il pattern space nello standard output, aggiungendo newline,
4) quindi cancella il pattern space.

5.2.1 $indirizzo$

L'$indirizzo$ ha la seguente forma: $[address[,address]]$.
Esso non è richiesto, ma se specificato deve essere:

1) un numero (che conta linee di input nei file di input);
2) un carattere "$" per l'ultima linea di input; oppure
3) un address di contesto: /regexp/ (espressione regolare preceduta e seguita da un delimitatore).

- Una linea di comando senza $indirizzo$ seleziona ogni pattern space.
- Una linea di comando con un $indirizzo$ seleziona ogni pattern space dato dall'$address$
- Una linea di comando con due $indirizzo$ seleziona il range inclusivo del primo pattern space tra i due $address$ (esempio: 2,6 seleziona dalla linea 2 alla linea 6)

Forniamo di seguito alcuni esempi di indirizzi:

- 8d Cancella l'ottava riga dell'input
- /^$/d Cancella tutte le righe vuote
- 1,/^$/d Cancella dall'inizio dell'input fino alla prima riga vuota compresa
- /Jones/p Visualizza solo le righe in cui è presente "Jones" (con l'opzione -n)
- s/Win/Lin/ Sostituisce con "Lin" la prima occorrenza di "Win" trovata in ogni riga dell'input
- s/H/A/g Sostituisce con "A" tutte le occorrenze di "H" trovate in ogni riga dell'input
- s/ *$// Cancella tutti gli spazi che si trovano alla fine di ogni riga; più precisamente, sostituisce gli spazi che si trovano alla fine di ogni riga con "" (niente).
- s/00*/0/g Riduce ogni sequenza consecutiva di zeri ad un unico zero.
- /GUI/d Cancella tutte le righe in cui è presente "GUI"
- s/GUI//g Cancella tutte le occorrenze di "GUI" (lascia inalterato il resto di ogni riga)

5.2.2 Comportamento di sed

- Se non si specificano azioni, sed stampa sullo standard output le linee in input, lasciandole inalterate
- Se non viene specificato un indirizzo o un intervallo di indirizzi di linea su cui eseguire l'azione, quest'ultima viene applicata a tutte le linee in input
- Gli indirizzi di linea si possono specificare come numeri o espressioni regolari
- Se vi è più di un'azione (comandi multipli), esse possono essere specificate sulla riga di comando precedendo ognuna con l'opzione -e, oppure possono essere lette da un file esterno specificato sulla linea di comando con l'opzione -f.

5.3 sed: esempi

- sed 'd' /etc/services Non visualizza nulla, ma cancella linea per linea il contenuto del file
- sed '1d' /etc/services | more Cancella la prima riga, il resto in stdio
- sed '1,10d' /etc/services | more Cancella le righe tra 1 e 10 in stdio

5.3.1 Comando stampa

```
# Stampa tutte le linee, e ripete quelle che contengono la stringa
~$ sed '/erore/p' esempio
"1 questo e' un esempio"
"2 questa riga contiene un erore"
"3 questa riga non contiene nessun errore"
"4 Oh, ecco un'altro erore"

"2 questa riga contiene un erore"
"4 Oh, ecco un'altro erore"

# Se voglio solo le linee che contengono la stringa:
~$ sed -n '/erore/p' esempio
"2 questa riga contiene un erore"
"4 Oh, ecco un'altro erore"
```

Esercizio:

- usa grep per selezionare tutte le linee che contengono la parola “when” oppure “ When”
- usa sed per selezionare tutte le linee che contengono la parola “when” oppure “ When”

Soluzione: *əlɪf ˌd/uəɳ[ʍʍ]/ˌ u- pəs* *əlɪf ˌuəɳ[ʍʍ]ˌ dəɹɓ*

Extra: un tipico esercizio di esame potrebbe essere del tipo “questo comando fa qualcosa, scrivere l’equivalente con quest’altro comando”.

5.3.2 Comando cancella

```
# Il comando d porta ad escludere linee dalla visualizzazione
~$ sed -n '/erore/d' esempio
"1 questo e' un esempio"
"3 questa riga non contiene nessun errore"

# Esclude le linee che iniziano con una stringa e terminano con un altra
~$ sed -n '/^questa.*erore.$/d' esempio
"1 questo e' un esempio"
"3 questa riga non contiene nessun errore"
"4 Oh, ecco un'altro erore"

#Cancella le righe tra 2 e 4
~$ sed '2,4d' esempio
"1 questo e' un esempio"
```

5.3.3 Ricerca e Sostituzione

```
# Cancella tutte le parole che contengono yourword
~$ sed 's/yourword//g' yuorfile

# Cancella le parole firstword e secondword
~$ sed 's/firstword//g' -e 's/secondword//g' yuorfile

# Sostituisce l'inizio della linea con uno spazio e
# lo ridireziona in file.indent
~$ sed 's/^/ /' file > file.indent

# Sostituisce da riga nulla a riga che inizia con END
# la stringa "hail" con la stringa "spez"
~$ sed '/^$/,/^END/s/hail/spez/g' myfile.txt
```

```
# Sostituisce "/usr/local" con "/usr"; poiché è presente "/"
# nella stringa, si usa al suo posto come separatore ":"
~$ sed -e 's:/usr/local:/usr:g' mylist.txt

# Cancella la parola più lunga che fa il match
# Esempio: myfile.html = "<b>This</b> is what <b>I</b> meant."
~$ sed -n 's/<.*>//g' myfile.html
"This is what I meant."
```

N.B: se aggiungo "g" alla fine allora lo fa con tutte le occorrenze di ogni linea (quindi se si ripete il pattern su una stessa linea fa la sostituzione con tutti); mentre senza "g" lo farà solo con la prima occorrenza di ogni linea (ma comunque di tutte le linee).

5.4 Sed e i gruppi

Spieghiamo si possano sfruttare le regex (regular expression) con il seguente esempio:
sed -rn 's/.*year=([0-9]+).*month=([0-9]+).*/my year: \1, my month: \2/p' $input.txt$

- -rn: extended regular expressions e no elimina ripetizioni. Con questa opzione possiamo risparmiarci un sacco di backslash, altrimenti avremmo dovuto scrivere ad esempio: \(e \).
- .* cattura un gruppo; si può avere un massimo di 9 gruppi e si riferiscono con \1, \2, …, \9. Inoltre con \0 si fa riferimento all'intera linea. Nell'esempio abbiamo due gruppi: .*year=([0-9]+).*month=([0-9]+).* quindi con \1 ci riferiamo a year mentre con \2 ci stiamo riferendo a month.
- La formattazione dell'output sarà: $my\ year$: \1, $my\ month$: \2

Altro esempio: Se volessimo sostituire una stringa "nome cognome luogo" con "Sig nome-cognome Da luogo" serve un'espressione per 3 stringhe separate '.*.*.*'.
Dunque useremo: sed -e 's/\(.*\) \(.*\) \(.*\)/Sig \1-\2 Da \3/' $myfile.txt$
Notare lo spazio tra \) e \(.

5.4.1 Centrare righe di un file

Esempio di script definito come script sed (quindi il terminale sa che per tutti i comandi deve usare l'interprete sed). Capire cosa fanno le varie linee è un buon esercizio (usare man sed per le info).

```
#!/usr/bin/sed -f
# Put 80 spaces in the buffer
1 { x
    s/^$/        /
    s/^.*$/&&&&&&&&/
    x }
# del leading and trailing spaces
 s/^[[:blank:]]*//
 s/[[:blank:]]*$//
#add a newline and 80 spaces to end of line
 G
# keep first 81 chars (80 + a newline)
 s/^\(.\{81\}\).*$/\1/
```

6. AWK

6.1 Aho, Kernighan and Weinberger

AWK è un linguaggio di scripting (fa parte dello standard POSIX):

- gawk è una sua implementazione ben documentata
- strumento ideato per processare file di testo strutturati in "record" (definibili dall'utente), farne report.
- molto usato per script "one liner" (sintassi simil-C)

6.1.1 Elementi di Awk

- La funzione awk cerca su file linee o altre unità di testo che contengono pattern;
- Quando una linea corrispondente ad un pattern, azioni speciali vengono eseguiti sulla linea.
- In awk i programmi sono "data-driven": descrivi cosa cerchi, poi esegui;
- Il programma è definito da un insieme di regole;
- Ogni regola è: azione da fare trovato il pattern.
- awk supporta due tipi di buffers:
  - field buffer: uno per ogni campo nel record corrente (nomi: $1, $2, …, $N,…)
  - record buffer: $0 contiene l'intero record.

6.1.2 Struttura di un programma awk

La sintassi di awk è awk [options] 'script' $file(s)$

Un programma awk è costituito da una sequenza di regole pattern { $action$ }
Dove pattern è uno tra:

- BEGIN prima di processare l'input
- END dopo aver processato l'input
- boolexpr fa match se è vera (es. $1 == "ciao")
- /regexp/ fa match se la regex fa match (es. /^July/

6.1.3 Esempio

Il seguente comando stampa i campi $1 e $3 dal pattern separati da uno spazio:

$$\text{awk} \ \underbrace{-F":"}_{\substack{separatore\\ dei\ campi}} \ ' \ \underbrace{/arun/}_{\substack{patter\\ da\ cercare}} \ \underbrace{\{print\ \$1\ "\ "\ \$3\}}_{\substack{azione\ da\ eseguire\\ se\ il\ pattern\ matcha}} \ ' \ \underbrace{/etc/passwd}_{\substack{file\ su\ cui\\ operare}}$$

6.2 Programmi awk

Ci sono quattro modi per eseguire un programma awk:

- **One-shot**: esegue un breve programma usa e getta
  awk 'program' inputFile1 inputFile2
  … dove program consiste di una serie di pattern e azioni
- **Read Terminal**: non si usano input file ma l'input si scrive direttamente da terminale
  awk 'program' <ENTER>
  $< input\ lines >$
  $< input\ lines >$
  …
  <ctrl-D>
- **Long**: mette programmi awk permanenti in $files$
  awk -f source-file input-file1 input-file2 …
- **Executable Scripts**: realizza programmi awk autonomi

6.2.1 awk ed espressioni regolari

È possibile combinare espressioni regolari e programmi awk utilizzando la seguente sintassi:

awk '<espressione>{<programma>}' <file>

Esempio

```
~$ awk '/dev\/hd/ { print "La partizione :"$1 "\t e usata al "$5}'
La partizione :/dev/hda5 e usata al 63%
```

Il programma viene eseguito solo sulle righe che corrispondono al pattern dell'espressione regolare.

6.2.2 BEGIN ed END

Gli statement BEGIN ed END consentono di eseguire operazioni prima e dopo il corpo del comando.

```
~$ df | awk 'BEGIN {print "Elenco partizioni"} /dev\/hd/ { print
    "La partizione:"$1"\t è usata al "$5} END {print
    "Fine Report\n"}'
Elenco partizioni
La partizione :/dev/hda5    è usata al 63%
Fine Report\n
```

Esempi:

```
# Calcolo della riga più lunga di un file
awk 'BEGIN { max=0 } { if (length($0) > max) max = length($0) }
    END { print max }' data

# Conta i processi appartenenti all'untente tassi
ps aux | awk '/^tassi/ { tot++ } END { print
    "total processes: " tot }'

# Stampa delle potenze del 2 fino a 9
awk 'BEGIN { for  (i=1; i<10; i++) print (2**i) }'
```

6.3 awk scripts

È possibile definire script awk, prendiamo l'esempio usato precedentemente:

```
~$ df | awk 'BEGIN {print "Elenco partizioni"} /dev\/hd/ { print
    "La partizione:"$1"\t è usata al "$5} END {print
    "Fine Report\n"}'
Elenco partizioni
La partizione :/dev/hda5    è usata al 63%
Fine Report\n
```

Questo diventa nello script chiamato "report.awk":

```
BEGIN {print "Elenco partizioni"}
/dev\/hd/ { print "La partizione :"$1 "\t è usata al "$5}
END {print "Fine Report\n"}
```

e si userà come segue: df | awk -f report.awk

6.3.1 awk: le variabili

awk usa molte variabili, alcune editabili, altre read-only.

- La variabile FS (Field Separator) identifica il separatore di input (di defaul spazi o tab)
- La variabile OFS (Output Field Separator) identifica il separatore di output
- La variabile ORS (Output Record Separator) identifica il separatore di "record" in output (default \n)

È possibile modificare il valore di queste variabili: BEGIN { FS=";"; OFS="---"; ORS="->\n<-" }

- NR contiene il numero di record processati (viene incrementata automaticamente
- RS separatore di record
- NF numero di campi nel record corrente

Ogni riferimento ad una variabile non definita comporta la creazione della stessa e la sua inizializzazione a “”. I riferimenti successivi utilizzeranno il valore corrente della stessa.

Esercizio: Dato il file “dati.txt” con il seguente contenuto:

```
100:2:Cliente 1
200:8:Cliente 2
500:2:Cliente 3
```

Calcolare il subtotale per ogni cliente (100*2, 200*8, 500*2) e poi il totale.

Soluzione: awk -f somma.awk dati.txt dove somma.awk avrà il seguente contenuto:

```
BEGIN {
        FS=":";
        print "Calcolo subtotale e totale"
}
{
        subtotale=$1*$2;
        print "Subtotate per "$3" ="subtotale;
        totale=totale+subtotale
}
END {
        print "Totale = "totale
}
```

6.3.2 Output formattato

awk consente di formattare l’output utilizzando la funzione printf (invece della funzione print) che segue la seguente sintassi: printf $formato, item1, item2, \ldots$

Esempio: awk 'BEGIN {w=5; p=3; s="abc"; printf "%d %4.3f %s\n",w,p,s}' il cui output sarà 5 3.000 abc

Nota: Lo statement BEGIN consente di eseguire programmi awk **senza** specificare un input (file o redirezione)

6.3.3 Redirezione in awk

È possibile utilizzare gli operatori di redirezione in script awk

- print items $> nomefile$
  - awk '{ print $2 > "phone-list"; print $1 > "name-list"}' nomefile
- print items $>> nomefile$
- È possibile utilizzare anche l’operatore “<”

Esecuzione di comandi:

```
awk '{
    print $1 > "names.unsorted";
    command = "sort -r > names.sorted">;
    print $1 | command
}' file
```

6.3.4 awk: array

Gli array sono associativi (le chiavi possono essere anche stringhe) e anche multi-dimensionali. Esempi:

```
#esistenza di una chiave
if (2 in array) print array[2]

#assegnamento alla chiave 2 e "mario"
array[2] = "pippo"; array["mario"] = 30

#visita di tutte le coppie chiave, valore
for (i in array) print i, array[i]

#cancellazione di un elemento
delete array[2]

#calcolo della dimensione
length(array)
```

7. Mount, dischi e partizioni

7.1 Montare un Filesystem

Una delle differenze che colpiscono subito chi si trova a usare per la prima volta uno Unix provenendo da sistemi operativi di casa Microsoft è senz'altro il diverso approccio che si ha con i filesystem e con tutti i dispositivi di memorizzazione (hard disk, cdrom, floppy, etc…).

Nello stesso albero delle directory di Linux (e più in generale di tutti gli Unix) i file sono disposti e ordinati in base alla loro funzione (in /boot vi sono i file necessari per il boot, in /home le directory personali dei vari utenti, in /etc i file di configurazione, in /usr i file relativi ai programmi, ecc.). In questo modo ogni supporto non viene visto come un'unità a sé stante (come gli identificativi di unità di Windows), ma si integra col preesistente albero delle directory.

Per poter accedere a queste risorse Linux offre lo strumento **mount** che ci permette di "agganciare" i contenuti di un dispositivo in una directory già esistente (di solito /mnt).

Linux, al contrario di molti altri OS, è in grado di gestire tutti i tipi di file system più comuni, dalla tradizionale FAT di Windows 9x a NTFS di Windows NT, dalle partizione BSD a quella di Solaris, etc…
Tutti i **device di Linux** sono identificabili tramite una voce nella directory /dev.

Gli hard disk **EIDE** si indicano con /dev/hdxy dove al posto della x c'è una lettera che indica il canale e se è master o slave (ad esempio /dev/hda indica il Primary Master, /dev/hdd il Secondary Slave).

- /dev/hda1 indica la prima partizione primaria sul Primary Master,
- /dev/hda5 la prima partizione logica sul Primary Master
- etc…

Per avere le idee più chiare sulla situazione dei nostri hd può risultarci comodo l'uso del comando dmesg | grep hd. Invece, per controllare che partizioni sono montate sulla nostra Linux Box diamo il comando mount senza opzioni.

7.1.1 I filesystem montabile

mount -t $tipodifs$ -o $opzione, altraopzione\ filesystem$ mountpoint

Se ad esempio volessimo montare un fs di tipo FAT32 presente sulla partizione /dev/hda1 in /mnt/windows scriveremo (da root): mount /dev/hda1/mnt/windows

Per "smontare" il nuovo filesystem possiamo usare il comando umount: umount /dev/hda1
Potendo mettere come parametro o il device o il mount point. È bene precisare che la directory /mnt/windows deve essere già esistente.

In questo caso mount è riuscito a determinare da solo di che filesystem (fs) si tratti. È buona cosa però passare come parametro il tipo di fs tramite l'opzione -t $tipo\_di\_fs$

ext2 per una ext2 (Linux),
reiserfs per reiserfs (Linux),
ext3 per ext3 (Linux),
vfat per FAT (Windows 9x),
msdos per MS-DOS (DOS),
ntfs per NTFS (Windows NT),
iso9660 per ISO9660 (CDROM),
hpfs per hpfs (OS/2),
hfs per HFS (Macintosh), ecc.).

7.1.2 Opzioni di un filesystem

Queste opzioni vanno espresse antecedendole con un -o e separandole con una virgola.
Le più significative sono:

- **rw** Monta il filesystem in lettura e scrittura (opzione messa di default)
- **ro** Monta il filesystem in sola lettura
- **exec** Permette l'esecuzione di file
- **noexec** Non permette l'esecuzione di file
- **remount** Rimonta un fs (normalmente utilizzato per cambiare delle opzioni a un filesystem già montato)

Per quel che riguarda **ext2** (e **ext3** che è una sua evoluzione) è da segnalare:

- **errors=remount -ro** In caso di errore rimonta il filesystem in sola lettura (in modo da dare l'opportunità di correggere gli errori). A differenza di ext2, l'fs di tipo **FAT** (comprendente vfat, msdos e umsdos), non dispone di una serie di caratteristiche che permettono di impostare tra le altre cose l'owner, il gruppo e i premessi dei file. Per permettere di impostare questi parametri anche per questo tipo di partizioni ci vengono incontro delle opzioni di mount. Le opzioni che seguono valgono anche per ntfs.
- **umask** Con umask=$valore$ potremo impostare quali permessi NON attribuire a tutti i file. I permessi vanno indicati nella forma ottale. Ad esempio umask=000 assegnerà a tutti i file i permessi di scrittura, lettura ed esecuzione per tutti gli utenti. Se viene utilizzata questa opzione, l'opzione noexec o exec viene ignorata.
- **uid** Con uid=$valore$ è possibile impostare il proprietario dei file e delle directory. Di default viene impostato l'utente root.
- **gid** Equivale a uid per quel che riguarda il gruppo (anziché il proprietario). Di default viene utilizzato il gruppo con gid=0 (di solito root)

Esempi:

- Montare la partizione /dev/sda1 sulla directory /mountdir (directory mountdir che va creata qualora non esista): mount/dev/sda1/mountdir
- Rimontare un filesystem già in uso cambiando però le opzioni (ad esempio in lettura/scrittura rw oppure in sola lettura r): mount -o remount,rw /dev/sda
- Montare una immagine ISO di un cdrom oppure di un DVD senza la necessità di dover masterizzare un supporto fisico: mount -o loop /home/image.iso/media/cdrom

7.1.3 FSTAB

È anche possibile impostare quali filesystem il sistema dovrà montare all'avvio del sistema. Per far questo dovremo editare il file /etc/fstab, file che contiene le informazioni dei filesystem presenti sul sistema.
$\langle file\ system\rangle\langle mount\ point\rangle\langle tipo\ di\ partizione\rangle\langle opzioni\rangle\langle dump\rangle\langle ordine\ per\ fsck\rangle$

- $\langle file\ system\rangle$ dovremo mettere il device della partizione da montare, sostituendo
- $\langle mount\ point\rangle$ con la directory dove verrà innestato il nuovo fs,
- $\langle tipo\ di\ partizione\rangle$ con il tipo di partizione (ext2, ext3, vfat, iso9660, ecc.),
- $\langle opzioni\rangle$ con le opzioni (quelle di mount) che vorremo mettere.
- $\langle dump\rangle$, senza soffermarci troppo nei dettagli, una buona scelta è mettere 0.
- Il campo $\langle ordine\ per\ fsck\rangle$ indica l'ordine con cui verrà eseguito fsck (per il controllo di settori danneggiati o di filesystem inconsistente) al boot. È bene mettere 1 per il filesystem con / come mount point 2 per le altre partizioni Linux (ext3, ext2 o reiserfs), 0 per tutti gli altri filesystem (partizioni di Windows, floppy, cdrom, ecc).

Alcune delle opzioni di FSTAB sono:

- noauto per non montare il dispositivo (ad esempio un cdrom) al boot ma per rendere immediato il mount con mount /mountpoint oopure mount /dev/device
- users per permettere a ogni utente di montare o smontare un dispositivo
- user per permettere a ogni utente di montare (ma non smontare) un dispositivo (può smontarlo solo chi lo ha montato)

7.2 Disks, Slices, Partitions, and Volumes

Ogni disco rigido è generalmente suddiviso in una seria di unità separate di dimensioni diverse chiamate **partizioni** o **sezioni**. Ogni disco contiene una qualche forma di tabella delle partizioni, chiamata VTOC (Volume Table Of Contents), che descrive dove iniziano le sezioni e quale è la loro dimensioni.

Ciascuna sezione può quindi essere utilizzata per archiviare informazioni di bootstrap, un file system, spazio di swap o essere lasciata come partizione grezza per l'accesso al database o altro uso.

7.2.1 Le partizioni

Il partizionamento divide un'unità disco in uno o più dischi logici. Ogni partizione viene trattata come un disco separato con il proprio file system.

Le informazioni sulle partizioni vengono archiviate in una tabella delle partizioni.
Esistono due tipi di partizioni che possono essere create utilizzando l'utilità fdisk:

- **Partizioni primarie**
- **Partizioni estese**

Lo schema di partizionamento originale per i dischi rigidi dei PC consentiva solo quattro partizioni, chiamate partizioni primarie. Per creare più di quattro partizioni, una di queste quattro partizioni può essere divisa in tante partizioni più piccole, chiamate partizioni logiche. Quando una partizione primaria è suddivisa in questo modo, è nota come partizione estesa.

Windows e Linux usano differenti identificazioni. Windows segna le partizioni con lettere e le chiama unità (non necessariamente corrispondono a unità fisiche). Linux usa tre lettere e una cifra per la identificazione, iniziando con 'h' per IDE ed ' s' per unità SCSI/SATA. La terza lettera segna il numero dell'unità, come vista dal BIOS, con a-d per primaria/secondaria master/slave per unità IDE ed un numero illimitato per unità SCSI/SATA, basato sulle limitazioni del controller. La cifra si riferisce al numero delle partizioni.

I numeri 1-4 sono usati per denominare le partizioni primarie, una delle quali può essere una partizione estesa, un contenitore di partizioni logiche.
Le partizioni logiche sono sempre numerate a partire da 5 e oltre. Fisicamente le partizioni logiche possono essere minori del loro numero, dipende dal numero di partizioni primarie presenti nel disco.
Le partizioni sono contate separatamente per ogni disco fisico, come riconosciuto dal sistema.

Le eccezioni sono le configurazioni RAID ed LVM.

Per visualizzare il disco e le partizioni su linux si può usare il comando fdisk -l.

7.2.2 Tipi di tabelle di partizioni

La tabella delle partizioni è una tabella di quattro elementi contenuta nel master boot record di un disco rigido. Ciascun elemento di questa tabella occupa 16 byte e contiene le seguenti informazioni relative a una partizione del disco:

1) flag partizione attiva: specifica se da questa partizione dev'essere caricato il nucleo del sistema operativo (1 byte),
2) indirizzo in formato CHS (Cylinder-Head-Sector) del primo settore della partizione (3 byte),

3) tipo di partizione: specifica se la partizione è primaria, estesa, nascosta, di ripristino o vuota, ed eventualmente, il tipo di file system utilizzato (1 byte),
4) indirizzo in formato CHS dell'ultimo settore della partizione (3 byte),
5) numero di settori del disco che precedono la partizione (4 byte), 6. numero di settori del disco che compongono la partizione (4 byte).

GUID Partition Table (GPT) è parte dello standard Extensible Firmware Interface (EFI). L'EFI utilizza il GPT laddove il BIOS utilizza il **Master Boot Record** (MBR).

MBR è composto da: **MBP** (Master Boot Program) è un codice eseguibile che si trova nei primi 446 byte del Master Boot Record; **MBT** (Master Boot Table) è una tabella che si trova subito dopo l'MBP, di è 64 byte; gli ultimi 2 byte sono riservati al magic number dell'MBR, che identifica la fine dello stesso.

GPT utilizza l'indirizzamento a blocchi logici.

7.2.3 Vantaggi di più partizioni

Fondamentalmente Linux è in grado di funzionare correttamente con una partizione, quella di Root, ed una partizione di Swap per la memoria virtuale.

Per un uso domestico, questo schema è decisamente consigliabile; qualora però si volesse reinstallare Linux, tutti i dati in esso contenuti andrebbero perduti. Per questo possiamo creare delle partizioni dedicate per quei files a noi più importanti come i dati personali dei nostri utenti, o tutta la posta e le code di stampa. Reinstallando Linux e, montando la partizione nel filesystem, non si perderebbero i dati.

Per ottenere maggiori prestazioni, si consiglia di creare le partizioni in un ordine logico ossia partendo dalle partizioni con una maggior frequenza di accesso. Un esempio di partizionamento per un server potrebbe essere, nell'ordine, Swap, /usr, /var, e /home.

Il vantaggio di avere filesystem su partizioni separate è che le diverse parti del sistema operativo sono in qualche modo protette le une dalle altre.
Se i tuoi utenti hanno riempito /home, i programmi che scrivono file di registro in /var non sono interessati se /home e /var sono partizioni separate.
Se il tuo disco viene danneggiato, solo la partizione danneggiata sarà danneggiata.
Lo svantaggio è che, nella maggior parte dei casi, se erroneamente hai allocato troppo poco spazio su disco per una partizione, non puoi rubare spazio da /var per avere più spazio su /home una volta configurato il sistema.

Ora alcuni importanti consigli da ricordare quando si ha a che fare con le partizioni:

- Windows richiede partizioni primarie.
- BSD e Solaris richiedono anch'essi partizioni primarie.
- Linux non richiede partizioni primarie e può essere installato su quelle logiche.
- Installate sempre prima i sistemi che richiedono partizioni primarie.
- Create sempre prima le partizioni prima di installare un sistema, pensando bene alle vostre esigenze.
- Non dimenticate le limitazioni di dimensione per filesystems vecchi (come FAT32)

7.2.4 Swap

Unix supporta la memoria virtuale, permettendo cioè di utilizzare spazio su disco come se fosse un'espansione di memoria RAM e aumentando di conseguenza la dimensione massima di memoria utilizzabile.

In linea di massima è consigliabile avere una memoria di Swap doppia rispetto alla RAM presente.

7.2.5 Linux

Un'installazione tipica di Linux (compreso di server X e un ambiente grafico completo) occupa intorno agli 800 MB più la partizione di swap. Le pretese si fanno più modeste se non si preferiscono i classici (e più snelli Windows Manager) ai più diffusi KDE e GNOME.

Riguardo al **tipo di partizioni** da utilizzare (primarie o logiche) consiglio di creare una partizione per il boot come primaria, utilizzare le restanti primarie disponibili, per poi allocare il nostro spazio rimanente in partizioni logiche all'interno di una estesa.

Creare:

1) una partizione primaria per /boot di 15 mb (attenzione che vecchie versioni di LILO non permettono il boot da partizioni che superano il 1024 cilindro per una limitazione del BIOS, ciò non È più un problema dalla versione 21.3),
2) una per /home (siate generosi con questa partizione se avete molto spazio),
3) una per /usr (anche questa merita una particolare attenzione, diciamo che un paio di GB sono abbondanti per ogni uso),
4) una per / dove risiederanno tutti i restanti file (un GB È più che sufficiente). Sulla dimensione della partizione di swap in genere ci si regola assegnando il doppio della RAM presente sul nostro PC (ma non superate i 128 MB).

Per la grandezza di ogni partizione dipenderà dalle vostre esigenze:

- se per esempio prevedete di creare molti account sul vostro PC, o di utilizzarlo in rete come server rendendo disponibile dello spazio web o ftp, allora conviene avere una partizione per le home directory abbastanza grande,
- mentre se pensate di installare molti programmi dovrete avere un occhio di riguardo per la partizione con mount point in /usr

8. VM e Docker

8.1 La virtualizzazione

Per virtualizzazione si intende la creazione di una versione virtuale di una risorsa normalmente fornita fisicamente. Qualunque risorsa hardware o software può essere virtualizzata: sistemi operativi, server, memoria, spazio disco, sottosistemi.

È quindi possibile seguire, su una sola macchina (detta sistema host), altri sistemi operativi (detti sistemi guest) simultaneamente, ad esempio mettendo a disposizione un sistema Windows e un sistema Linux contemporaneamente.

Dato un sistema caratterizzato da un insieme di risorse (hardware e software), **virtualizzare il sistem** significa presentare all'utilizzatore una visione delle risorse del sistema diversa da quella reale.
Ciò si ottiene introducendo un **livello di indirezione** tra vista logica (sistema virtuale) e quella fisica delle risorse (sistema reale: risorse HW/SW).

8.1.1 Esempi di virtualizzazione

- **Astrazione**: in generale un oggetto astratto (risorsa virtuale) è la rappresentazione semplificata di un oggetto (risorsa fisica):
    - esibendo le proprietà significative per l'utilizzatore
    - nascondendo i dettagli realizzativi non necessari.

  Es.: tipi di dato vs. rappresentazione binaria nella cella di memoria.
  Il **disaccoppiamento** è realizzato dalle operazioni (interfaccia) con le quali è possibile utilizzare l'oggetto.
- **Linguaggi di programmazione**. La capacità di portare lo stesso programma (scritto in un linguaggio di alto livello) su architetture diverse è possibile grazie alla definizione di una macchina virtuale in grado di interpretare ed eseguire ogni istruzione del linguaggio, indipendentemente dall'architettura del sistema (sistema operativo e hardware):
    - interpreti (esempio Java Virtual Machine)
    - compilatori
- **Virtualizzazione a livello di processo**. I sistemi multitasking permettono la contemporanea esecuzione di più processi, ognuno dei quali dispone di una macchina virtuale (CPU, memoria, dispositivi) dedicata. La virtualizzazione è realizzata dal **kernel** del sistema operativo.
- **Virtualizzazione di sistema**. Una singola piattaforma hardware viene condivisa da più sistemi operativi, ognuno dei quali è installato su una diversa macchina virtuale.
  Il disaccoppiamento è realizzato da un componente chiamato VMM (*Virtual Machine Monitor*) o *hypervisor*) il cui compito è consentire la condivisione da parte di più macchine virtuali di una singola piattaforma hardware. Ogni macchina virtuale è costituita oltre che dall'applicazione che in essa viene eseguita, anche dal sistema operativo utilizzato.
  Il VMM è il mediatore unico nelle interazioni tra le macchine virtuali è l'hardware sottostante, che garantisce **isolamento** tra le VM e **stabilità** del sistema.

8.1.2 Tipologie di virtualizzazione

Hardware

- Questo concetto fa riferimento a tecnologie che consentono la fornitura di componenti hardware, facendo ricorso a soluzioni software, e questo indipendentemente da quella che è la loro base fisica.
- Una VM (virtual machine), altro non è che un computer virtuale che con l'utente finale si comporta esattamente come farebbe un computer fisico dotato di un suo hardware e relativo sistema operativo. Le VM dunque, operano come "sistemi guest virtuali" su uno o più sistemi fisici, denominati host.

- L’hypervisor stabilisce il livello di astrazione tra il sistema virtuale e la base fisica. Gli Hypervisor sono dei software che hanno la funzione di gestire le risorse hardware a disposizione: Cpu, Ram, periferiche, memoria di archiviazione ed altre, condividendole con più sistemi guest.

Virtualizzazione completa
- Attraverso questa forma di virtualizzazione, l’hypervisor di ogni macchina virtuale riproduce integralmente un ambiente hardware. In questa maniera, ogni VM ha un suo quantitativo di risorse hardware rese disponibili dall’hypervisor, e può di conseguenza eseguire delle applicazioni.
- L’hardware fisico di quello che è il sistema host invece, rimane del tutto nascosto all’S.O. guest
- L’approccio permette di fatto il funzionamento di sistemi guest che non sono modificati.

Paravirtualizzazione
- Con la virtualizzazione completa viene fornito una soluzione hardware virtuale per ciascun VM, mentre con la paravirtualizzazione l’hypervisor rende disponibile solamente un’API, ovvero un’interfaccia di programmazione. Questo permette ai sistemi operativi guest di poter accedere all’hardware fisico di quello che è il sistema host.
- In termini di prestazioni assolute dunque, la paravirtualizzazione offre dei vantaggi rispetto alla virtualizzazione completa.
- Dal punto di vista dell’utente finale, una virtual machine non presenta differenze di sorta rispetto ad un computer fisico. In altre parole, la virtualizzazione hardware fa riferimento alla possibilità di scegliere differenti server virtuali per più utenti sulla base di una piattaforma di elaborazione molto potente: il popolare concetto di hosting condiviso.

8.1.3 Obiettivi e vantaggi della virtualizzazione

Obiettivo: disaccoppiare il comportamento delle risorse hardware e software di un sistema di elaborazione, così come viste dall’utente, dalla loro realizzazione fisica.

Le motivazioni che spingono alla virtualizzazione di un sistema sono diverse, a seconda di chi si appresta a virtualizzare un sistema.
Gli sviluppatori hanno la possibilità di creare degli ambienti di prova nei quali far girare i loro software senza intaccare l’integrità del loro sistema, racchiudendo quindi i loro test all'interno di un'area controllata e priva di rischi.
Nell'ambito server è possibile avere server diversi, che offrono servizi diversi, sulla medesima macchina fisica, risparmiando sull'acquisto di nuove macchine serve.

I vantaggi della virtualizzazione sono i seguenti.
- **Uso di più SO sulla stessa macchina fisica**: più ambienti di esecuzione (eterogenei) per lo stesso utente:
    - legacy systems
    - Possibilità di esecuzione di applicazioni concepite per un particolare sistema operativo.
- **Isolamento degli ambienti di esecuzione**: ogni macchina virtuale definisce un ambiente di esecuzione separato (sandbox) da quelli delle altre:
    - possibilità di effettuare testing di applicazioni preservando l’integrità degli ambienti e del VMM
    - Sicurezza: eventuali attacchi da parte di malware o spyware sono confinati alla singola macchina virtuale.
- **Consolidamento HW**: possibilità di concentrare più macchine (ad esempio i server) su un’unica architettura HW per un utilizzo efficiente dell’hardware (es. server farm):
    - Abbattimento costi hardware
    - Abbattimento costi amministrazione
- **Gestione facilitata delle macchine**: è possibili effettuare in modo semplice:
    - la creazione di macchine virtuali (virtual appliances)

- l'amministrazione di macchine virtuali (reboot, ricompilazione kernel, etc…)
- migrazione "a caldo" di macchine virtuali tra macchine fisiche:
  - possibilità di manutenzione hardware senza interrompere i servizi forniti dalle macchine virtuali
  - disaster recovery
  - workload balancing: alcuni prodotti prevedono anche meccanismi di migrazione automatica per far fronte in modo "automatico" a situazioni di sbilanciamento.

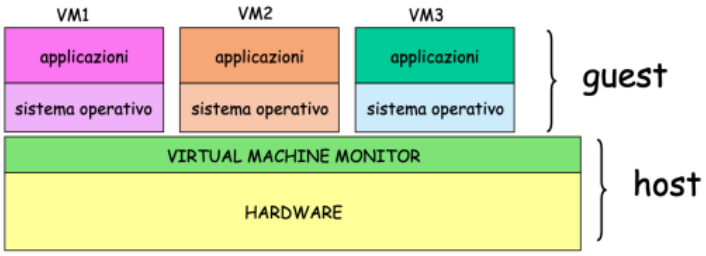
8.1.4 Virtual Machine Monitor

In generale, il VMM deve offrire alle diverse macchine virtuali le risorse (virtuali) che sono necessarie per il loro funzionamento: CPU, Memoria, Dispositivi di I/O.

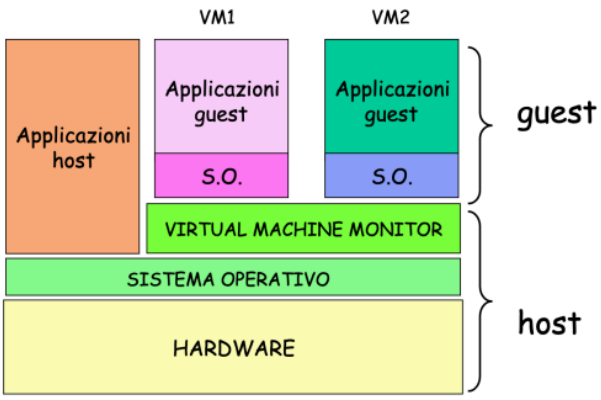
- **Livello** dove è collocato il VMM:
  - VMM di sistema: eseguono direttamente sopra l'hw dell'elaboratore (es. vmware esx, xen)
  - VMM ospitati: eseguiti come applicazioni sopra un sistema operativo esistente (es. vmware player, parallels, virtualPC, virtualbox, UserModeLinux)
- **Modalità di dialogo** per l'accesso alle risorse fisiche tra la macchina virtuale ed il VMM:
  - **Virtualizzazione completa** (vmware): le machcine virtuali usano la stessa interfaccia (istruzioni macchina) dell'architettura fisica.
  - **Paravirtualizzazione** (xen): il VMM presenta un'interfaccia diversa da quella dell'architettura hardware.

VMM di sistema vs. VMM ospitati:

- VMM di Sistema (esempi: vmware esx, xen, kvm).
  - Le funzionalità di virtualizzazione vengono integrate in un sistema operativo leggero, costituendo un unico sistema posto direttamente sopra l'hardware dell'elaboratore.
  - È necessario corredare il VMM di tutti driver necessari per pilotare le periferiche.
  - **Host**: piattaforma di base sulla quale si realizzano macchine virtuali. Comprende la macchina fisica, l'eventuale sistema operativo ed il VMM.
  - **Guest**: la macchina virtuale. Comprende applicazioni e sistema operativo.



- VMM ospitato (prodotti: User Mode Linux, VMware Server/Player, Virtual Box, Parallels)
  - Il VMM viene installato come un'applicazione sopra un sistema operativo esistente, che opera nello spazio utente e accede l'hardware tramite le system call del sistema operativo su cui viene installato.
  - Più semplice l'installazione (come un'applicazione).
  - Può fare riferimento al sistema operativo sottostante per la gestione delle periferiche e può utilizzare altri servizi del sistema operativo (es. scheduling, gestione delle risorse).
  - Peggiore la performance.



Per visualizzare dettagli su come installare e configurare la VM VirtualBox andare a questo link.

8.1.5 VPN e virtualizzazione

Un ultimo riferimento sulle tipologie di virtualizzazioni, che ricordiamo sono davvero numerose, riguarda le reti. Sempre più diffuse sono infatti le **VPN**, ovvero le Virtual Private Network, reti virtuali basate su reti fisiche.

Le VPN vengono utilizzate per la realizzazione di connessioni sicure. Un esempio può essere quello del collaboratore che necessita di accedere da una postazione esterna alla rete aziendale. Dato che internet è una rete pubblica, la protezione dei tuoi dati non è garantita, in questo caso si consiglia la virtualizzazione.

Molte aziende produttrici di software offrono soluzioni di virtualizzazione tramite i processi di crittografia e autenticazione. In questo modo la connessione avviene in una rete privata.

8.1.6 Proprietà

- **Isolation**: hypervisor fanno un ottimo lavoro isolando il guest dall'host, quindi si possono utilizzare le macchine virtuali per eseguire software difettoso o non attendibile in modo ragionevolmente sicuro.
- **Snapshots**: è possibile scattare "istantanee" della tua machcina virtuale, catturando l'intero stato della macchina (disco, memorai, etc…), apportare modifiche alla tua macchina e quindi ripristinare uno stato precedente. Ciò è utile, tra le altre cose, per testare azioni potenzialmente distruttive.
- **Resources**: condiviso con la macchina host; tenerne contro quando si assegnano le risorse fisiche.
- **Networking**: molte opzioni, il NAT predefinito dovrebbe funzionare correttamente per la maggior parte dei casi d'uso.
- **Guest addons**: molti hypervisor possono istanllare software nel guest per consentire una migliore integrazione con il sistema host.

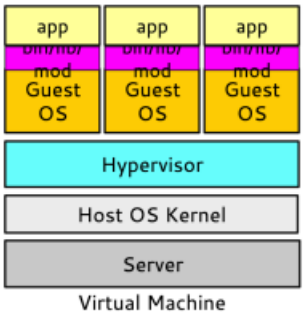
8.2 Docker

Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker raccoglie il software in unità standardizzate chiamata **container** (possiamo considerarli come versioni leggere di virtual machine) che offrono tutto il necessario per la loro corretta esecuzione, incluse le librerie, strumenti di sistema, codice e runtime.

I Docker containers forniscono un ambiente contenuto nella macchina locale. Tutti gli studenti hanno lo stesso ambiente e non possono "romperti" il tuo computer. I Docker containers sono leggeri, portabili e hanno un avvio rapido.

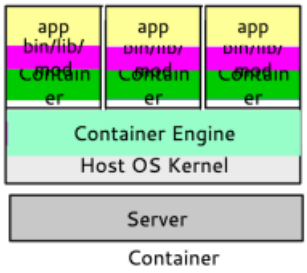
8.2.1 Containers vs VMs

- Le macchine virtuali possono essere eseguite su differenti sistemi operativi.
- **Container Engine**: livello di astrazione e isolamento tra il sistema operativo e l'ambiente delle applicazioni. Abilita e disabilita i containers. Pacchetti di applicazioni e dei loro ambienti.
- I containers vengono virtualizzati a livello del sistema operativo, con più containers eseguiti direttamente sul kernel del sistema operativo.
- I containers sono molto più leggeri: condividono il kernel del sistema operativo, l'inizializzazione è molto più veloce, e usano una frazione della memoria rispetto al booting dell'intero sistema operativo.



Virtual Machine

- I dockers containers sono tipicamente **stateless** (la memoria usata dal container viene cancellata quando il container viene arrestato)!
- Docker usa la virtualizzazione a livello del sistema operativo, quindi i containers condividono l'host del sistema operativo senza il bisogno di un hypervisor. Questo significa che i containers sono tipicamente più leggeri e veloci di una macchina virtuale.
- La piattaforma Docker può eseguire più containers contemporaneamente e facilita la comunicazione tra loro.



- I registri online Docker consentono una semplice implementazione di immagini Docker.

8.2.2 Come funziona un container?

- **Client**: il modo in cui interagisci con Docker. Nel nostro caso sarà la linea di comando.
- **Daemon**: un servizio in background che ascolta le richieste API e gestisce gli oggetti Docker.
- **Docker Desktop**: un'applicazione facile da installare che include un client (sua CLI che GUI), un daemon, e altri servizi.
- **Docker registry**: un luogo dove le immagini docker sono memorizzate. Noi useremo Docker Hub, un registro pubblico che viene configurato in Docker da default.
- **Docker Objects**:
    - Un'**immagine** è un modello di sola lettura per un container.
    - Un **container** è un'istanza eseguibile di un'immagine che può essere avviata, arrestata etc…
- **Namespace**: avvolge una risorsa globale di sistema in una astrazione che fa sembrare ai processi all'interno del namespace come se avessero la loro privata istanza di risorsa globale. I cambiamenti alla risorsa globale sono visibili a tutti i processi all'interno del namespace, ma sono invisibili agli altri.
- **Cgroups**: limita, controlla e contabilizza l'utilizzo delle risorse per una serie di processi.

8.2.3 Quali sono i vantaggi?

- **Isolation**: i containers virtualizzano CPU, memoria, e risorse network al livello SO; fornendo agli sviluppatori una sandboxed view del sistema operativo isolato dalle altre applicazioni. Gli sviluppatori, usando i containers, sono capaci di creare ambienti prevedibili isolati dalle altre applicazioni.
- **Productivity enhancement** (miglioramento della produttività): i containers possono includere dipendenze software che servono alle applicazioni (versioni specifiche di linguaggi di programmazione, librerie software) garantendo di essere consistente a prescindere dalla distribuzione dell'appllicazione. Di conseguenza gli sviluppatori e i team operativi IT spendono meno tempo con il debugging e le diagnosi in ambienti differenti.
- **Deployment simplicity**: i containers consentono di creare un package per l'intera applicazione, astraendo il sistema operativo, la macchina e persino il codice stesso, quindi lo sviluppo e la distribuzione sono semplificati poiché i containers sono capaci di essere eseguiti virtualmente ovunque (sistemi Linux, Windows e Mac; macchine virtuali e non; public cloud).
- **Easy portability**: il formato dell'immagine Docker per i containers aiuta ulteriormente la portabilità. Docker V2 image manifest è una specifica per le immagini del container che consente immagini multiarchitettura e supporta immagini indirizzabili dal contenuto.
- **Operational efficieny and reliability**: i containers sono perfetti per le architetture e le applicazioni orientate ai servizi perché ogni servizio limitato a delle risorse specifiche possono essere containerizzati. Servizi separati possono essere considerati come black boxes.
    - Questo incrementa l'efficienza perché ogni container può essere controllato in maniera sicura e avviato/fermato quando necessario in modo indipendente dagli altri.
    - L'affidabilità aumenta perché la separazione e la divisione del lavoro permette ad ogni servizio di continuare la sua esecuzione anche se un altro è in guasto.

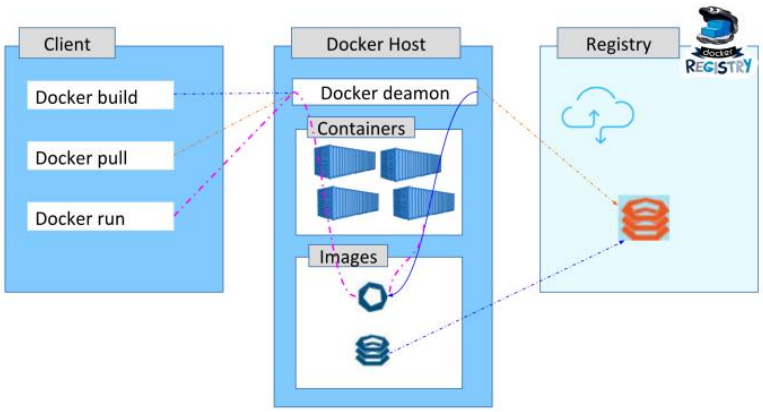
- **Easy versioning**: Un package di un nuovo container può essere creato per ogni nuova versione di una applicazione, incluse tutte le dipendenze, i moduli e le librerie necessarie per la "giusta" versione.
- **Security**: i containers aggiungono un layer addizionale di sicurezza dal momento in cui l'applicazione non viene eseguita direttamente nell'host del sistema operativo. Esistono vincoli di sicurezza se l'applicazione in esecuzione all'interno dei contenitori ha privilegi di root.

8.2.4 Problemi con più versioni

Se installi un package SomePackage, esso verrà istallato nel path python 2.7:
/usr/lib64/python2.7/site-packages/orca
Se dopo avrai bisogno di un software con pyhton3 e questo software ha bisogno della libreria orca, non sarai in grado di compilare poiché la libreria orca è istallata per python2.7. Questo problema non è facile da debuggare.

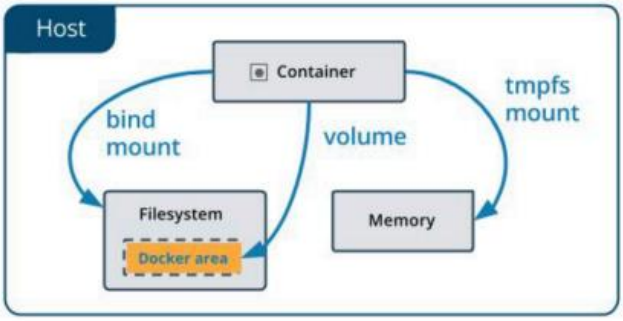
- Gli ambienti virtuali python permettono ai packages Python di essere istallati in una locazione isolata per una particolare applicazione, piuttosto che essere istallati in un ambiente globale.
- Gli ambienti virtuali hanno le loro directories di installazione e non condividono librerie con gli altri ambienti virtuali.
- Dockerfile per creare un contenitore che racchiude una versione specifica di Python e l'applicazione

8.2.4 Docker Registry e Hub



Docker Registry è il luogo in cui l'immagini Docker sono immagazzinate. Il registro può essere sia una repository locale dell'utente sia una repository pubblica come un Docker Hub che permette ad utenti multipli di collaborare nella costruzione di un'applicazione. Anche più team nella stessa organizzazione possono scambiare o condividere i containers caricandoli sul Docker Hub. **Docker Hub** è la repository cloud propria di Docker simile a GitHub.

8.2.5 Docker Volume



I volumi sono il meccanismo preferito per rendere persistenti i dati generati e usati dai containers Docker.

I volumi sono completamente gestiti da Docekr.

8.2.6 Comandi Docker

Installare da docker.com

Comandi semplici:

- docker version
- docker images

Per eseguire un semplice programma "hello world": docker run hello-world
docker verifica se è un'immagine locale con questo nome e in tal caso la esegue, altrimenti tenta di scaricarla automaticamente.

Ci sono molte immagini pubbliche disponibili che possono essere usate per lavorare con Docekr.

Il seguente esempio richiama un'immagine hello-world usando il comando docker pull:

```
[ ~ ]docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:31b9c7d48790f0d8c50ab433d9c3b7e17666d6993084c002c2ff1ca09b96391d
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest

[ ~ ]docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
hello-world   latest    bf756fb1ae65   12 months ago   13.3kB
```

Per creare un container da una immagine si può usare il comando docker create:

```
[ ~ ]docker create hello-world

2ffd5f2c5a7562fbf1d7b89a14c11a52e5843dd7938f380a8cd53f3952da99de
```

Per eseguire un container possiamo usare il comando docker container start. L'opzione -i esegue il container in modo interattivo mostrerà l'output:

```
[ ~ ] docker container start -i 2ffd5f2c5a7562fbf1d7…

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Per avere una lista di containers che abbiamo costruito, si può usare il comando docker container ls. L'opzione -a permette di vedere sia i containers in esecuzione che quelli fermati:

```
[ ~ ]docker container ls -a

CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS                      PORTS     NAMES
5017fd2b94c2   hello-world   "/hello"   7 minutes ago    Exited (0) 7 minutes ago              stoic_nobel
5f0cea57eacf   ubuntu        "bash"     10 minutes ago   Exited (127) 8 minutes ago            condescending_neumann
2ffd5f2c5a75   hello-world   "/hello"   14 minutes ago   Exited (0) 13 minutes ago             hungry_mclaren
```

Eseguire i containers in maniera interattiva permette di eseguire comandi all'interno del container se supportati:

```
[ ~ ]docker run -it openjdk
Unable to find image 'openjdk:latest' locally
latest: Pulling from library/openjdk
a73adebe9317: Pull complete
8b73bcd34cfe: Pull complete
1227243b28c4: Pull complete
Digest: sha256:7ada0d840136690ac1099ce3172fb02787bbed83462597e0e2c9472a0a63dea5
Status: Downloaded newer image for openjdk:latest
Jan 21, 2021 4:48:58 PM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
|  Welcome to JShell -- Version 15.0.2
|  For an introduction type: /help intro

jshell> System.out.println("hello world");
Hello world
```

Per vedere quali containers sono attualmente in esecuzione si può usare il comando docker ps. Questo è utile quando ci sono containers in esecuzione in backgroud.

8.2.7 Creare un container da git

Docker può buildare automaticamente immagini leggendo le istruzioni da un Dockerfile. Un dockerfile è un documento di testo che contiene tutti i comandi che un utente può chiamare da linea di comando per assemblare un'immagine.

Le istruzioni non sono case-sensitive. Però è convenzione scriverle in UPPERCASE per distinguerle più facilmente dagli altri argomenti.

La sintassi è la seguente:

- **FROM** L'immagine base da usare
- **RUN** Eseguire i comandi quando l'immagine docker è costruita
- **WORKDIR** Specifica la directory dove i comandi sono eseguiti
- **USER** Cambia utente
- **COPY** Copie di file
- **CMD** Esegue comandi quando il container è in esecuzione

Esempio Dockerfile:

```
FROM openjdk

RUN useradd -ms /bin/bash ojdk
RUN mkdir -p /home/ojdk/app/ && chown -R ojdk:ojdk /home/ojdk/app

WORKDIR /home/ojdk/app

COPY *.java ./
COPY junit-* ./

USER ojdk

RUN javac -cp "junit-4.10.jar:." *.java

COPY --chown=ojdk:ojdk . .

CMD ["java", "-cp", "junit-4.10.jar:.",
"org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
```

1) Usa l'immagine OpenJDK per avere un ambiente java preconfigurato
2) Aggiunge un nuovo utente "ojdk" che useremo per l'esecuzione degli scripts
3) Crea una directory che conterrà il nostro file e darà i permessi ai nostri utenti
4) Cambia la working directory con la directory che abbiamo creato
5) Copia i file java e junit
6) Cambia l'utente in "ojdk"
7) Compila tutto il codice
8) Copia i file dalla working directory e dà i permessi ad ojdk
9) Esegue tutti i test

Per costruire un'immagine docker usando un dockerfile possiamo usare il comando docker image build e fornirgli la directory dove esiste il dockerfile. L'opzione --tag permette di dare un nome e un tag all'immagine docker.

```
TestRepo $ docker image build . --tag "calculator:latest"
Sending build context to Docker daemon  590.3kB
Step 1/9 : FROM openjdk
 ---> e105e26a0a75
Step 9/10 : COPY --chown=ojdk:ojdk . .
 ---> 47cff2b55e3c
Step 10/10 : CMD [ "java", "-cp", "junit-4.10.jar:.", "org.junit.runner.JUnitCore", "TestAdd", "TestSub"]
 ---> Running in c8395bc770b4
Removing intermediate container c8395bc770b4
 ---> 6b345c94e511
Successfully built 6b345c94e511
Successfully tagged calculator:latest
```

Possiamo usare docker run -it per eseguire la nostra immagine in maniera interattiva e aprire una shell bash nella nostra working directory:

```
TestRepo $ docker run -it calculator bash
[ojdk@419a727a1ca1 app]$ ls
Calculator.class  README.md      TestSub.class
Calculator.java   TestAdd.class  TestSub.java
Dockerfile        TestAdd.java   junit-4.10.jar
[ojdk@419a727a1ca1 app]$
```

8.3 Docker-compose

Docker Composer, un tool di gestione piuttosto versatile che rede l'adozione di Docker ancora più agevole e facilmente gestibile l'adozione di applicazioni di questo tipo.

Docker Compose è un tool ufficiale installabile su Linux (incluso Raspberry Pi OS), Windows, macOS e che consente, in modo piuttosto pratico, di gestire i propri container salvandone la configurazione di istanziamento in un unico file di configurazione in formato YAML, per esempio:

```yaml
version: '3'
services:
    db:
        build: ./mysql
        image: poc/mysql-for-wordpress
        volumes: db_data:/var/lib/mysql
        restart: always
        environment:
            MYSQL_ROOT_PASSWORD: somewordpress
            MYSQL_DATABASE: wordpress
            MYSQL_USER: wordpress
            MYSQL_PASSWORD: wordpress
    wordpress:
        depends_on: db
        image: wordpress:latest
        ports: "8000:80"
        restart: always
        environment:
            WORDPRESS_DB_HOST: db:3306
            WORDPRESS_DB_USER: wordpress
            WORDPRESS_DB_PASSWORD: wordpress
            WORDPRESS_DB_NAME: wordpress
    volumes:
        db_data: {}
```

Questo file di configurazione, per esempio, istruisce Docker Compose sull'istanziamento di due container, db (basato sull'immagine del motore MySql per Docker) e wordpress (basato sull'immagine del CRM WordPress per Docker) e altro.

Docker Compose serve, per l'appunto, a gestire in modo immediato i propri container. I suoi comandi infatti consentono di:

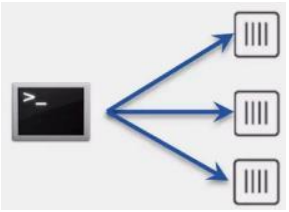
- avviare, fermare, e riavviare le applicazioni istanziate;
- vedere lo stato dei servizi in esecuzione;
- consultare lo stream dei log dei servizi in esecuzione;
- eseguire comandi all'interno dei container
- e altro.

In pratica, senza l'adozione di Docker Compose ogni qual volta è necessario mettere in esecuzione un container, è necessario conoscerne il comando ed eseguirlo.

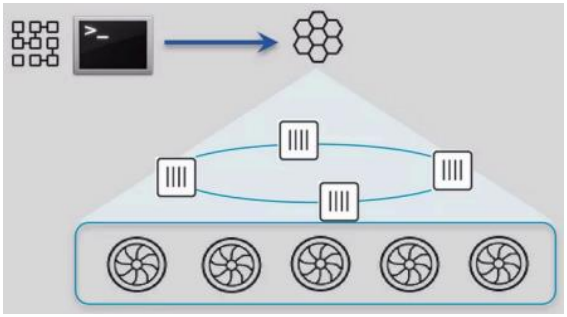
Usando Docker Compose affiancato a Docker, ogni qual volta si ha necessità di aggiungere un container al proprio "stack" è sufficiente aggiungere una porzione di codice di configurazione ad esso relativa al proprio

file docker-compose.yaml, il quale consentirà a Docker Compose di “conoscere” le caratteristiche del nuovo container (oltre a quelli già presenti) e quindi consentirne una rapida e pratica gestione.

8.3.1 Con o senza compose



- Senza compose:
  - Costruire ed eseguire un container alla volta
  - Connessione manuale dei containers tra loro
  - È necessario fare attenzione alle dipendenze e all'ordine di avvio
- Con compose:
  - Definizione dell’app multicontainer nel file compose.yml
  - Singolo comando per distribuire l’intera applicazione
  - Gestione autonoma delle dipendenze del contenitore
  - Lavora con Docker Swarm, Networking, Volumes, Universal Control Plane



8.3.2 Per praticità…

Immaginiamo banalmente di dover amministrare **un certo numero di container**: anche solo passando alla nostra amata domotica personale, volendo istanziare tutte le componenti necessarie in un sistema di media complessità, potremmo arrivare a definire anche una decina.

Va da sé che per ognuno sarebbe **necessario segnarsi da parte il corretto comando di esecuzione** (in base ai tanti parametri diversi di ognuno) per eseguirlo nuovamente ogni qual volta sia necessario cancellare e ricreare il container (per esempio, in caso di aggiornamento dell’immagine per sopravvenuto aggiornamento dell’applicazione).

Con Docker Compose, invece, una volta dichiarate nel suo file di configurazione le configurazioni dei vari container, **di esse ci si può dimenticare**: basterà infatti utilizzare i suoi comandi rapidi.

docs.docker.com/compose/install/linux

8.3.3 “docker-compose” o “docker”?

Il modo più recente per eseguire Docker Compose è quello di utilizzarlo come plug-in di Docker: in sostanza, lo si installa come plug-in e questo lo aggiunge quale possibile sotto-comando del comando principale “docker“. Precedentemente (ma la pratica è ancora valida e operativa), Docker Compose poteva venire installato come applicazione a sé, la quale poteva venire evocata col comando “docker-compose“.

In pratica, i comandi (esemplificativi): docker compose up -d e docker-compose up -d sono **negli effetti la stessa cosa**, ma possono essere utilizzati:

- Il primo, solo installando Docker Compose come plug-in di Docker;
- Il secondo, solo installando Docker Compose come applicazione a sé (ormai pratica deprecata).

Si possono istallare e quindi usare entrambi, anche se ha poco senso.

8.3.4 docker-compose

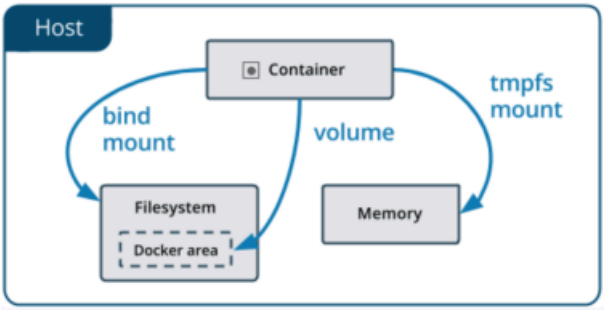
Usare Compose è essenzialmente un processo a tre passi:

1) Definisci il tuo ambiente dell’app con un dockerfile in modo che possa essere riprodotto ovunque
2) Definisci i servizi che compongono la tu app in un file compose.yaml in modo che possono essere eseguiti insieme in un ambiente isolato
3) Esegui docker compose e il comando docker compose avvia ed esegue l’applicazione

Di default tutti i file creati dentro un container sono archiviati su un layer container scrivibile.
Questo significa che:

- I dati non persistono quando il container non esiste più, e può essere difficile estrarre i dati dal container se un altro processo li necessita.
- Il layer scrivibile di un container è strettamente collegato alla macchina host su cui è in esecuzione il container. I dati non possono essere spostati facilmente
- spostare facilmente i dati da qualche altra parte.
- Un layer scrivibile di un container ha bisogno di un driver d'archiviazione per gestire il filesystem. Il driver d'archiviazione fornisce un filesystem d'union, usando il kernel Linux. Questa astrazione in più riduce le performance rispetto all'uso di *data volumes*, che scrive direttamente all'host del filesystem.

Docker ha due opzioni per i containers che archiviano file sulle macchine host, così che i file sono persistenti anche quando il container viene arrestato: volumi, e bind mounts.



- I volumi sono archiviati in una parte dell'host del filesystem, ciò significa che è gestito da docker (in linux: /var/lib/docker/volumes/). I processi non-docker non possono modificare questa parte di filesystem. I volumi sono il miglior modo per rendere persistenti i dati in Docker.
- Bind mounts può essere archiviato ovunque all'interno dell'host system. Potrebbero anche essere file di sistema o directories importanti. I processi non docker sul docker host o un docker container può modificarli in qualsiasi momento.
- tmpfs mount sono archiviati solo nella memoria dell'host system, e non sono mai scritti nel filesystem dell'host system.

File Compose. L'utilizzo di un file YAML, tipicamente denominato docker-compose.yml, Docker Compose consente di definire i servizi, le reti e i volumi dell'applicazione. Questo file funge da modello completo per l'intera applicazione, semplificando la gestione, la riproduzione e la condivisione della configurazione dell'applicazione.

Servizi. In Docker Compose i servizi indicano singole istanze di container create da specifiche immagini Docker. Questi servizi sono la base di un'applicazione multi-container e possono essere interconnessi attraverso le reti, condividendo i dati con i volumi.

Reti. Docker Compose facilita la creazione di reti personalizzate per le applicazioni containerizzate, consentendo una comunicazione fluida tra i servizi. Sebbene Compose generi automaticamente una singola rete per tutti i servizi per impostazione predefinita, è possibile definire reti aggiuntive per isolare servizi specifici o stabilire topologie complesse.

Volumi. Docker Compose offre volumi per conservare i dati generati e utilizzati dai contenitori Docker. I volumi facilitano la condivisione dei dati tra i contenitori e garantiscono la conservazione dei dati quando un contenitore viene rimosso o aggiornato.

Scalabilità. Con Docker Compose è possibile scalare facilmente i servizi per gestire carichi di lavoro maggiori, regolando il numero di istanze del contenitore per un servizio specifico. Questo può essere fatto usando l'attributo scale o l'opzione --scale della riga di comando.

Sostituzione di variabili d'ambiente. Docker Compose supporta la sostituzione delle variabili d'ambiente, consentendo di adattare il file Compose a vari ambienti o fasi della pipeline di sviluppo. Questa funzione consente una maggiore flessibilità e personalizzazione in diversi scenari di distribuzione.

Compose preserva tutti i volumi usati dai tuoi servizi. Quando docker compose viene eseguito, se trova containers da esecuzioni precedenti, copia i volumi del vecchio container al nuovo container. Questo processo assicura che tutti i dati che hai creato non vengano persi.

8.4 FTP, SSH, TELNET

Quando lavori su un sistema operativo Linux, puoi aver bisogno di comunicare con altri devices. Per questo scopo, ci sono alcuni servizi che puoi sfruttare.

I servizi che possono aiutarti nella comunicazione sono: networks, altri sistemi Linux e utenti in remoto.

8.4.1 SSH

SSH sta per Secure Shell, è usato per connettersi in modo sicuro ad un computer in remoto. Rispetto a Telnet, SSH è sicuro dove la connessione clinet/server è autenticata usando un certificato digitale e le password sono crittografate. Quindi è ampiamente utilizzato dagli amministratori di sistema per controllare server Linux da remoto.

Per abilitare SSH sul tuo sistema Ubuntu, hai bisogno di istallare il server OpenSSH. Per farlo basta eseguire il seguente comando da terminale: `sudo apt install openssh-server` (la porta di default è 22)

Il server SSH supporta vari schemi di autenticazione. I due più popolari sono:

- autenticazione basata su password
- autenticazione basata su chiave pubblica

Passi per impostare chiavi di sicurezza SSH:

1) Con il comando `ssh-keygen` creare la coppia di chiavi ssh.
2) Copiare e istallare la chiave pubblica ssh usando il comando `ssh-copy-id` su un server Linux o Unix.
3) Aggiungersi all'account sudo o all'amministratore del gruppo wheel.
4) Disabilitare la password del login per gli account root
5) Testare la password senza l'accesso con le chiavi ssh usando il comando `ssh user@server-name`

8.4.2 FTP E TELNET

TELNET (TELecommunication NETwork) e FTP (File Transfer Protocol) sono entrambi protocolli del livello applicativo. Sono protocolli orientati alla connessione; creano una connessione tra il server e l'host remoto.

TELNET. Questo software collega il personal computer a un server di rete e converte i dati in testo normale. Fondamentalmente, viene utilizzato per l'accesso remoto a un sistema. Semplifica le modifiche e il controllo del server.

FTP. FTP viene utilizzato per il trasferimento di pagine Web e per scaricare file da altri server diversi. Fondamentalmente, viene utilizzato per trasferire i file da un sistema a un altro in modo affidabile ed efficiente.

8.5 ADE

Agile Development Environment (ADE) è uno strumento modulare basato su Docker per garantire che tutti gli sviluppatori di un progetto dispongano di un ambiente di sviluppo comune e coerente.

In altre parole, ADE sfrutta i containers Docker creando un ambiente di sviluppo riproducibile, semplificando il processo per i nuovi sviluppatori.

Grazie a ADE gli sviluppatori non hanno bisogno di avere nessuna conoscenza riguardo docker per iniziare: ade start inizializza il container, ade enter apre il container, e ade stop arresta il container; tutto ciò che dovrebbe persistere tra il riavvio (es. repository clone, file di log, etc…) è mantenuto nella home directory, la quale è associata ad una directory sull'host.

Una volta all'interno del container, gli sviluppatori hanno la corretta versione degli strumenti per lo sviluppo e delle dipendenze di terze parti già installata e pronta all'uso, e possono lanciare gli editor e gli IDEs come Atom, VSCode, e CLion.

Se uno sviluppatore vuole provare un tool (o aggiornarlo), possono installarlo ed esso verrà aggiornato dopo il riavvio di ADE (ade stop seguito da ade start, oppure ade start -f). In alternativa, se il nuovo strumento si rileva utile, metterlo a disposizione per il resto del team è semplice come aggiornare l'immagine Docker, un aggiornamento che gli altri sviluppatori possono scaricare con ade start --update.

ADE si integra con la CI (Continuos Integration) pipeline.

- **ade-cli** è uno strumento di riga di comando che fornisce i comandi ade start, ade enter, e ade stop.
- **ADE image** è una immagine docker che ade-cli usa per creare un container; questa immagine ha bisogno di essere buildata in modo compatibile con ADE
- **ADE** si riferisce ad un container iniziato da un'immagine ADE con ade-cli

8.5.1 ADE, Docker e GITlab

ADE non nasconde Docker, ADE infatti consente a chiunque conosca Docker di sfruttare funzionalità che non sono fornite immediatamente da ade-cli.
Ad esempio, quando lavorano con sensori reali, gli sviluppatori devono montare dispositivi USB, aprire porte o utilizzare configurazioni di rete personalizzare per poter inviare dati alla loro applicazione.
Invece di scrivere wrapper complicati (e limitati) attorno ai comandi Docker nativi, ade-cli consente di utilizzare comandi Docker nativi per rendere questi dispositivi disponibili all'interno dell'ambiente di sviluppo.

ADE usa docker e gitlab per gestire ambienti di strumenti di sviluppo per progetto e immagini di volume opzionali. Le immagini del volume possono contenere strumenti di sviluppo aggiuntivi o versioni software rilasciate. Consente un facile passaggio da un branche all'altro per tutte le immagini.

8.5.2 Terminologia

- **Docker** permette la creazione di *container* con risorse isolate dall'host del sistema operativo
- Le **immagini** sono snapshots del filesystem per un container. Sono basate su un dockerfile, e sono create usando una docker build. Le immagini possono essere caricate su un *registro*, dove gli altri possono scaricarlo.
- Un **container** è una istanza in esecuzione di un'immagine. Quando un container è cancellato, qualsiasi cambiamento al suo filesystem è di norma scartato. Un container può pubblicare parti del suo filesystem come *volumi*, i quali possono essere montati su altri container.
- Un **registro** è un applicazione server-side che contiene immagini Docker pre-buildate
- **FNQ** è il termine per il path simile all'URL che descrive un'immagine (ad esempio http://registry.gitlab.com/apexai/ade-atom:latest)
- Un **tag** è una versione specifica di un'immagine (ad esempio *latest* del precedente FQN)

ADE crea un Docker container da una *base image* e la monta volumi di sola lettura aggiuntivi in /opt. La base image fornisce strumenti di sviluppo (come vim, udpreplay, etc…), e i volumi forniscono strumenti di sviluppo addiozionali (es.: IDEs, librerie di terze parti) o rilasci di versioni software.

```
ade             | v0.1.0   | v0-1-0   | registry.gitlab.com/autowar
autowareauto    | v0.1.0   | v0-1-0   | registry.gitlab.com/autowar
atom            | v1.35.0  | latest   | registry.gitlab.com/apexai/
cuda            | v8.0     | v8-0     | registry.private-gitlab/cud
```

- La prima riga della tabella è la *base image*, tutte le altre voci sono *volumi* e hanno una corrispettiva voce in /opt. Ogni riga mostra il nome dell'immagine, informazioni sulla versione (git, hash o tag), il docker tag, e il FQN di ogni immagine. Se non vengono fornite informazioni sulla versione durante la creazione dell'immagine docker, la seconda colonna sarà vuota.

8.5.3 Installare ADE

- Docker
- Per rendere ade disponibile globalmente, installarlo da qualche parte nel tuo PATH (i path più comuni sono ~/.local/bin e /usr/local/bin)
- Scaricare il binario collegato staticamente dalla pagina release del progetto ade-cli (es.: per x86_64)

```
cd /path/from/step/above
wget https://gitlab.com/ApexAI/ade-cli/-/jobs/1341322851/artifacts/raw/dist/ade+x86_64
mv ade+x86_64 ade
chmod +x ade
./ade --version
    4.3.0
./ade update-cli
./ade --version
    <latest-version>
```

- Per abilitare l'autocompletamento, aggiungere al tuo .zshrc o .bashrc:

```
if [ -n "$ZSH_VERSION" ]; then
    eval "$(_ADE_COMPLETE=source_zsh ade)"
else
    eval "$(_ADE_COMPLETE=source ade)"
fi
```

8.5.4 ADE Home e ADE files

1) ADE necessita di una directory sull'host che verrà montata come home directory dell'utente all'interno del container. Verrà popolato con dotfiles e dovrà essere diverso dalla directory home dell'utente sull'host. Nel caso in cui si utilizzi ADE per più progetti, si consiglia di utilizzare più directory home dedicate ad ADE, una per ciascun progetto.
2) ADE cercherà una directory contenente un file denominato **.adehome** iniziando con la working directory corrente e continuando con le directory principali per identificare la directory home ADE da montare.

Per impostare ADE per un progetto, bisogna preparare due componenti:

- Una **base-image** che è l'immagine Docker che fornisce tutti i programmi e le dipendenze per il progetto (dockerfile)
- Un file .aderc che specifica l'immagine docker da avviare e gli argomenti addizionali da eseguire.
- In aggiunta alla **base-image** e al file .aderc, si consiglia di utilizzare i volumi per fornire programmi e librerie autonomi di grandi dimensioni. I volumi consentono al progetto di aggiornare diversi componenti di ADE senza richiedere agli utenti di scaricare un'immagine di dimensione sempre maggiore.

Il file env.sh configura l'ambiente predefinito su ade enter

8.5.5 entrypoint file

Il codice seguente è stato preso da:
https://gitlab.com/ApexAI/minimal-ade/-/blob/master/entrypoint?ref_type=heads

1) Ci sono alcune linee che aiutano il debug dell'entryoint:

```
set -e

if [[ -n "$DEBUG" ]]; then
    set -x
fi
```

2) L'immagine docker eseguita in CI dovrebbe comportarsi come un'immagine docker regolare:

```
if [[ -n "$GITLAB_CI" ]]; then
    exec "$@"
fi
```

Questo assicura che ADE e CI lavorino insieme per fornire agli sviluppatori un unico ambiente.

3) L'ambiente dell'utente è configurato all'intero di ADE:
   a. Impostare il timezone

```
if [[ -n "$TIMEZONE" ]]; then
    echo "$TIMEZONE" > /etc/TIMEZONE
    ln -sf /usr/share/zoneinfo/"$TIMEZONE" /etc/localtime
    dpkg-reconfigure -f noninteractive tzdata
fi
```

   Questo assicura che il programma abbia il corretto orario

   b. Creare un utente dentro il container ed essere certi che sia nel gruppo corretto

```
groupdel "$GROUP" &>/dev/null || true
groupadd -og "$GROUP_ID" "group"

useradd -M -u "$USER_ID" -g "$GROUP_ID" -d "/home/$USER" -s /bin/bash "$USER"

groupdel video &>/dev/null || true
groupadd -og "${VIDEO_GROUP_ID}" video
gpasswd -a "${USER}" video

for x in /etc/skel.*; do
    target="/home/$USER/${basename "$x"}"
    if [[ ! -e "$target" ]]; then
        cp -a "$x" "$target"
        chown -R "$USER":"$GROUP" "$target"
    fi
done
```

   Questo assicura che l'utente si comporti nello stesso modo sia in ADE che nell'host. Questo passo include la creazione di una home directory basata su /etc/skel/, in modo diverso da come una distribuzione Linux crea una home directory.

4) Chiamare lo script .adeinit per ogni volume:

```
if [[ -z "$SKIP_ADEINIT" ]]; then
    for x in /opt/*; do
        if [[ -x "$x/.adeinit" ]]; then
            echo "Initializing $x"
            sudo -Hu "$USER" -- bash -lc "$x/.adeinit"
            echo "Initializing $x done"
        fi
    done
fi
```

Ciò consente ai volumi di fornire script che devono essere eseguiti all'avvio per configurare l'istanza ADE in modo appropriato per il volume.

5) Infine, si avvia ADE:

```
echo 'ADE startup completed.'
exec "$@"
```

Una volta che il dockerfile è completo, per eseguire l'immagine: docker build -t <image_name>:<tag>

8.5.6 .aderc

È anche possibile configurare ADE usando variabili d'ambiente. Per effettuare queste configurazioni a livello di progetto, bisogna aggiungere le variabili di ambiente al file .aderc. Come minimo, il file .aderc (preso qui) dovrebbe includere una lista di immagini:

```
export ADE_IMAGES="
    image:latest
"
```

Spesso, bisogna anche includere l'istanza e il registro Gitlab, così che ade sappia dove scaricare l'immagine:

```
export ADE_GITLAB=gitlab.com
export ADE_REGISTRY=registry.gitlab.com
export ADE_IMAGES="
    registry.gitlan.com/autowareauto/ade:v0-1-0
    registry.gitlan.com/autowareauto:v0-1-0
    registry.gitlan.com/apexai/ade-atom:latest
"
```

8.6 Vagrant

Vagrant (si può installare qui) offre una soluzione astraendo le differenze di fondo tra le piattaforme VM, consentendo ai team DevOps di effettuare il provisioning di nuove VM con un'unica CLI e una sintassi di configurazione coerente.

I team che adottano Vagrant possono creare macchine virtuali coerenti indipendentemente dal fatto che vengano eseguite in un ambiente on-premise o in un ambiente cloud e possono passare facilmente da un ambiente all'altro.

Vagrant ha a disposizione un unico strumento per imparare, indipendentemente dal fatto che stiano costruendo macchine virtuali per piattaforme on-premise, piattaforme cloud o entrambe. È possibile modificare i file id configurazione di Vagrant facilmente e rieseguirli o rivederli in un secondo momento per capire come è stata costruita una VM.

Vagrant permette inoltre ai team DevOps di migrare verso il cloud o da un provider all'altro senza dover riprogettare il processo di creazione delle macchine virtuali. Le macchine virtuali create da Vagrant offrono un elevato livello di isolamento, spesso sfruttando il supporto specializzato per la virtualizzazione integrato nelle moderne CPU. Questo rende le macchine virtuali e gli strumenti che le creano, come Vagrant, la scelta migliore quando la sicurezza e l'isolamento sono una priorità assoluta.

Molte distribuzioni Linux, come Ubuntu e Fedora, forniscono anche box Vagrant ufficiali su cui i team DevOps possono costruire. Questo riduce il tempo necessario per creare macchine virtuali personalizzate.

8.6.1 Perché Vagrant?

Le macchine virtuali non sono sufficienti poiché il provisioning non è tracciabile e quindi non reperibile. Vagrant invece consente di avere delle macchine virtuali completamente "scriptabili" sia in fase di configurazione (RAM, networking, spazio disco…) che di provisioning (installazione di MongoDB, RabbitMQ, …). Ognuno di questi script potrà poi essere aggiunto al nostro source control preferito e arrivare a versionare vari scenari realizzando quella pratica nota come **infrastructure as code**.

Un altro vantaggio di Vagrant risiede nella capacità di virtualizzare la virtualizzazione.

Inoltre, con Vagrant si può creare e gestire macchine virtuali da linea di comando in modo semplice e veloce; condividere immagini di macchine virtuali; lavorare sulla macchina virtuale via terminale per mezzo di SSH; e si ha una gestione semplificata di una cartella condivisa per editare file da host (e non da macchina virtuale).

8.6.2 Docker o Vagrant?

Vagrant permette di creare e gestire ambienti di macchine virtuali in un unico flusso di lavoro. Sia che si stiano creando delle macchine virtuali locali da eseguire con gli hypervisor o solo delle macchine virtuali per il cloud, Vagrant offre comunque un formato di configurazione coerente, un'unica CLI e dei provisioner condivisi per installare il software e modificare le configurazioni.

Docker offre la possibilità di impacchettare il software e la configurazione di supporto in immagini che vengono eseguite in modo coerente su più piattaforme. Inoltre, permette ai team DevOps di avere la certezza che il software venga eseguito su una workstation locale nello stesso modo in cui viene eseguito su una piattaforma cloud gestita.

Sebbene Vagrant e Docker condividano l'obiettivo di creare ambienti ripetibili, lo fanno in modi diversi ma complementari. Vagrant e Docker non sono tecnologie che si escludono a vicenda ed entrambi gli strumenti possono essere utilizzati fianco a fianco. Ad esempio, i team DevOps possono utilizzare Docker per sviluppare ed eseguire applicazioni e Vagrant per ricreare ambienti specializzati per riprodurre problemi.

Docker può anche essere eseguito all'interno di una macchina virtuale creata da Vagrant, magari per testare nuove versioni di Docker o per testare strumenti distribuiti come immagini Docker in un ambiente isolato.

8.7 Ansible

Ansible è una piattaforma software open source per l'automazione e la gestione dell'infrastruttura IT, creata da Red Hat (link per l'installazione qui). È progettato per automatizzare le attività ripetitive e distribuire rapidamente le applicazioni e può essere utilizzato per gestire un'ampia gamma di sistemi, inclusi server, dispositivi di rete e risorse cloud. Ansible si basa su una semplice sintassi YAML e utilizza un'architettura basata su push tramite SSH, il che significa che invia comandi ai nodi che gestisce anziché richiedere l'installazione di agenti su tali nodi. L'operatività viene gestita attraverso playbook Ansible, ovvero modelli di attività di automazione eseguite senza necessità di coinvolgimento umano.
Il nome "ansible" viene dalla fantascienza: fa riferimento ad una macchina capace di comunicazioni istantanee o superluminali.

Ansible è costituito da diversi componenti. I sistemi gestiti possono includere server, storage, networking e software. Questi sono i target del sistema di gestione della configurazione. L'obiettivo è mantenere questi sistemi in stati noti e determinati. Un altro aspetto di un sistema di gestione della configurazione è la descrizione dello stato desiderato per il sistema. Il terzo elemento principale è il software di automazione, che ha il compito di garantire che i sistemi e il software di destinazione siano mantenuti nello stato desiderato.

9. IO

9.1 File in UNIX

Il kernel di UNIX vede tutti i file come flussi non formattati di byte; il compito di interpretare ogni struttura logica interna ad un file lasciato alle applicazioni.
I file UNIX sono strutture composte dai tre seguenti elementi:

- **Nome**: stringa di caratteri alfanumerici utilizzata da utenti e programmi per fare riferimento al file.
- **Inodo**: struttura dati che contiene le informazioni sul file necessarie al SO per la sua gestione (attributi), oltre agli indirizzi per accedere ai dati contenuti nel file.
- **Dati**: i dati effettivamente contenuti nel file.

Il sistema assegna a ciascun file un identificatore numerico, detto ***i-node***.

9.1.1 Inodo

Le informazioni contenute all'interno di ogni i-nodo residente in un file system sono le seguenti:

- *Modo del file*. Flag di 16 bit dove sono memorizzati il tipo dle file ed i suoi permessi di accesso (read, write, execute)
- *Contatore link*. Numero di riferimenti all'inodo nelle directory
- *ID proprietario*. Identificatore del proprietario del file
- *ID gruppo*. Identificatore del gruppo a cui appartiene il proprietario
- *Dimensione*. Dimensione in byte del file
- *Indirizzi*. Informazioni di indirizzamento ai dati del file
- *Ultimo accesso*. Tempo dell'ultimo accesso ai dati del file
- *Ultima modifica*. Tempo dell'ultima modifica ai dati del file
- *Ultima modifica stato*. Tempo dell'ultima modifica all'inodo

9.2 Accesso/creazione di File

Quando un processo si riferisce ad un file per nome (path assoluto o relativo), il kernel suddivide il path componente per componente, controllando i permessi di accesso alle directory relative.

In caso di esito positivo, il kernel:

- **Ritorna l'inodo** del file, se il file esiste e il processo chiede di accedervi;
- **Assegna un inodo** non usato, se il processo chiede di creare un nuovo file.

In entrambi i casi, il kernel restituisce al processo un **intero non negativo**, detto descrittore del file, che resta associato al file (attraverso il relativo inodo) fino a quando il file non viene rilasciato.

È attraverso i descrittori che il kernel accede ai file e ne permette le elaborazioni da parte dei processi.

9.2.1 I/O di basso livello

- La maggior parte delle operazioni sui file oridnari in mabiente UNIX si possono eseguire utilizzando solo le cinque chiamate di sistema open, read, write, lseek, close.
- Il kernel associa un *file descriptor* ad ogni file aperto
    - Il file descriptor è un intero
    - Quando un file viene aperto con open, la funzione open restituisce il file descriptor associato al file
- Le costanti simboliche $STDIN\_FILENO(0), STDOUT\_FILENO(1)$ e $STDERR\_FILENO(2)$ sono definite in $unistd.h$

9.2.2 Descrittori di file

Alla richiesta di aprire un file esistente o di creare un nuovo file, il kernel ritorna un descrittore di file al processo chiamante. Quando si vuole leggere o scrivere su un file si passa come argomento a read e write il descrittore ritornato da open.

Per convenzione il descrittore 0 viene associato allo standard input, 1 allo standard output e 2 allo standard error. I numeri 0, 1 e 2 possono essere sostituiti dalle costanti STDIN_FILENO, STDOUT_FILENO e STDERR_FILENO definite nell'header <*unistd.h*>

9.2.3 La funzione open

```
#include <sys/types.h>  // data types
#include <sys/stat.h>   // data returned by stat()
#include <fcntl.h>      // file control options

int open(const chat *pathname, int oflag, ... /* mode_t mod */);
```

- Restituisce il descrittore del file, -1 in caso di errore;
- Permette sia di aprire un file già esistente che di creare il file nel caso in cui questo non esista;
- pathname è il pathname (assoluto o relativo) del file.
- oflag permette di specificare le opzioni, mediante costanti definite in <fcntl.h>, combinate con "|" (or bit-a-bit)
- mode è il modo del file ed è un parametro opzionale, utilizzato solo nel caso di creazione del file ("…" modo ISO C per dire variabile).

oflag può assumere diversi valori (definiti nell'header <fcntl.h>):

- O_RDONLY apri solo in lettura
- O_WONLY apri solo in scrittura
- O_RDWR apri in lettura e scrittura

Solo una delle precedenti costanti può essere specificata, con una combinazione OR di:

- O_APPEND esegue un append dalla fine del file per ciascuna write
- O_CREAT crea il file se non esiste
- O_EXCL se utilizzato insieme a O_CREAT, ritorna un errore se il file esiste
- O_TRUNC se file esiste e aperto con successo write-only/read-write

S_IRWXU	Permesso di lettura, scrittura ed esecuzione per il proprietario
S_IRURS	Permesso di letture per il proprietario
S_IWUSR	Permesso di scrittura per il proprietario
S_IXUSR	Permesso di esecuzione per il proprietario
S_IRWXG	Permesso di lettura, scrittura ed esecuzione per il gruppo
S_IRGRP	Permesso di lettura per il gruppo
S_IWGRP	Permesso di scrittura per il gruppo
S_IXGRP	Permesso di esecuzione per il gruppo
S_IRWXO	Permesso di lettura, scrittura ed esecuzione per gli altri
S_IROTH	Permesso di lettura per gli altri
S_IWOTH	Permesso di scrittura per gli altri
S_IXOTH	Permesso di esecuzione per gli altri

I permessi per open

Esempi di open:

```
open("prova.txt", O_RDONLY)
open("prova.txt", O_RDONLY | O_CREAT, S_IRWXU)
open("prova.txt", O_RDWR | O_CREAT | O_EXCL, S_IRWXU)
```

9.2.4 La funzione create

Un nuovo file può essere creato anche con:

```
#include <fcntl.h>

int create(const chat *pathname, mode_t mod);

//equivalente a:
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

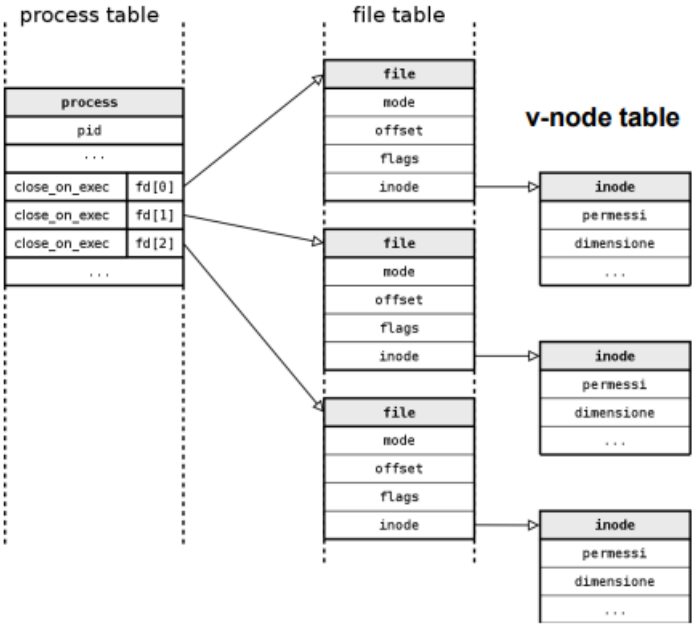
Il nuovo file è aperto solo per la scrittura.

9.3 Implementazione nel kernel

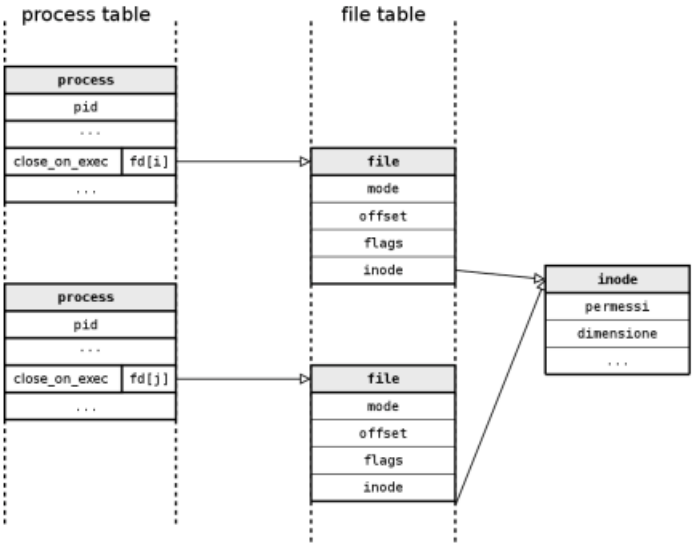
Il kernel usa due strutture dati indipendenti per gestire i file aperti

- Ogni processo mantiene la lista dei propri file descriptor (chiave astratta per accedere ai file, in POSIX, inter)
- Ogni file descriptor punta ad un elemento della file table
- La file table specifica per ogni file aperto:
    - La modalità di apertura del file (lettura, scrittura o entrambe)
    - Le opzioni come O_APPEND, etc…
    - L'offset corrente
    - L'inode corrispondente

Un processo con tre descrittori aperti:



Due processi che accedono allo stesso file:



9.3.1 Trattare gli errori

- Molte system call restituiscono -1 in caso di errore
- Per avere più informazioni, si usa la variabile globale **errno** (error number)
- La funzione $perror(\text{const char *})$ stampa la stringa passata come parametro, e poi un messaggio in base al valore corrente di errno.

9.3.2 La funzione close

```
#include <unistd.h>

int close(int filedes)
```

- Chiude il file identificato da filedes e precedentemente aperto con open
- Restituisce 0 in caso di successo o -1 in caso di errore

9.4 L'offset

Ad ogni file aperto è associato un intero, detto offset, che rappresenta la posizione (espressa in numero di byte dall'inizio del file) in cui verrà effettuata la prossima operazione di I/O.

L'offset è inizializzato a zero da open (a meno che non sia specificato O_APPEND).

Le operazioni di read e write incrementano il valore dell'offset di un numero di byte pari al numero di byte letti/scritti.

9.4.1 lseek

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

off_t lseek (int filedes, off_t offset, int whence);
```

- Modifica l'offset corrente del file
- Restituisce il nuovo valore dell'offset, o -1 in caso di errore

Il valore del parametro offset è interpretato in base al parametro whence:

- SEEK_SET: L'offset corrente è posto a offset byte dall'inizio del file.
- SEEK_CUR: L'offset corrente è incrementato di offset byte
  - Il valore del parametro offset può essere sia positivo che negativo
- SEK_END: L'offset è posto a offset byte dalla fine del file.

- Il valore del parametro offset può essere sia positivo che negativo.

Per conoscere l'offset corrente, è sufficiente eseguire:

```
off_t currpos = lseek(filedes, 0, SEEK_CUR);
```

Esempio: Testa lo stdio per vedere se può fare seeking

```
int main(void) {
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

```
$./a.out
cannot seek
$./a.out < /etc/passwd
seek OK
```

9.4.2 Funzione read

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);
```

- Restituisce il numero di byte effettivamente letti
  - 0 se ci troviamo alla fine del file
  - -1 in caso di errore
- L'operazione di lettura avviene partendo dall'offset corrente del file.
  - L'offset viene incrementato opportunamente

Il numero di byte letti può essere diverso dal parametro nbytes quando:

- Il numero di byte ancora presenti nel file è inferiore ad nbytes.
- La lettura avviene da un terminale (si legge una riga alla volta)
- La lettura avviene da un buffer di rete (nbyte superiore)
- La lettura avviene da una pipe o una FIFO
- L'operazione è interrotta da un segnale

9.4.3 Funzione write

```
#include <unistd.h>

ssize_t write(int filedes, void *buf, size_t nbytes);
```

- Restituisce il numero di byte effettivamente scritti
  - -1 in caso di errore
- L'operazione di scrittura avviene partendo dall'offset corrente del file
  - L'offset viene incrementato opportnamente

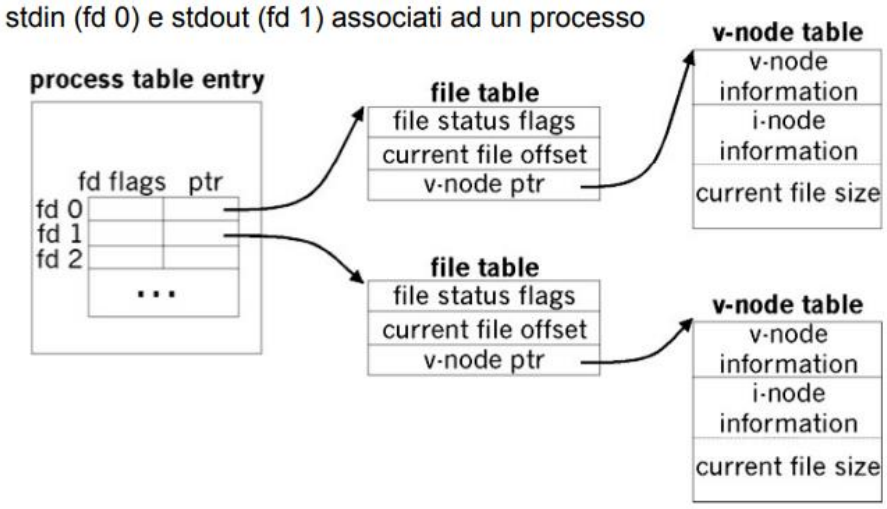
9.4.4 Esercizi

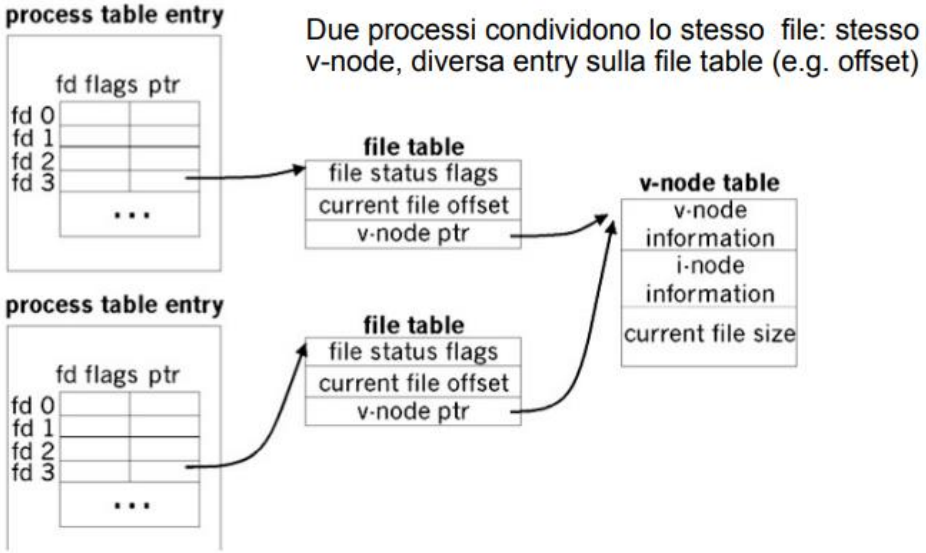
- Scrivere un programma che mostra il contenuto di un file a byte alterni (un carattere sì e uno no)
- Scrivere un programma che mostra il contenuto di un file alla rovescia, cioè a partire dall'ultimo carattere fino ad arrivare al primo

9.5 Condivisione di file

Il kernel utilizza tre strutture dati per la gestione dell'I/O:

1) Ciascun processo ha un elemento nella **tabella dei processi**. Tale elemento è un "vettore" di descrittori di file aperti, ciascuno con un puntatore ad un elemento della **tabella dei file**.
2) Il kernel possiede una tabella per ciascun file aperto con i flag di stato del file (lettura, scrittura, append, …), l'offset corrente ed un puntatore ad un elemento della **tabella dei v-node**
3) Ciascun file aperto (o device) ha una **struttura v-node**. Il v-node contiene informazioni sul tipo di file e sulle funzioni che operano su di esso (informazione in i-node).





9.6 Accesso ad un file

Cosa accade quando un processo cerca di accedere ad un file?

- Quando un processo accede ad un file mediante una write, l'elemento della tabella dei file relativo all'offset viene aggiornato e, se necessario (modificato size), viene aggiornato l'i-node.
- Se il file è aperto con O_APPEND, un flag corrispondente è messo nella tabella dei file (ogni write alla fine del file)
- Una chiamata ad lseek modifica solo l'offset corrente del file e non viene eseguita nessuna operazione di I/O.
- Se si chiede di posizionarsi alla fine del file, il valore corrente dell'offset nella tabella dei file viene preso dal campo della tavola di i-node che descrive la dimensione.

9.6.1 Esempio

Prendiamo due programmi:

```
1   // PROGRAMMA A
2   strcpy(string, "aaaaaaaaa\n");
3   fd = open("testfile", O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
4
5   if (fd < 0) {
6       perror("Errore in apertura");
7       exit(1);
8   }
```

```c

do {
    if (write(STDOUT_FILENO, "Comando:", 8) < 8)
        perror("write error");

    input = getchar();
    __fpurge(stdin);
    string[0] = input;
    write(fd, string, 10);

    lseek(fd, (off_t) 3, SEEK_SET); // sposta l'offset ad "INIZIO" file

    if (write(STDOUT_FILENO, "Eseguito\n", 9) < 9)
        perror("write error");

} while (input != 'f');

close(fd);
```

```c
// PROGRAMMA B
strcpy(string,"bbbbbbbbb\n");
fd = open("testfile", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);

lseek(fd, 0, SEEK_END); // sposta l'offset a FINE file

do {
    if (write(STDOUT_FILENO, "Comando:", 8) < 8)
        perror("write error su stdout");

    input = getchar();
    __fpurge(stdin);
    string[0] = input;

    if (write(fd, string, 10) < 10)
        perror("write error");

    if (write(STDOUT_FILENO, "Eseguito\n", 9) < 9)
        perror("write error su stdout");
} while (input != 'f');

close(fd);
```

Eseguendo i due programmi nel seguente ordine:

1) Esegui A
2) Esegui B
3) A scrive 5 stringhe
4) B scrive 7 Stringhe
5) A scrive 5 stringhe
6) B scrive 5 stringhe
7) Termina B
8) Termina A

3)
qaaaaaaaa
waaaaaaaa
eaaaaaaaa
raaaaaaaa
taaaaaaaa

4)
1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb

5)
1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
qaaaaaaaa
waaaaaaaa
eaaaaaaaa
raaaaaaaa
taaaaaaaa

6)
1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
8bbbbbbbb
9bbbbbbbb
0bbbbbbbb
1bbbbbbbb
2bbbbbbbb

8)
1bbbbbbbb
2bbbbbbbb
3bbbbbbbb
4bbbbbbbb
5bbbbbbbb
6bbbbbbbb
7bbbbbbbb
8bbbbbbbb
9bbbbbbbb
0bbbbbbbb
1bbbbbbbb
2bbbbbbbb
faaaaaaaaa

9.7 Duplicazione di File descriptor

Un file descriptor può essere duplicato utilizzando:

```
#include <unistd.h>

int dup(int filedes);
int dup2(int filedes, int filedes2);
```

- dup ritorna un file descriptor che punta allo stesso file indirizzato da *filedes*.
  - Il valore ritornato da dup è il minimo file descriptor non utilizzato
- dup2 prende in input *filedes2*, il file descriptor da usare nella duplicazione.
  - Se *filedes2* è aperto, dup2 chiude il file prima di duplicare il descrittore *filedes*, se *filedes* è uguale a *filedes2* ritorna *filedes* e non chiude
  - dup2 è una operazione atomica

9.8 Ottenere info su file

```
int stat(const char *file_name, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

- Queste system call prendono in input un puntatore ad una struttura stat che conterrà le informazioni sul file
- stat e lstat prendono in input il nome del file
  - lstat dà informazioni sui link simbolici (info su link simbolico, non su file linkato)
- fstat prende in input il file descriptor del file (il file deve essere aperto)

9.8.1 La struttura stat

```
struct stat {
    mode_t st_mode;         // file type e mode (permissions)
    uid_t st_uid;           // user ID of owner
    gid_t st_gid;           // group ID of owner
    ino_t st_ino;           // inode number
    dev_t st_dev;           // device number (file system)
    dev_t st_rdev;          // device type (if inode device)
    nlink_t st_nlink;       // number of link
    off_t st_size;          // total size, in bytes
    time_t st_atime;        // time of last access
    time_t st_mtime;        // time of last modification
    time_t st_ctime;        // time of last change
    blksize_t st_blksize;   // blocksize for filesystem I/O
    blkcnt_t st_blocks;     // number of blocks allocated
}
```

- **Tipo di file**
  - Regular file: contiene "dati" di qualche tipo (anche **gli eseguibili sono regular file**)
  - Directory file: contiene nomi e puntatori a inode (è necessario utilizzare system call specifiche per manipolarlo)
  - Block Special file: rappresentano particolari device (per esempio i dischi)
  - Character Special file: rappresentano particolari device (per esempio la scheda audio)
  - FIFO (o pipe): utilizzati per comunicazione tra processi
  - Socket: utilizzati per comunicazione tra processi
  - Symbolic Link: link simbolico (o soft)

  Il tipo di file associato ad un pathname od un file descriptor è codificato nel campo st_mode della struttura stat.

  Per interpretare st_mode, si usano le seguenti macro:
  - S_ISREG(m): is regular file?
  - S_ISDIR(m): is directory?
  - S_ISCHR(m): is character device?
  - S_ISBLK(m): is block device?
  - S_ISFIFO(m): is fifo?
  - S_ISLNK(m): is symbolic link?
  - S_ISSOCK(m): is socket?

  Le macro prendono come argomento il campo st_mode
- **User e Goup ID**

  Ad ogni file sono associati uno User ID (uid) ed un Group ID (gid)
  - Memorizzati in st_uid e st_gid della struttura stat

  Si ricorda che ogni processo possiede i seguenti ID:
  - Real user e Real group: utente (e gruppo) che ha lanciato il processo
  - Effective user ed Effective group: utente (e gruppo) che determina i diritti di accesso al processo

9.8.2 Accesso ai File

Il campo st_mode codifica i permessi di accesso ai file.

Per accedere ad un file è necessario:

- Avere diritto di esecuzione su TUTTE le directory nel path
  - Esempio: /home/utente/LSO/esempio.txt
  - Il permesso di lettura sulla directory consente di leggere i nomi dei file ma non di aprirli
- Avere i permessi di accesso specifici per il file

Per creare un file in una directory

- Permessi di scrittura sulla directory

Per cancellare un file in una directory

- Permessi di scrittura sulla directory (non sul file!)

L'accesso ai file è regolato dalla seguente sequenza:

- Se l'effective user del processo è 0 (superuser): OK
- Se l'effective user del processo è uguale all'owner del file
  - Controlla i permessi del gruppo ed, in caso, nega l'accesso
- Se l'effective group del processo è uguale al group del file
  - Controlla i permessi del gruppo ed, in caso, nega l'accesso
- Altrimenti controlla i permessi per gli "altri".

9.8.3 Funzione access

```c
#include <unistd.h>

int access(const char *pathname, int mode);
```

- Controlla i permessi di accesso ad un file in base ad UID e GID "reali"
  - Mentre normalmente valgono UID e GID effettivi
- Il parametro mode può assumere i valori
  - R_OK, W_OK, X_OK: Lettura, scrittura o esecuzione
  - F_OK: Esistenza

9.8.4 Funzione chmod e fchmod

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

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

- Consentono di modificare i permessi di accesso ai file
  - chmod prende in input un pathname
  - fchmod prende in input un file descriptor (file deve essere aperto)
- Il parametro "mode" può essere una combinazione delle seguenti costanti:
  - S_ISUID, S_ISGID, S_ISVTX Set used id exe, group id exe, saved text
  - S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR owner
  - S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP group
  - S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH others

9.8.5 Funzione chown

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

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

- Modificano il campo st_uid ed st_gid del file indicato dal pathname (chown e lchown) o dal file descriptor (fchown)
- Se il parametro "owner" o "group" è uguale a -1, il campo corrispondente non viene modificato
- In molti sistemi, solo un processo del superuser può modificare il campo st_uid
- Un processo può modificare il gruppo se è owner del file e il parametro group è uguale all'effective GID del processo o ad uno dei gruppi "alternativi"

10. Processi

10.1 Programma

Un programma C inizia la sua esecuzione obbligatoriamente con la chiamata alla funzione **main**. Lo standard C stabilisce che la funziona main può non avere argomenti o prendere i due argomenti **argc** e **argv[]**:

- $\text{int}\ main(\text{void});$
- $\text{int}\ main(\text{int}\ argc, \text{char}^*\ argv[\,]);$
    - $argc$ rappresenta il numero di argomenti passati in input (incluso il nome del programma, definito dal nome del file contenente la chiamata a main);
    - $argv$ è un vettore di puntatori a carattere e rappresenta la **lista di argomenti** (opzioni e parametri) del programma, con $argv[0]$ che rappresenta il nome del *programma*

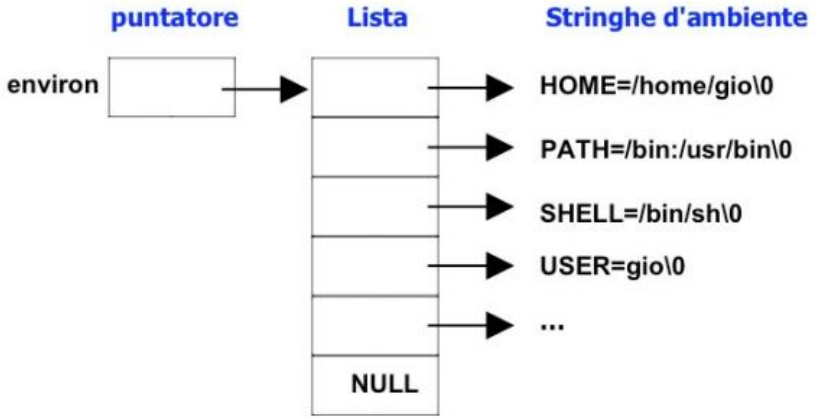
10.1.1 Lista di Ambiente

Ad ogni **programma** viene passata, oltre alla lista degli argomenti di input, una **lista d'ambiente**. Essa serve alle applicazioni per definire il contesto in cui operano (directory home, tipo di terminale, nome utente, etc…).

Analogamente alla lista degli argomenti, la **lista d'ambiente** è un array di puntatori a carattere, in cui l'ultimo puntatore punta a NULL.

Ciascun puntatore della lista contiene l'indirizzo di una stringa del tipo $nome = valore$, detta **stringa d'ambiente**.

L'indirizzo dell'array di puntatori è contenuto nella variabile globale **environ**: $\text{extern char}^{**}\ environ;$



10.1.2 Variabili di Ambiente

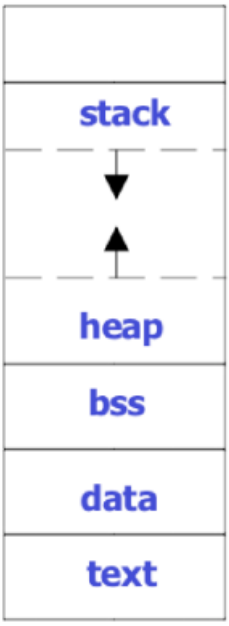
Le **variabili d'ambiente** sono definite e modificate operando sulle **stringhe d'ambiente**, grazie ad opportune funzioni. Le due più importanti sono:

```
#include <stdlib.h>

char* getenv(const char* name);
int putenv(char* string);
```

- **getenv** restituisce il puntatore alla stringa $valore$ associata al nome della variabile d'ambiente $nome$ nella stringa d'ambiente $nome = valore$. Se $nome$ non esiste restituisce il puntatore nullo.
- **putenv** aggiunge alla lista d'ambiente la stringa $string$ della forma $nome = valore$. Se $nome$ esiste ne aggiorna il valore. Restituisce zero se *OK*, non zero se *ENOMEM*.

10.1.3 Layout in memoria di un Programma

Questo segmento dell'area di memoria riservata ad un programma serve per la memorizzazione degli argomenti della linea di comando e delle variabili d'ambiente.



Lo **stack** serve a memorizzare le variabili locali e le informazioni relative alle chiamate di funzioni. Lo stack è implementato come una coda LIFO (*Last In, First Out*), per permettere l'uso annidato e ricorsivo delle funzioni.

Lo **heap** è dove viene effettuata l'allocazione dinamica di memoria (funzioni *malloc, calloc, realloc, free, etc.*.). Anche quest'area è implementata come una coda LIFO, ed è tipicamente gestita a basso livello dalla syscall **sbrk**.

Le variabili globali non inizializzate, analogamente ai puntatori a variabili globali, vengono memorizzati nel segmento **bss**, detto anche *uninitialized data segment*. I dati in tale segmento sono inizializzati dal kernel come valori numerici pari a zero o puntatori nulli prima che il programma vada in esecuzione.

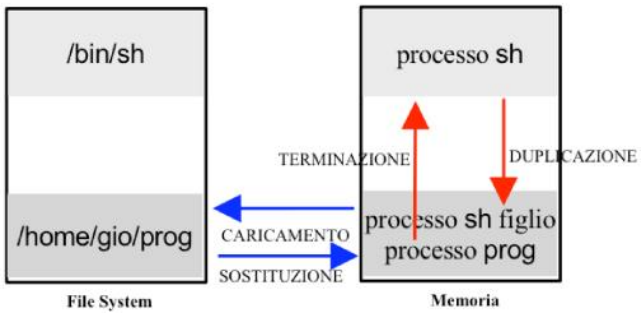
L'area **data** contiene le variabili globali ed inizializzate del programma.

L'area **text** contiene le istruzioni macchina relative al programma. Quest'area è in sola lettura, perché condivisa tra tutti i processi che eseguono uno stesso codice binario.

10.1.4 Esecuzione di un Programma in UNIX

Qual è il meccanismo per l'esexuzione di un nuovo programma in UNIX?

```
$ prog  <invio>
<eventuale output di prog>
$
```



- Quando il programma viene eseguito (una funzione exec), una routine start-up prende i command-line $args$ e l'env prima del lancio della funzione main.
- La fine dell'esecuzione avviene in diversi modi: return, exit, _exit, pthread_exit, abort, segnale, etc…

10.2 Controllo dei processi

Per il controllo dei processi occorrono primitive per gestire:

- Creazione di processi
- Esecuzione di programmi
- Terminazione di Processi
- Identificazione e proprietà dei processi

Il controllo dei processi in UNIX si esplicita mediante:

- L'**identificazione** dei processi:
  - Identificatore di processo (pid);
  - Identificatore di gruppo-processi (pgid);
  - Identificatore di utente (famiglia uid);
  - Identificatore di gruppo-utenti (famiglia gid).

- L'impiego di funzioni (primitive di controllo di processo) per:
    - Duplicare un processo esistente (fork, vfork);
    - Caricare un nuovo programma (famiglia exec);
    - Attendere la terminazione di un processo (wait, waitpid);
    - Terminare un processo (exit, _exit).

10.2.1 Identificazione di Processi

I processi in UNIX si dividono in processi di **sistema** e di **utente**; ogni processo è identificato da un numero non negativo (PID); unico PID che però viene riciclato quando termina (PID=0 scheduler, PID=1 init, PID=2 pagedaemon)

Ogni **processo utente** eredita una serie di altri identificativi:

- Il gruppo di processi cui esso appartiene (PGID);
- Lo UID e il GID dell'utente che lo ha mandato in esecuzione (RUID e RGID);
- Lo UID e i GID (gruppi primario e supplementari) di utente con i quali esso ah accesso ai file (EUID, EGID, EGID supplementari)

Ogni processo ha un PID ed un Parent's PID (ppid).
I processi formano un albero.

- "init" è la radice dell'albero
    - Viene lanciato direttamente dal kernel
    - Non ha padre
    - Ha pid = 1
- Quando un procesos termina, i suoi figli diventano figli di "init"

Le seguenti funzioni restituiscono gli identificativi e le credenziali di processo:

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

pid_t getpid(void); // identificativo del processo PID
pid_t getppid(void); // identificativo del genitore PPID

uid_t getuid(void); // credenziale utente reale RUID
uid_t geteuid(void); // credenziale utente effettivo EUID

gid_t getgid(void); // credenziale gruppo reale RGID
gid_t getegid(void); // credenziale gruppo effettivo EGID
```

10.3 Creazione di processi

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

// restituiscono il PID se OK, -1 se EAGAIN, ENOMEM o EPERM
pid_t fork(void);
pid_t vfork(void); // vfork da usare con exec, lavora nello
                   // spazio del genitore e aspetta il figlio
```

L'unico modo per istruire il kernel a creare un nuovo processo è di chiamare la funzione fork (vfork) da un processo esistente. Il processo creato viene detto figlio. Il processo che chiama fork (vfork) viene detto genitore.

Ogni chiamata a fork (vfork) ha due pid_t di ritorno:

- Al genitore viene restituito l'identificativo del figlio;
- Al figlio viene restituito l'identificativo 0;

Il figlio procede indipendentemente dal padre.

- **Memoria**: il figlio ottiene una copia nuova della memoria del padre (variabili globali e locali)
- **File aperti**: i descrittori vengono copiati come con dup; i processi condividono l'offset!
- **Segnali**: per ogni segnale, il figlio continua ad avere la stessa reazione del padre.

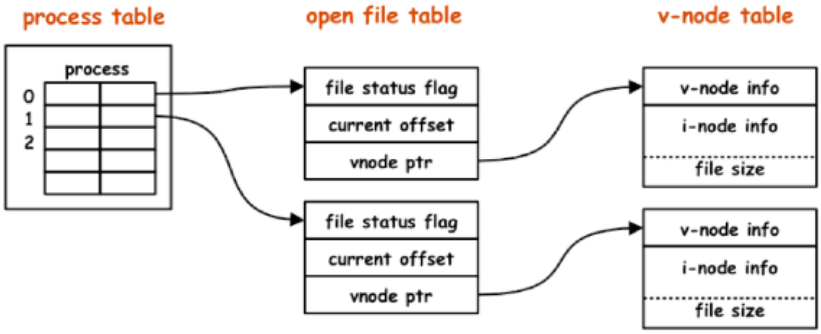
10.3.1 Creazione di Processi-fork

Ad una chiamata **fork** il kernel esegue le operazioni seguenti:

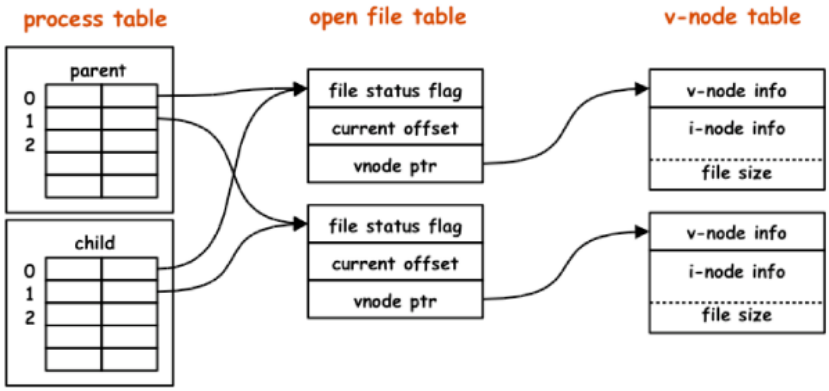
- Alloca uno spazio nella tabella dei processi per il figlio;
- Assegna un PID al figlio, unico nel sistema;
- Fa una copia dell'immagine del genitore, ad eccezione dei segmenti di memoria condivisi;
- Incrementa i contatori del file del genitore, per registrare che anche il figlio possiede tali file;
- Assegna al processo figlio lo stato READY;
- Restituisce il PID del figlio al genitore e il PID 0 al figlio;
- A seconda della routine di allocazione, può:
    - rimanere nel genitore;
    - trasferire il controllo al figlio;
    - trasferire il controllo ad un altro processo.

Una caratteristica della chiamata fork è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child.

Prima della fork:



Dopo la fork:



È importante notare che padre e figlio condividono lo stesso file offset. Se così non fosse, avremmo un problema:

- Un processo esegue fork e poi attende che il processo figlio termini (system call wait)
- Supponiamo che lo stdout sia rediretto ad un file, e che entrambi i processi scrivano su stdout
- Se padre e figlio non condividessero lo stesso offset:
    - Il figlio scrive su stdout e aggiorna il proprio current offset
    - Il padre sovrascrive stdout e aggiorna il proprio current offset

Esercizi con fork:

- Scrivere un programma C che apre un file, effettua una fork e scrive messaggi diversi sul file a seconda che sia padre o figlio. Come si alternano i messaggi nel file?
- Nel programma prima descritto spostare la open dopo la fork e verificare se il contenuto del file è cambiato rispetto al programma precedente e spiegarne il perché.

10.3.2 Funzione vfork()

```c
#include <unistd.h>

pid_t vfork(void);
```

Crea un nuovo processo come fork, ma non copia lo spazio di indirizzamento, il processo creato viene eseguito nello spazio del genitore finché il figlio non esegue exec o exit.

Prendiamo il seguente programma:

```c
int glob = 6;
int main (void) {
    int var;
    pid_t pid;
    var = 88;

    printf("before vfork\n");
    if ( pid = vfork()) {
        perror("vfork"),
        exit(0);
    }
    else if (pid == 0) {
        glob++;
        var++; // cambia le variabili del padre
        _exit(0);
    }
    printf("pid = %d, glob = %d, var = %d", getpid(), glob, var);

    exit(0);
}
```

```
$./a.out
before vfork
pid = 2624, glob = 7, var = 89
```

10.3.3 Race Condition

Una race condition si verifica quando più processi elaborano dati condivisi e l'effetto dell'insieme di siffatte elaborazioni dipende dall'ordine in cui i processi sono eseguiti.

L'ordine in cui vengono elaborate le istruzioni per un genitore de un figlio dopo un fork (vfork) in generale dipende dallo scheduler e dal carico del sistema.

Una chiamata fork (vfork) può causare una race condition, se le istruzioni relative al genitore ed al figlio operano su dati condivisi; il rilevamento di una race condition di questo tipo può essere molto difficile da rilevare in esecuzione.

10.4 Esecuzione di programmi

L'esecuzione di un nuovo programma in UNIX si ottiene con al chiamata ad una delle funzioni exec; le funzioni exec non generano un nuovo processo ma modificano il layout in memoria del processo chiamante per l'esecuzione del nuovo programma.

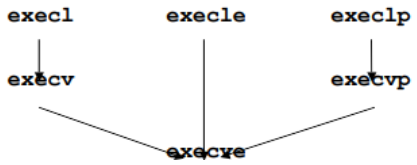
Quando un processo chiama una delle funzioni exec, le aree di memoria text, data, heap e stack relative al processo corrente vengono sostituite in base al nuovo programma. Il nuovo programma incomincia la propria esecuzione a partire dalla funzione main.

10.4.1 La famiglia di system call exec

Se fork fosse l'unica primitiva per creare nuovi processi, la programmazione in ambiente Unix sarebbe ostico, dato che si potrebbero creare soltanto copie dello stesso processo.

La famiglia di primitive exec può essere utilizzata per superare tale limite in quanto le varie system call exec permettono di iniziare l'esecuzione di un altro programma sovrascrivendo la memoria del processo chiamante.

In realtà tutte le funzioni chiamano in ultima analisi execve che è l'unica vera system call della famiglia. Le differenze tra le varianti stanno nel modo in cui vengono passati i parametri. Le altre cinque sono semplicemente funzioni di libreria che invocano execve.



Esempio di utilizzo di execl:

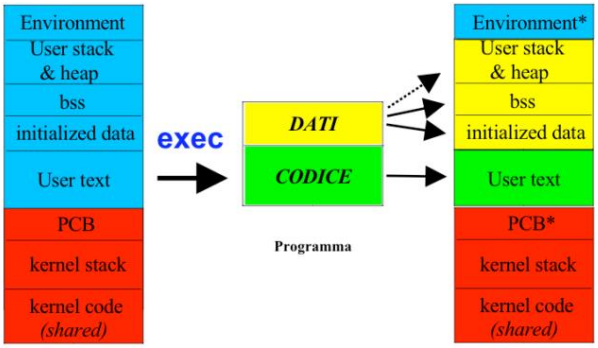
```c
#include <stdio.h>
#include <unistd.h>

main() {
    printf("Esecuzione di ls\n");
    execl("/bin/ls", "ls", "-l", (char *)0);
    perror("La chiamata di execl ha generato un errore\n");
    exit(1);
}
```

- Si noti che execl elimina il programma originale sovrascrivendolo con quello passato come parametro.
- Quindi le istruzioni che seguono una chiamata a execl verranno eseguire soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.

10.4.2 Chiamata alla system call exec

Quando un processo chiama una delle system call exec:

- Il processo viene rimpiazzato completamente da un nuovo programma (text, data, heap, stack vengono sostituiti)
- Il nuovo programma inizia a partire dalla sua funzione main
- Il process ID non cambia



Esistono sei versioni di exec:

- Con/senza gestione della variabile di ambiente PATH; se viene gestita la variabile d'ambiente, un comando corrispondente ad un singolo filename verrà cercato nel PATH
- Con variabili di ambiente ereditate / con variabili di ambiente specificate
- Con array di argomenti / con argomenti nella chiamata (null terminated)

Per default, i file aperti dal processo corrente restano aperti dopo una exec; questo comportamento si può cambiare usando la system call fcntl. Questo comportamento è utile per redirigere STDIN e STDOUT.

Cosa viene ereditato da exec? PID e PPID; real uid e real gid; supplementary gid; process group ID; session ID; terminale di controllo; current working directory; root directory; maschera creazione file (umask); file locks; maschera dei segnali; segnali in attesa.

10.4.3 execl

```c
int execl(const char *pathname, const char *arg0, ...);
```

- execl accetta il nome di un programma da eseguire ed un numero variabile di argomenti per il programma
- l'ultimo argomento deve essere un puntatore nullo di tipo char*
- execl("$a.out$","$a.out$","xxx",(char*)$NULL$) esegue il programma a.out, con argomenti "a.out" e "xxx"
- se execl ha successo, il controllo non viene mai restituito al chiamante (il processo chiamante diventa il nuovo programma); altrimenti, restituisce -1

10.4.4 Utilizzo combinato di fork e exec

L'utilizzo combinato di fork per creare un nuovo processo e di exec per eseguire nel processo figlio un nuovo programma costituisce un potente strumento di programmazione in ambiente Linux.

Esempio:

```
main() {
    pid_t pid;
    switch (pid = fork()) {
        case -1:
            fatal("fork failed");
            break;
        case 0:
            execl("/bin/ls", "ls", "-l", (char *)0);
            fatal("exec failed");
            break;
        default:
            wait((int *)0);
            printf("ls completed\n");
            exit(0);
    }
}
```

Esercizi:

- Scrivere un programma che manda in esecuzione un eseguibile il cui filename è inserito come argomento sulla linea comando e ne aspetta la terminazione.
- Aggiungere al programma precedente la capacità di lanciare eseguibili residenti in qualsiasi directory memorizzata nella variabile di ambiente $PATH

10.4.5 Ambiente di un processo

L'ambiente di un processo è un insieme di stringhe (terminate da \0). Un ambiente è rappresentato da un vettore di puntatori a caratteri terminato da un puntatore nullo.
Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma: identificatore = valore.

Per accedere all'ambiente da un programma C, è sufficiente aggiungere il parametro char $**envp$ a quelli del main oppure usare la variabile globale extern char $**environ$.

L'ambiente di default di un processo coincide con quello del processo padre. Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia exec, memorizzando in envp l'ambiente desiderato:

- execle($path, arg0, arg1, \dots, argn, (char\ *)0, envp$);
- execve($path, argv, envp$);

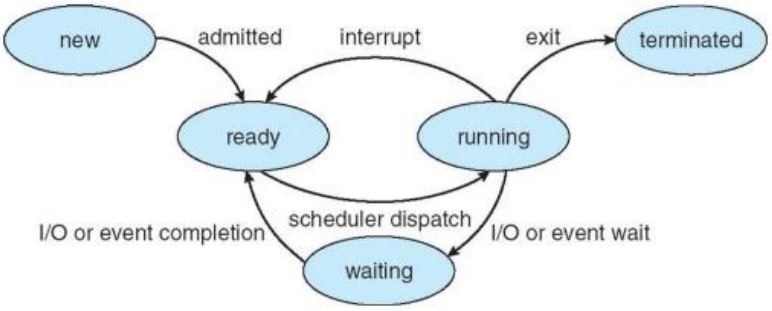
10.4.6 current working directory e root directory

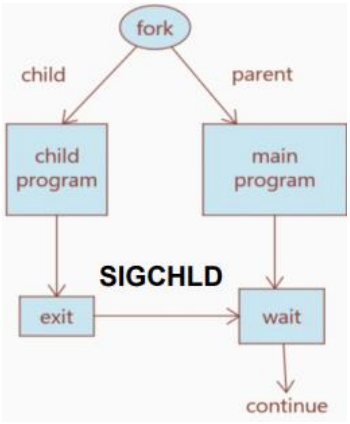
Ad ogni procesos è associata una current working directory che viene ereditata dal processo padre.
La chiamata di sistema seguente consente di cambiarla: int chdir(const char $*path$);

Inoltre, ad ogni processo è associata una root directory che specifica il punto di inizio del file system visibile dal processo stesso. Per cambiare la root directory: $\text{int chroot}(\text{const char } *path)$;

10.5 Terminazione di processi

Quando un processo termina viene inviato il segnale SIGCHLD al padre; il procesos diventa uno "zombie" finché il padre non chiama una wait/waitpid.





Esistono tre modi per terminare in modo **normale**:

- eseguire un return da main (è equivalente a chiamare exit)
- chiamare la funzione exit: $\text{void } exit(\text{int } status)$;
  - invoca di tutti gli exit handlers che sono stati registrati
  - chiude tutti gli I/O stream standard
  - è specificata in ANSI C
- chiamare la system call _exit: $\text{void } \_exit(\text{int } status)$;
  - ritorna al kernel immediatamente
  - è chiamata come ultima operazione da $exit$
  - è specificata nello standard POSIX.1

Esistono due modi per terminare in modo **anormale**:

- Quando un processo riceve certi segnali (per informazioni usa $man\ 7\ signal$)
  - Generati dal processo stesso
  - Generati da altri processi
  - Generati dal kernel
- Chiamando abort: $\text{void } abort(\ )$;
  - La chiamata ad $abort$ costituisce un caso speciale del primo caso dei tre sopra elencati, in quanto genera il segnale SIGABRT

10.5.1 Processi zombie

Cosa succede se il padre termina prima del figlio?

- Il processo figlio viene "adottato" dal processo init (PID=1), in quanto il kernel vuole evitare che un processo divenga "organo" (cioè senza un PPID)
- Quando un processo termina, il kernel esamina la tabella dei processi per vedere se aveva figli; in tal caso, il PPID di ogni figlio viene posto uguale a 1

Cosa succede se il figlio termina prima del padre?

- Generalmente il padre aspetta mediante la funzione wait che il figlio finisca ed ottiene le varie informazioni sull'exit status
- Se il figlio termina senza che il padre lo "aspetti", il padre non avrebbe più modo di ottenere informazioni sull'exit status del figlio
- Per questo motivo, alcune informazioni sul figlio vengono mantenute in memoria e il processo diventa uno zombie

10.5.2 System call wait e waitpid

```
pid_t wait(int *status);
```

- Blocca il processo finché un figlio termina
  - non blocca se c'è un figlio zombie
- Restituisce il pid del processo terminato
  - -1 in caso di errore
  - Ad esempio, se un processo non ha figli
- "status" contiene il valore di uscita del figlio
  - Se non ci interessa, passiamo NULL
  - Se diverso da NULL, il termination status viene messo in questa locazione

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

- Come wait, ma aspetta un figlio specifico (options può essere lasciato a zero)
- Argomento pid:
  - $pid == -1$ si comporta come wait
  - $pid > 0$ attende la terminazione del figlio con process id uguale a pid
  - $pid == 0$ attende la terminazione di qualsiasi figlio con process group ID uguale a quello del chiamante
  - $pid < -1$ attende la terminazione di qualsiasi figlio con process group ID uguale a $-pid$
- Parametro $options$:
  - 0, waitpid attente per la terminazione di (almeno) un figlio specificato da pid
  - WCONTINUED restituisce lo stato di ogni figlio specificato da pid che ha ripreso l'elaborazione dopo uno stop
  - WNOHANG non si blocca se il child non ha terminato, ritorna immediatamente (con valore 0) se non individua alcun figlio che è terminato
  - WUNTRACED restituisce lo stato di ogni figlio specificato da pid che è in stop ed il cui attuale stato non è stato riportato

La wait e waitpid sono utilizzate per ottenere informazioni sulla terminazione dei processi figli.
Quando un processo chiama wait o waitpid:

- può bloccarsi, se tutti i suoi figli sono ancora in esecuzione
- può ritornare immediatamente con il termination status di un figlio, se un figlio ha terminato ed il suo termination status è in attesa di essere raccolto
- può ritornare immediatamente con un errore, se il processo non ha alcun figlio

Nota: se eseguiamo una system call wait quando abbiamo già ricevuto SIGCHLD, essa termina immediatamente, altrimenti si blocca.

10.5.3 La funzione system

```
int system(char *command);
```

- Esegue un comando, aspettando la sua terminazione
- È una funzione della libreria standard definita in ANSI C (quindi non è una system call, anche se svolge una funzione analoga alla fork + exec)
- Il suo modo di operare è fortemente dipendente dal sistema; in genere chiama `/bin/sh -c command`
- Non è definita in POSIX.1, perché non è un'interfaccia al sistema operativo, ma è definita in POSIX.2

È comodo poter eseguire un comando UNIX da un programma C. Per tale ragione l'ANSI C definisce la funzione system, che rappresenta un interfaccia alla *Bourne shell* implementata mediante le primitive fork, exec e waitpid.

11. Interprocess Communication (IPC)

11.1 Segnali

Un “segnale” è un interrupt “software”, anche se la terminologia più corretta è “exception” visto che “interrupt” è usata solo per interrupt “hardware”.

Consente la comunicazione asincrona tra processi e/o tra device e processo. Ogni segnale ha un proprio nome: tutti i nomi cominciano per “SIG” e sono associati ad interi positivi; definiti in <signal.h> .

Caratteristiche dei segnali:

- Ogni segnale ha un identificatore che inizia con i tre caratteri SIG (ad esempio SIGABRT è il segnale di abort)
- Numero senali: 15-40, a seconda della versione di UNIX (POSIX:18, Linux:38)
- I nomi simbolici corrispondono ad un intero positivo
    - Definizioni di costanti in bits/signum.h

Generazione di segnali:

- Pressione di tasti speciali sul terminale
    - Es.: premere il tasto ctrl-c genera il sengale SIGINT
- Eccezioni hardware
    - Divisione per 0 (SIGFPE)
    - Riferimento non valido a memoria (SIGSEGV)
    - L’interrupt viene generato dall’hardware, e catturato dal kernel; questi invia il segnale al processo in esecuzione
- System call kill
    - Permette di spedire un segnale ad un altro processo
    - Limitazione: uid del processo che esegue kill deve essere lo stesso del processo a cui si spedisce il segnale, oppure 0 (root)
- Comando kill
    - Interfaccia shell alla system call kill
- Condizioni software
    - Eventi asincroni generati dal software del sistem operativo, non dall’hardware della macchina.
    - Esempi: terminazione di un chil (SIGCHLD), generazione di un alarm (SIGALRM)

I segnali vengono inviati in modo asincrono, quindi, non è possibile sapere quando il processo riceverà un segnale.

11.1.1 Azioni associate a segnali

È possibile indicare al kernel l’azione da intraprendere quando un segnale è generato per un processo:

- **Ignora**: Valida per quasi tutti i segnali tranne SIGKILL e SIGSTOP
- **Catch** del segnale: indicare una procedura da eseguire (signal handler).
    - Ad esempio: SIGCHLD: esegui le operazioni associate alla terminazione di un figlio; SIGINT: (CTRL-C) “cancella file temporanei” …
    - Il kernel informa il processo chiamando una funzione specificata dal processo stesso (signal handler)
    - Il signal handler gestisce il problema nel modo più opportuno
- **Default**: Eseguire l’azione di default.
    - Per molti segnali “critici”, l’azione di default consiste nel terminare il processo

- Può essere generato un file di core (eccetto quando bit set-user-id e set-group-id settati e uid/gid sono diversi da owner/group o mancano di permessi in scrittura per la directory il core file è troppo grande)

11.1.2 Catturare i segnali

Un handler (gestore) è una funzione del tipo:

```
void funzione(int num segnale) { printf("%d", num segnale); }
```

Una volta che l'handler termina, si torna al punto in cui il programma era stato interrotto.

```
typedef void *sighandler_t(int);

sighandler_t signal(int signum, sighandler_t handler);
```

- $signal(\text{SIGINT}, foo)$ imposta la funzione foo come handler del segnale SIGINT
  - Si può anche richiedere di ignorare il segnale: $signal(\text{SIGINT}, \text{SIG\_IGN})$
  - Oppure ritornare alla reazione di default: $signal(\text{SIGINT}, \text{SIG\_DFL})$
- Restituisce l'impostazione precedente, cioè uno dei seguenti valori:
  - Indirizzo dell'handler precedente
  - SIG_DFL: reazione di default
  - SIG_IGN: ignorare il segnale
  - SIG_ERR: errore
- Lo stesso signal handler può gestire più segnali.
  - Questo è il motivo per cui prende in input un intero, la codifica del segnale
- È sufficiente utilizzare ogni volta la signal indicando uno per volta tutti i segnali da gestire.

11.1.3 Inviare i segnali

I segnali si inviano ai processi oppure a gruppi di processi identificati da un process group.
Per inviare un segnale, bisogna averne il permesso; in pratica, si possono inviare segnali solo ai propri processi.

Inviare segnali da shell: $kill$ [-<segnale>] <pid> oppure $kill$ -l

- $kill$ -INT 127 invia il segnale SIGINT al processo il cui pid è 127
- $kill$ -l elenca tutti i segnali ed i loro valori numerici
- $kill$ 127 equivale a $kill$ -TERM 127

Inviare segnali in C: `int kill(pid_t pid, int sig);`

- $\text{kill}(127, SIGINT)$ invia il segnale SIGINT al processo il cui pid è 127
- restituisce 0 in caso di successo e -1 in caso di errore
- il parametro pid può assumere i seguenti valori:
  - $pid > 0$: identifica il processo con $\text{PID} = pid$
  - $pid = 0$: Tutti i processi con group ID pari al group ID del processo che esegue la kill
  - $pid = -1$ Inviato a tutti i processi del sistema per cui il valore che esegue la kill ha il permesso di inviare un segnale.
- Il parametro sig può assuemere i seguenti valori:
  - $sig > 0$: è un intero specificato in signal.h
  - $sig = 0$: è utilizzato per verificare se il processo ha i permessi per inviare un segnale al/i processo/i specificati da pid (non viene inviato alcun segnale ma è utile per verificare l'esistenza id un processo)

11.1.4 alcuni segnali in C

```
unsigned int alarm(unsigned int seconds);
```

- $alarm(30)$ prenota un segnale SIGALRM, che sarà inviato tra 30 secondi
- Restituisce 0 se non c'era nessuna sveglia già prenotata; altrimenti restituisce il tempo rimanente perché la vecchia sveglia suonasse
- $alarm(0)$ cancella la prenotazione precedente
- Esiste un'unica "sveglia" per processo.
- Può trascorrere un tempo "indefinito" da quando il kernel genera il segnale fino all'esecuzione del handler
- Attenzione: l'azione di default di SIGALRM è la terminazione quindi prima di eseguire l'alarm bisogna definire l'handler (ammenoché non si voglia terminare)

```
// Restituiscono 0 se OK, -1 su errore
int sigemptyset(sigset_t *set); // tutti i sengali sono esclusi da *set
int sigfillset(sigset_t *set); // tutti i segnali sono inclusi in *set
int sigaddset(sigset_t *set, int signum); // aggiunge signum a *set
int sigdelset(sigset_t *set, int signum); // rimuove signum da *set

// Restituisce 1 se vero, 0 se falso, -1 su errore
int sigismember(const sigset_t *set, int signum); // verifica se signum
                                                  // appartiene a *set
```

- In alcuni casi è necessario definire un insieme di segnali per indicare al kernel quali segnali "bloccare"
- Non è possibile rappresentare tutti i segnali in un unico intero (troppi segnali disponibli)
- Per questo motivo POSIX definisce il tipo sigset_t

```
// Restituisce 0 se 0k, -1 su errore
int sigprocmask(int how, const sigste_t *set, sigset_t *oldset);
```

- La "signal mask" identifica un insieme di segnali bloccati dal processo.
- sigprocmask consente di leggere, modificare od eseguire entrambe le operazioni sulla maschera dei segnali.
- Se $oldset$ è non nullo, conterrà la "vecchia" maschera
- Se set è non nullo, la nuova maschera viene calcolata in base i parametri set e how

11.1.5 Esercizi

1) Usando il comando kill della shell, inviare il segnale SIGINT ai processi con pid 1 e 2
2) Lanciare un editor di testi, metterlo in background (ctrl-Z) e poi terminarlo inviandogli il segnale SIGTERM.
3) Scrivere un programma C "aspetta.c", che scrive un messaggio su standard output ogni volta che riceve i segnali SIGINT o SIGUSR1; il programma non deve mai terminare spontaneamente.

11.2 PIPE

Per cooperare, i processi hanno bisogno di comunicare; i segnali sono un primo modo di farlo (il messaggio è il numero del segnale e, eventualmente, le informazioni in siginfo_t). Vogliamo però trasmettere informazioni arbitrarie.

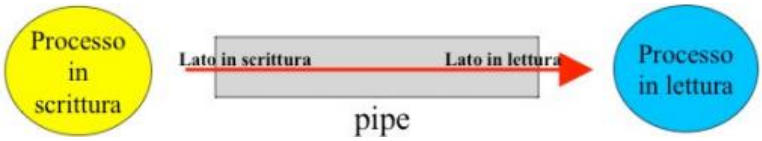
Le pipe forniscono un meccanismo attraverso il quale l'output di un processo diviene l'input per un altro processo.

Il più semplice esempio di tale meccanismo è fornito dall'operatore di pipe | di una shell: $ls \mid grep\ old$. La comunicazione tra i due comandi è unidirezionale (da ls a $grep$, il viceversa non è possibile), e la sincronizzazione è ottenuta arrestando $grep$ quando non c'è nulla da leggere e arrestando ls quando la pipe è piena.

Le pipe sono canali di comunicazione a senso unito tra due processi imparentati (tipicamente, un padre e un figlio). Un processo scrive sulla pipe (usando write) e un altro processo legge dalla stessa pipe (usando read).

Le pipe tra processi, realizzate attraverso la chiamata di sistema pipe o la funzione popen della libreria STDIO, sono del tutto analoghe: i dati immessi da un processo nella pipe con successive scritture sono accodati nell'attesa che un altro processo le legga; la coda è gestita con criterio FIFO.

Una pipe ha una dimensione finita, il cui valore è definito dalla costante PIPE_BUF: soltanto un numero massimo di byte può essere scritto o letto ogni volta.

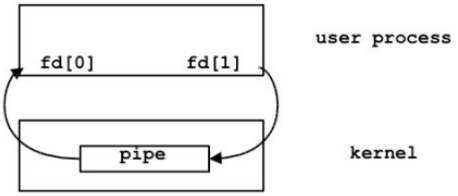


Una pipe è subordinata all'organizzazione gerarchica dei processi: affinché due processi possano comunicare in pipeline è necessario un antenato comunce che abbia predisposto una pipe a questo fine.

11.2.1 La funzione pipe

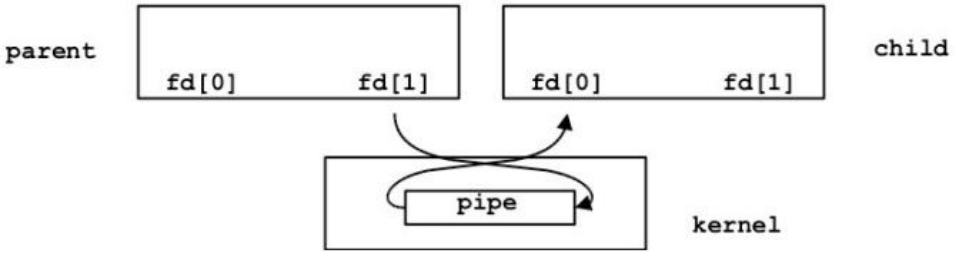
```c
#include <unistd.h>

int pipe(int filedes[2]);
```

- L'argomento $filedes$ è costituito da due descrittori di file:
  - $filedes[0]$ è aperto in lettura e rappresenta il lato in lettura della pipe;
  - $filedes[1]$ è aperto in scrittura e rappresenta il lato in scrittura della pipe;
- Restituisce 0 in caso di successo, -1 altrimenti
- L'output di $filedes[1]$ (estremo di write della pipe) è l'input per $filedes[0]$ (estremo di read della pipe)



Tipicamente, un processo crea una pipe e poi chiama fork:

- Cosa succede dopo la fork dipende dalla direzione dei dati
- I canali non utilizzati vanno chiusi. Esempio:
  - Il parent chiude l'estremo di read ($close(fd[0])$;)
  - Il child chiude l'estremo di write ($close(fd[1])$;)



11.2.2 Leggere e scrivere sulle pipe

Una volta creati, è possibile utilizzare le normali chiamate read/write sugli estremi.

Chiamata read:

- Se l'estremo di write è aperto restituisce i dati disponibili, restituendo il numero di byte; successive chiamate si bloccano fino a quando nuovi dati non saranno disponibili.
- Se l'estremo di write è stato chiuso restituisce i dati disponibili, restituendo il numero di byte; successive chiamate restituiscono 0, per indicare la fine del file.

Chiamata write:

- Se l'estremo di read è aperto i dati in scrittura vengono bufferizzati fino quando non saranno letti dall'altro processo
- Se l'estremo di read è stato chiuso viene generato un segnale SIGPIPE
    - Ignorato/catturato: write restituisce -1 e errno=EPIPE
    - Azione di default: terminazione

All'inizio una pipe è vuota (read si blocca in attesa di dati): write aggiunge dati alla pipe e read legge e rimuove dati dalla pipe. Nota che non si possono leggere più volte gli stessi dati da un pipe e non si può chiamare lseek su una pipe (i dati si ottengono in ordine FIFO).

Una pipe con una estremità chiusa si dice rotta (broken), viene generato il segnale SIGPIPE e write restiutisce un errore mentre read (se la pipe è anche vuota) restituisce 0.

Nel seguente programma il processo padre crea una pipe e in seguito esegue una chiamata fork( ), generando un processo figlio. Ciò che succede dopo la chiamata fork( ) dipende da come i dati fluiscono nel canale. In questo caso, il padre scrive sulla pipe e il figlio legge da essa. È importante sottolineare come sia il processo padre sia il processo figlio chiudano inizialmente le estremità inutilizzate del canale.

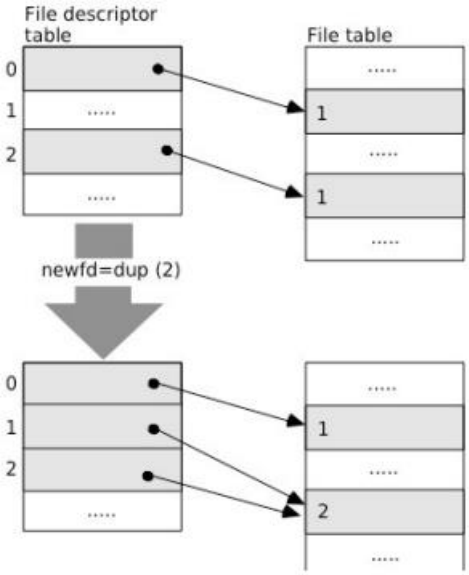
```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25

int main(void) {
    char w_msg[BUFFER_SIZE] = "Greetings";
    char r_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    // crea la pipe
    if (pipe(fd) == -1) {
        fprintf(stderr, "Pipe filed");
        return 1;
    }

    // crea tramite fork un processo figlio
    pid = fork();
    if (pid < 0) { // errore
        fprintf(stderr, "Fork filed");
        return 1;
    }
    if (pid > 0) { // processo padre
        // chiude l'estremità inutilizzata della pipe
        close(fd[0]);

        // scrive sulla pipe
        write(fd[1], w_msg, strlen(w_msg) + 1);
        close(fd[1]);
    }
```

```
34      else { // processo figlio
35          // chiude l'estremità inutilizzata della pipe
36          close(fd[1]);
37
38          // legge dalla pipe
39          read(fd[0], r_msg, BUFFER_SIZE);
40          printf("read %s", r_msg);
41          close(fd[0]);
42      }
43
44      return 0;
45  }
```

11.2.3 Pipe tra due programmi: duplicazione

Per associare lo stdout o lo stdin di un programma, rispettivamente, al lato in scrittura o a quello in lettura di una pipe, si può utilizzare una delle funzioni seguenti. In particolare, un file descriptor esistente viene duplicato da una delle seguenti funzioni:

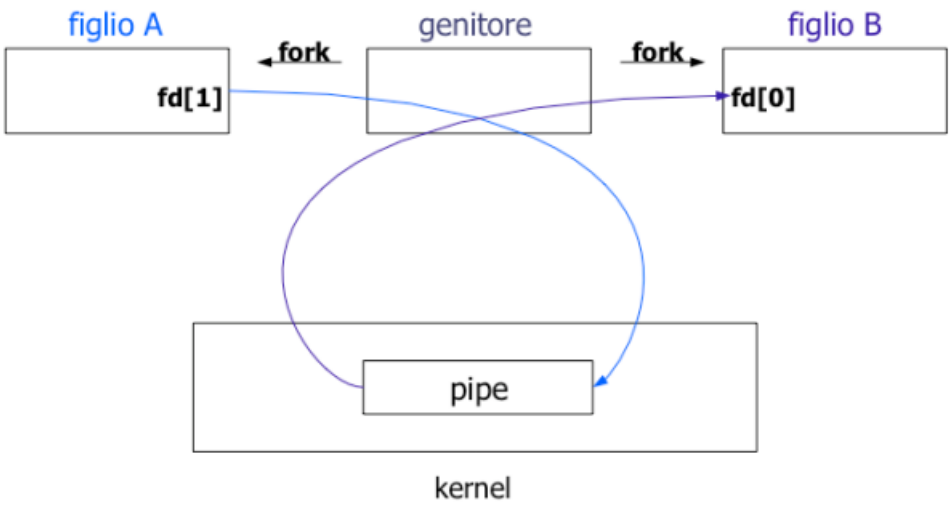
```
#include <unistd.h>

int dup(int filedes);
int dup2(int filedes, int filedes2);
```



- Entrambe le funzioni “duplicano” un file descriptor, ovvero creano un nuovo file descriptor che punta alla stessa file table entry del file descriptor originario
- Nella file table entry c’è un campo che registra il numero di file descriptor che la “puntano”
- Restituiscono il file descriptor selezionato (filedes2 nel caso di dup2) in caso di successo, -1 altrimenti
- Funzione dup
    - Selezione il più basso file descriptor libero della tabella dei file descriptor
    - Assegna la nuova file descriptor entry al file descriptor selezionato
- Funzione dup2
    - Duplica il descrittore di file $filedes$, nel nuovo descrittore $filedes2$
    - Se $filedes2$ è già aperto, esso viene chiuso prima della duplicazione.

Per realizzare una pipe tra due programmi, come `ls | grep old` la sequenza di eventi è precisamente la seguente:

1. Il genitore crea una pipe usando la funzione pipe;
2. Il genitore crea due figli con la funzione fork, dopodiché chiude entrambi i lati della pipe;
3. Il figlio scrittore chiude il lato in lettura della pipe ed associa il proprio stdout al altor in scrittura della pipe;
4. Il figlio lettore chiude il lato in scrittura della pipe ed associa il proprio stdin al lato in lettura della pipe;
5. Ciascuno dei figli carica con una exec il proprio programma;
6. Al termine della comunicazione, lo scrittore e il lettore chiudono il lato della pipe di loro competenza.



Figlio A scrittore e figlio B lettore

11.2.4 popen e pclose

L'esecuzione di un comando da parte di un processo in modo che quest'ultimo possa riceverne l'output o inviargli l'input è una operazione molto comune, per la quale è dunque preferibile avere delle funzioni di più alto livello.

La libreria standard di IO fornisce invero le funzioni ***popen*** e ***pclose***, che consentono al programmatore di evitare le azioni esplicite di creazione di una pipe, generazione di un figlio, chiusura del lato della pipe non utilizzato da parte die processi scrittore e lettore, exec di una shell per l'esecuzione del comento e, infine, attesa della terminazione di quest'ultimo.

Più precisamente, $popen$ crea una pipe, crea un figlio, chiude i lati non utilizzati della pipe ed esegue il comando; mentre $pclose$ attende che l'esecuzione del comando sia terminata e chiude la pipe.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
int pclose(FILE *filepointer);
```

La funzione $popen$ esegue il comando $cmdstring$ e restituisce un puntatore a file per il processo chiamante. Se $type$ è uguale a $'w'$ il puntaotre a file è connesso allo stdin del comando, se invece $type$ è uguale a $'r'$ tale puntatore è connesso allo stdout.

Il comando $cmdstring$ è eseguito come sh -c $cmdstring$, ossia la shell espande i caratteri speciali.

11.2.5 Named pipe (o FIFO)

Le pipe possono essere utilizzate solo se due processi hanno un antenato comunte. Le FIFO (o pipe con nome) possono essere utilizzate per consentire la comunicazione tra due processi **arbitrari**. Devono condividere solo il **nome** della FIFO.

I file speciali FIFO consentono di superare alcune delle limitazioni delle pipe. Essi difatti, rispetto a queste ultime, offrono i seguenti vantaggi:

- Una volta creati, esistono nel file system fintanto che non vengono esplicitamente cancellati;
- Possono essere usati da processi che non hanno un comune antenato.

I file FIFO possono essere creati in due modi: o attraverso la shell, con il comando mkfifo; oppure, all'interno di un programma, con la chiamata alla funzione mkfifo.
Una volta creato un file FIFO, su di esso si possono effettuare le operazioni usuali di IO su file (opne, read, write, close. **Non** la lseek).

È possibile che più processi scrivano sulla stessa FIFO. Se il numero di byte scritti sulla FIFO è inferiore a PIPE_BUF, le scritture sono "atomiche".

L'utilizzo di O_NONBLOCK consente di non bloccare le operazioni di open/read/write (attenzione ad errori e SIGPIPE). Come per le pipe, se si esegue una write su di un file FIFO che nessun processo ha aperto in lettura, è generato il segnale SIGPIPE.

È comune la situazione in cui più processi scrivono su di uno stesso file FIFO: affinché i dati non si mischino è necessario utilizzare operazioni atomiche di scrittura.
Come per le pipe, la costante PIPE_BUF stabilisce il massimo numero di byte che possono essere scritti in maniera atomica in un file FIFO.

```
int mkfifo(char *pathname, mode_t mode);
```

- Crea un FIFO dal $pathname$ specificato
- La specifica dell'argomento $mode$ è identica a quella di open, creat ($mode$ codifica i permessi di accesso al file mediante un numero ottale, ad esempio $0644 = \text{rw-r--r--}$)

Come funziona un FIFO?

- Una volta creato un FIFO, le normali chiamate open, read, write e close possono essere utilizzate per leggere il FIFO
- Il FIFO può essere rimosso utilizzando $unlink$
- Le regole per i diritti di accesso si applicano come se fosse un file normale

Chiamata $\boldsymbol{open}$

- File aperto senza flag O_NONBLOCK
  - Se il FIFO è aperto in sola lettura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in scrittura
  - Se il FIFO è aperto in sola scrittura, la chiamata si blocca fino a quando un altro processo non apre il FIFO in lettura
- File aperto con flag O_NONBLOCK
  - Se il FIFO è aperto in sola lettura, la chiamate ritorna immediatamente
  - Se il FIFO è aperto in sola scrittura, e nessun altro processo lo ha aperto in lettura, la chiamata restituisce un messaggio di errore

Chiamata $\boldsymbol{write}$

- Se nessun processo ha aperto il file in lettura viene generato un segnale SIGPIPE:
  - Ignorato/catturato: write restituisce -1 e errno=EPIPE
  - Azione di default: terminazione

Atomicità

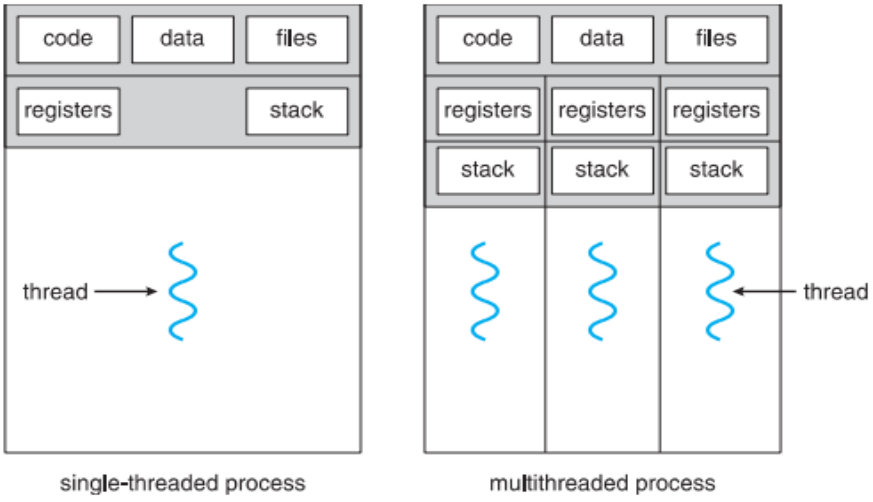
- Quando si scrive su un pipe, la costante PIPE_BUF specifica la dimensione del buffer del pipe (in genere pari a 4086, vedi $/\text{usr/include/linux/}limits.h$)
- Chiamate $write$ di dimensione inferiore a PIPE_BUF vengono eseguite in modo atomico
- Chiamate $write$ di dimensione superiore a PIPE_BUF possono essere eseguite in modo non atomico.

12. Thread

12.1 Introduzione

Un thread è l'unità di base d'uso della CPU e comprende un identificatore di thread (ID), un program counter, un insieme di registri, ed uno stack. Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali.

Un processo tradizionale, chiamato anche ***heavyweight process*** (processo pesante), è composto da un solo thread. Un processo multithread, poiché è fatto di più thread (detti peer thread) è in grado di lavorare a più compiti in modo concorrente. La figura successiva mostra la differenza tra un processo tradizionale, a singolo thread, e uno multithread (si noti che ogni thread nell'ambiente multithread ha un proprio insieme di registri e di stack che non condivide).



12.1.1 Motivazione

Il context switch tra processi richiede molto lavoro al SO: oltre a cambiare il valore dei vari registri, deve spostarsi dalle aree dati e di codice del processo uscente, a quelle del processo entrante.

Se dati e codice di quest'ultimo erano stati swappati in memoria secondaria, occorre prima riportarli in memoria primaria. Se due (o più) processi potessero condividere dati e codice, il context switch fra di loro sarebbe molto più veloce.

Per soddisfare questo tipo di esigenza è nato il concetto di thread.

12.2 POSIX thread

Ad ogni thread è associato in modo esclusivo il suo stato della computazione, fatto da: valore del program counter e degli altri registri della CPU; uno stack.

Ma un thread condivide con i suoi peer thread il codice in esecuzione, i dati ed i file aperti.

Come un processo ha un pid di tipo pid_t, un thread ha un proprio tid di tipo $pthread\_t$.

Per identificare i thread si possono usare le seguenti funzioni:

```
// restituisce un valore diverso da zero se uguali, 0 se diversi
int pthread_equal(pthread_t t1, pthread_t t2);

//restituisce il tid del thread corrente
pthread_t pthread_self(void);
```

12.2.1 Creazione Thread

Analogamente ai processi, quando un thread viene creato esso è associato ad un pezzo di codice.
Tuttavia, solo se il processo ospite (e quindi il suo corrispondente programma) è single-threaded, il thread è associato all'intero programma.
Se il processo ospite è multi-threading, ciascun thread di tale processo è associato ad una funzione da eseguire, detta funzione di avvio.

Anche i thread, come i processi, possono assumere gli stati running, sleeping, blocked e terminated.
Quando un thread termina la situazione è analoga al caso dei processi: è il programmatore che deve preoccuparsi affinché tutte le risorse impegnate dal thread siano rilasciate al sistema, facendo in modo che venga effettuata una wait a livello di thread.

Quando un programma viene mandato in esecuzione tramite una chiamata exec, viene creato un singolo thread, detto thread principale o iniziale; ulteriori thread vanno creati esplicitamente.

Ogni thread ha numerosi attributi, da assegnare alla creazione: livello di priorità, dimensione iniziale della pila, etc…
All'atto della creazione di un thread è necessario specificare la funzione di avvio: il thread inizierà la propria esecuzione richiamando la funzione di avvio.

Per creare thread addizionali relativi ad uno stesso processo, Posix prevede la funzione:

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*start_func)(void *), void *arg);
```

- Se la chiamata ha successo, tid punta al thread ID;
- $attr$ permette di specificare gli attributi del thread (se $attr = NULL$, gli attributi sono quelli di default)
- $start\_func$ è l'indirizzo dell'argomento accettato dalla funzione di avvio;
- Restituisce 0 in caso di successo, un intero positivo (secondo le convenzioni di <sys/errno.h>) in caso di errore.
- arg è l'argomento passato alla funzione start.

12.2.2 Risorse condivise

Siccome i thread condividono la memorai, è meglio non usare una variabile globale (come errno) per i codici d'errore; quindi, le funzioni pthread restituiscono direttamente un codice d'errore (come ad esempio pthread_create).

La funzione $\text{char } *strerror(\text{int } n)$; restituisce un messaggio corrispondente al codice d'errore n.
I thread di uno stesso processo condividono: la memoria; il pid e il ppid; i file descriptor; le reazioni ai segnali (cioè, le chiamate a signal influenzano tutti i thread).
I thread non condividono: lo stack.

I thread condividono lo stesso spazio di indirizzamento, e quindi vedono le stesse variabili: se uno dei due modifica una variabile, la modifica è vista anche dagli altri thread.

Nel caso dei processi tradizionali, una cosa simile è ottenibile solo usando esplicitamente un segmento di memoria condivisa.

Ma i thread di un task possono condividere variabili in maniera ancora più semplice, usando variabili globali.

12.2.3 Terminare un thread

- Invocare $exit()$ (_exit, _Exit) fa terminare l'intero processo.
- Analogamente un segnale ad un thread uccide il processo.
- Per terminare solo il thread corrente, si può:
    - Invocare $return$ dalal rutine di start, il valore di ritorno è l'exit code
    - Invocare pthread_exit
    - Un altro thread del processo può chiamare pthread_cancel

Un thread può richiedere esplicitamente la propria terminazione grazie alla chiamata seguente, lasciando traccia del proprio stato di terminazione per quei thread che attendono per lui:

```
#include <pthread.h>

int pthread_exit(void *status);
```

- $status$ punta all'oggetto che definisce lo stato di terminazione del thread. Quest'ultimo non deve essere una variabile locale al thread chiamante, pena la sua scomparsa alla terminazione del thread stesso.

Un thread può attendere per la terminazione di un altro thread relativo allo stesso processo:

```
#include <pthread.h>

int pthread_join(pthread_t *tid, void **status);
```

- tid è l'ID del thread del quale si vuole attendere la terminazione;
- $status$ punta al valore restituito dal thread per cui si è atteso, indicante il suo stato di terminazione (se $status = NULL$, tale stato non viene restituito);
- Restituisce 0 in caso di successo, un intero positivo (secondo le convenzioni di <sys/errno.h>) in caso di errore.

12.2.4 Cancellare un thread

```
int pthread_cancel(pthread_t tid);
```

- Chiede che il thread specificato da tid venga terminato (non aspetta la terminazione)
- Restituisce 0 se OK, un codice d'errore altrimenti

In ogni istante, un thread può essere cancellabile o non cancellabile. Quando partono tutti i thread sono cancellabili.

Quando un altro thread chiama pthread_cancel:

- Se il thread è cancellabile, viene cancellato
- Se non è cancellabile, la richiesta di cancellazione viene memorizzata, in attesa che il thread diventi cancellabile

In taluni casi è opportuno far sì che lo stato di terminazione di un thread T non venga memorizzato fintanto che un altro thread T' relativo allo stesso processo attenda per T, ma sia invece cancellato subito dopo la terminazione di T:

```
int pthread_detach(pthread_t *tid);
```

- tid è l'ID del thread che si vuole distaccare;
- Restituisce 0 in caso di successo, un intero positivo (secondo le convenzioni di <sys/errno.h>) in caso di errore.

Se non ci interessa il valore di ritorno di un thread, conviene crearlo in detached state; però, poi non possiamo chiamare pthread_join.

12.2.5 Thread e segnali

Le chiamate a signal influenzano tutti i thread. Se arriva un segnale a un processo, succede che:

- Se il processo ha impostato un handler, il segnale arriva ad uno qualunque dei thread (che esegue l'handler)
- Se invece la reazione al segnale consiste nel terminare il processo, tutti i thread vengono terminati.

```
int pthread_kill(pthread_t tid, int signo);
```

- Manda il segnale $signo$ al thread specificato da tid
  - Se è impostato un handler, viene eseguito nel thread tid
  - Se non è impostato un handler, e il comportamento di default è di terminare il processo, vengono comunque terminati tutti i thread
- Restituisce 0 se OK, un codice d'errore altrimenti.

12.2.6 Attributi di un thread

Un thread può essere creato in "detached state" (stato sconnesso).
Un thread può bloccare i tentativi di essere cancellato (cancellabilità).
Altri attributi: posizione e dimensione dello stack; attributi real-time.

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- Inizializza e distrugge una struttura per gli attributi di un thread. Uso:
  - Si alloca una struttura pthread_attr_t (struttura opaca)
  - Si chiama pthread_attr_init
  - Si modificano gli attributi contenuti nella struttura usando apposite funzioni (vedere dopo)
  - Si passa la struttura a pthread_create
  - Si distrugge la struttura con pthread_attr_destroy
- Restituiscono 0 se OK, un codice d'errore altrimenti.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

- Imposta l'attributo $detachstate$ della struttura puntata da $attr$
- L'argomento $detachstate$ può essere:
  - PTHREAD_CREATE_JOINABLE (default)
  - PTHREAD_CREATE_DETACHED
- Restituisce 0 se OK, un codice d'errore altrimenti

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

- Imposta la cancellabilità a $state$ e restituisce la vecchia cancellabilità in $oldstate$.
- $state$ e $oldstate$ possono assumere i valori:
  - PTHREAD_CANCEL_ENABLE
  - PTHREAD_CANCEL_DISABLE
- Restituisce 0 se OK, un codice d'errore altrimenti

12.3 Sincronizzazione

I thread condividono la memoria, quindi può succedere che più thread accedano a stessi dati causando race condition.

Dunque, sono necessari meccanismi di sincronizzazione:

- mutex (semaforo binario)
- condition variable

La sincronizzazione è necessaria quando si accede a variabili condivise con operazioni non atomiche.

Se ad esempio abbiamo l'istruzione $x{+}{+}$; questa potrebbe essere scomposta in: carica la variabile; incrementa l'accumulatore; memorizza l'accumulatore. Può quindi succedere che un thread si inserisca in mezzo a queste operazioni causando race condition.

12.3.1 Mutex

Un mutex Posix è caratterizzato dalle seguenti proprietà:

- È una variabile di tipo pthread_mutex_t che può essere inizializzata con diversi attributi
- Può assumere solo i due stati alternativi **locked** (chiuso) o **unlocked** (aperto)
- Può essere chiuso solo da un processo alla volta, ed il processo che chiude il mutex ne diviene il possessore fino alla successiva chiusura
- Può essere riaperto solo dal proprio possessore
- Deve essere condiviso tra tutti i processi che intendono sincronizzare l'accesso ad una regione critica (**blocco cooperativo**).

Un mutex è un semaforo binario (locked o unclocked) mantenuto in una struttura pthread_mutex_t che va allocata e inizializzata:

- Se la struttura è allocata statisticamente: pthread_mutex_t c = PTHREAD_MUTEX_INITIALIZER
- Se la struttura è allocata dinamicamente: chiamare pthread_mutex_init

Inizializzare e distruggere un mutex:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Inizializza e distrugge un mutex, rispettivamente
- Quando inizializzato è in stato unlocked
- Restituiscono 0 se OK, un codice d'errore altrimenti
- $attr$ può essere NULL (attributi di default)

Usare i mutex:

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

- Acquisiscono e rilasciano il semaforo
- Restituiscono 0 se OK, un codice d'errore altrimenti
- Se il semaforo è locked:
  - $lock$ blocca i thread finché il semaforo si libera
  - $trylock$ invece non blocca, ma restituisce subito l'errore EBUSY

La sincronizzazione può essere:

- Per sezione critica
  - Solo quando una struttura condivisa viene modificata in un unico punto nel codice
  - È sufficiente associare un mutex alla sezione critica
- Per "struttura"
  - Quando la struttura può essere modificata in più punti nel codice
  - Utile se più struttura devono essere condivise contemporaneamente
  - È necessario associare un mutex alla "struttura"

Attributi Mutex:

```
#include <pthread.h>

// crea in attr un attributo di mutex
int pthread_mutexattr_init(pthread_mutexaatr_t *attr);

// dealloca l'attributo di mutex in attr
int pthread_mutexattr_destroy(pthread_mutexaatr_t *attr);
```

- *attr* è un attributo di mutex come quelli richeisti dalla pthread_mutex_init( )
- Entrambe restituiscono sempre 0
- Attualmente LinuxThreads supporta solo l'attributo relativo al tipo di mutex

12.3.2 Tipologie di Mutex

- **fast**: semantica classica, pertanto un thread che esegue due mutex_lock() consecutivi sullo stesso mutex causa uno stallo
- **recursive**: conta il numero di volte che un thread blocca il mutex e lo rilascia solo se esegue un pari numero di mutex_unlock( )
- **error-checking**: controlla che il thread che rilascia il mutex sia lo stesso thread che lo possiede

Inizializzazione di un mutex:

- statica, macro per inizializzare un mutex:

```
fastmutex = PTHREAD_MUTEX_INITIALIZER;
recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

- dinamica, chiamata di libreria:

```
int pthread_mutex_init(pthread_mutex_t *mp,
                       const pthread mutexattr t *mattr);
```

  - *mp* è un mutex precedentemente allocato
  - *mattr* sono gli attributi del mutex: NULL per il default
  - Restituisce sempre 0

lock( ): blocca un mutex

- Se era sbloccato il thread chiamante ne prende possesso bloccandolo immediatamente e la funzione ritorna subito
- Se era bloccato da un altro thread il thread chiamante viene sospeso sino a quando il possessore non lo rilascia
- Se era bloccato dallo stesso thread chiamante dipende dal tipo di mutex
  - fast: deadlock, perché il chiamante stesso, che possiede il mutex, viene sospeso in attesa di un rilascio che non avverrà mai
  - error-checking: la chiamata fallisce
  - recursive: la chiamata ha successo, ritorna subito, incrementa il contatore del numero di lock eseguiti dal thread chiamante

N.B.: una soluzione base per i deadlock (condizione di attesa ciclica) è acquisire i mutex sempre nello stesso ordine. Ma ciò non è sempre possibile; quindi, può essere necessario utilizzare algoritmi specifici e pthread_mutex_trylock

unlock( ): sblocca un mutex che si assume fosse bloccato. Ad ogni modo la semantica esatta dipende dal tipo di mutex:

- fast: il mutex viene lasciato sbloccato e la chiamata ha sempre successo
- recursive: si decrementa il contatore del numero di lock eseguiti dal thread chiamante sul mutex, e lo si sblocca solamente se tale contatore si azzera

- error-checking: sblocca il mutex solo se al momento della chiamata era bloccato e posseduto dal thread chiamante, in tutti gli altri casi la chiamata fallisce senza alcun effetto sul mutex

Funzioni per fissare/conoscere il tipo:

```
#include <pthread.h>

// restituisce 0 se OK, codice d'errore altrimenti
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int kind);

// restituisce sempre 0
int pthread_mutexattr_gettype(pthread_mutexattr_t *attr, int kind);
```

- fast: PTHREAD_MUTEX_FAST_NP
- recursive: PTHREAD_MUTEX_RECURSIVE_NP
- error-checking: PTHREAD_MUTEX_ERRORCHECK_NP

Limiti del mutex:

- Se un thread attende il verificarsi di una condizione su una risorsa condivisa con altri thread. Con i soli mutex sarebbe necessario un ciclo del tipo:

```
while(1) {
    lock(mutex);
    if (/*condizione sulla risorsa condivisa*/)
        break;
    unlock(mutex);
    // ...
}
/*sezione critica*/;
unlock(mutex);
```

  - Attesa, e verifica ciclica sulla variabile, finché la condizione verificata non rompe il ciclo, quindi sblocco.
- I mutex risultano inefficienti e ineleganti come strumento di cooperazione.

12.3.3 Condition variable

Le condition variable sono strumenti di sincronizzazione tra thread che consentono di: attendere passivamente il verificarsi di una condizione su una risorsa condivisa; segnalare il verificarsi di tale condizione.

La condizione interessa sempre e comunque una risorsa condivisa; pertanto, le condition variable possono sempre associarsi al mutex della stessa per evitare race condition sul loro utilizzo.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Le condition variable servono per attendere che una condizione si verifichi, escludendo race conditions. Una variabile di condizione è mantenuta in una struttura pthread_cond_t che va allocata ed inizializzata:

- Se la struttura è allocata staticamente: pthread_cond_t c = PTHREAD_COND_INITIALIZER
- Se la struttura è allocata dinamicamente: chiamare pthread_cond_init

Per inizializzare e distruggere una condition variable:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond)
```

- Per la destroy non devono esistere thread in attesa
- Restituiscono 0 se OK, un codice d'errore altrimenti
- $attr$ può essere NULL (attributi di default)

Usare una condition variable:

Thread che aspetta una condizione

```
mutex_lock(m)

while (/*condizione false*/)
    cond_wait(c, m)

// fà qualcosa

mutex_unlock(m)
```

Thread che rende la condizione vera

```
mutex_lock(m)

// rendi la condizione vera

cond_broadcast(c)

mutex_unlock(m)
```

Attendere una condition variable:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- Attende che $cond$ sia segnalata come vera
- Restituisce 0 se OK, un codice d'errore altrimenti
- Il mutex protegge la condizione
  - Deve essere acquisito prima di chiamare $cond\_wait$
  - Durante l'attesa, $cond\_wait$ rilascia il mutex
  - Finita l'attesa, $cond\_wait$ riprende il mutex
- Al momento della chiamata il mutex deve essere bloccato
- Rilascia il mutex, il thread chiamante rimane in attesa passiva di una segnalazione sulla condition variable
- Nel momento di una segnalazione, la chiamata restituisce il controllo al thread chiamante, e questo rientra in competizione per acquisire il mutex

Il seguente programma chiama $do\_work()$ mentre $flag$ è settato, altrimenti si blocca in attesa che venga segnalato un cambiamento nel suo valore:

```
#include <pthread.h>

extern void do_work();
int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag() {
    pthread_mutex_init(&thread_flag_mutex, NULL);
    pthread_cond_init(&thread_flag_cv, NULL);
    thread_flag = 0;
}

void* thread_function(void *thread_arg) {
    while (1) {
        // attende sengale sulla variabile condizione
        pthread_mutex_lock(&thread_flag_mutex);

        while (!thread_flag)
            pthread_cond_wait(&thread_flag_cv, &thread_flag_mutex);

        pthread_mutex_unlock(&thread_flag_mutex);

        do_work(); // fa qualocsa
    }
    return NULL;
}
```

Segnalazione:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Uno dei thread che sono in attesa sulla condition variable viene risvegliato
  - Se più thread sono in attesa, ne viene scelto uno ed uno solo effettuando una scelta non deterministica
  - Se non ci sono thread in attesa, non accade nulla

Timed Wait:

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

- $abstime$ è la specifica di un tempo assoluto; es: $\{0,0\}$
- Permette di restare in attesa fino all'istante specificato restituendo il codice di errore ETIMEDOUT al suo scadere
- Restituisce 0 in caso di successo oppure un codice d'errore

Broadcast:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Causa la ripartenza di tutti i thread che sono in attesa su $cond$
  - Se non ci sono thread in attesa, non succede niente
- Restituiscono 0 in caso di successo oppure un codice d'errore

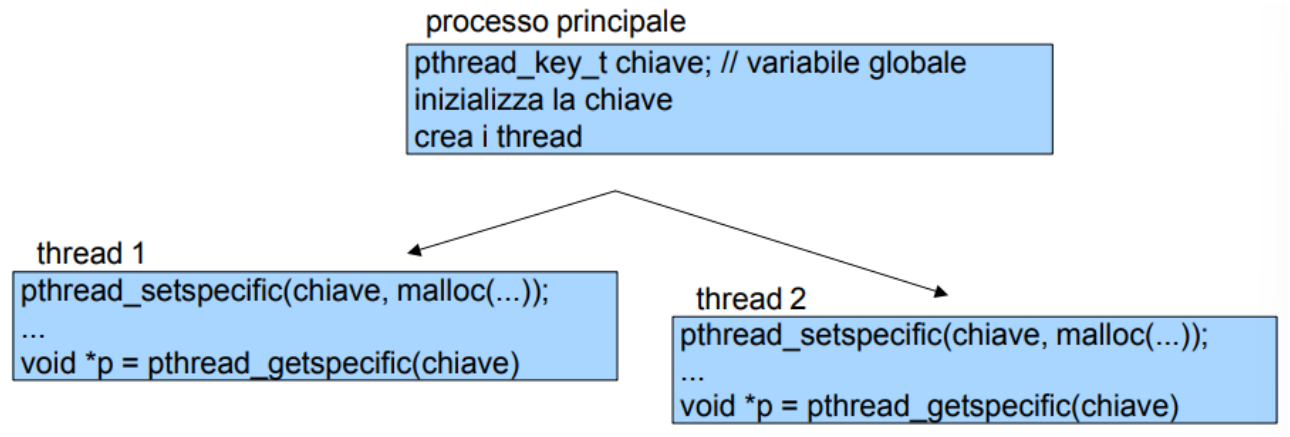
12.4 Thread Specific Data

I thread condividono il segmento di dati ma ci sono appositi meccanismi per avere dati privati (TSD).

Ogni thread possiede un'area di memoria privata, la TSD area, indicizzata da chiavi.
La TSD area contiene associazioni tra le chiavi ed un valore di tipo void^*

- Diversi thread possono usare le stesse chaivi ma i valori associati variano di thread in thread
- Inizialmente tutte le chiavi sono associate a NULL.

Associare a una stessa chiave, dati diversi per ciascun thread:



Funzioni per TSD:

- Creare una chiave per dati privati:

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

  - key è l'indirizzo della chiave da inizializzare
  - $destructor$ è un puntatore alla funzione distruttore che deve essere chiamata alla terminazione di un thread (pthread_exit( ))
  - Restituisce 0 se OK, un codice d'errore altrimenti

- Deallocare una chiave TSD:
  ```
  int pthread_key_delete(pthread_key_t key);
  ```
  - Restituisce 0 se OK, un codice d'errore altrimenti

- Associare un certo valore ad una chiave TSD:
  ```
  int pthread_key_setspecific(pthread_key_t *key, const void *val);
  ```
  - Associa l'indirizzo val alla chiave key, per il thread chiamante
  - Restituisce 0 se OK, un codice d'errore altrimenti

- Ottenere il valore associato ad una chiave TSD:
  ```
  int pthread_key_getspecific(pthread_key_t *key);
  ```
  - Restituisce l'indirizzo associato alla chiave key nel thread chiamante
  - Se nessun indirizzo è stato associato a key restituisce NULL

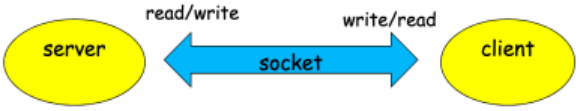
12.5 Esercizio

- realizzare un programma che accetta da riga di comando due numeri interi n ed m, e crea n produttori ed m consumatori
- produttori e consumatori condividono un array di 100 interi
- ogni produttore aspetta un numero casuale di secondi tra 1 e 10, e poi produce (cioè inserisce nell'array) da 1 a 5 numeri casuali. Se il produttore trova l'array pieno, salta il turno
- ogni consumatore aspetta che ci sia un numero da consumare, e poi stampa a video il proprio tid e il valore consumato

13. Socket

13.1 Panoramica

Le socket definiscono un canale di comunicazione bidirezionale. Le socket possono essere locali (stessa macchina) oppure di tipo TCP/UDP (tramite rete).



13.1.1 Domini e Stili di Comunicazione

I socket permettono di specificare il tipo di comunicazione attraverso le nozioni di dominio e di stile;

Il **dominio** di un socket equivale alla scelta di una famiglia di protocolli. Le correnti release del kernel di molti Unix prevedono ventisei diverse famiglie.
Ogni domino è individuato univocamente da un intero non negativo, cui corrispondono una o più costanti simboliche del tipo PF_$nomefamiglia$, definite nell'header <sys/socketh>.

Tra i domini più importanti si ricordano:

- PF_LOCAL (o PF_UNIX, o PF_FILE), per le comunicazioni in locale tramite filesystem (reale o virtuale);
- PF_INET, la famiglia TCP/IP con Ipv4;
- PF_INET6, la famiglia TCP/IP con Ipv6;
- PF_IPX, famiglia di protocolli per reti Novell;
- PF_APPLETALK, famiglia di protocolli per reti Appletalk.

A ciascun dominio possono corrispondere in teoria uno o più schemi di indirizzamento, ossia tipi di indiriziz, per cui i socket prevedono la nozione di famiglia di indirizzi, ciascuna individuata univocamente attraverso un numero non negativo, cui corrisponde (sempre tramite l'header socket.h) una costante simbolica.

Lo **stile di comunicazione** è individuato da costanti simboliche del tipo SOCK_$nomestile$, definite anch'esse nell'header <sys/socketh>.

Gli stili principali sono:

- SOCK_STREAM, corrispondente ad un canale di trasmissione bidirezionale a flusso, con connessione sequenziale ed affidabile;
- SOCK_DGRAM, che consiste in una trasmissione a pacchetti (datagram) di lunghezza max, prefissata, senza connessione e non affidabile;
- SOCK_RAW, per l'accesso a basso livello ai protocolli di rete ed alle varie interfacce.

Assegnato un dominio, la scelta dello stile di comunicazione corrisponde in pratica ad individuare uno specifico protocollo tra quelli appartenenti al dominio.
Non tutte le combinazioni "dominio-stile" sono valide, in quanto non è detto che in una famiglia di protocolli esista un elemento per ciascuno dei possibili stili.

13.1.2 Indirizzamento

In Unix le strutture dati per la gestione degli indirizzi sono progettate per essere adatte a diversi domini di comunicazione e protocolli. Ad esempio: sa_family = AF_INET; sa_family = PF_LOCAL,PF_UNIX.

Generico indirizzo:

```c
#include <sys/socket.h>

struct sockaddr {
    unsigned short sa_family; // family address, AF_xxx
    char sa_data[14]; // 14 bytes of protocol address
};
```

Siamo interessati a $sa\_family = \text{AF\_INET}$, cioè al dominio "internet". In tal caso servono 2 byte per un numero di porta e 4 byte per un indirizzo IP:

```
#include <netinet/in.h>

struct sockaddr_in {
    unsigned short sa_family; // family address
    unsigned short int sin_port; // Port numbet: 2byte
    struct in_addrt sin_addr; // IP internet address: 4byte
    unsigned char sin_zero[8] // 8 bytes
};
```

- Si noti che sockaddr_in ha lo stesso numero di byte di sockaddr visto precedentemente.

13.1.3 Funzioni Socket

Creare una socket:

```
#include <sys/socket.h>

int socket(int family, int type, int protocol);
```

- Apre una socket, allocando una voce nella tabella dei file del kernel e permettendo la specifica del dominio, dello stile e del protocollo di comunicazione
- $family$ è la famiglia cui appartiene il protocollo (dominio)
  - Nel nostro caso PF_LOCAL oppure PF_INET
- $type$ definisce il tipo di comunicazione
  - Tipo $=$ SOCK_STREAM
- $protocol$ specifica il protocollo della famiglia
  - Protocollo $= 0$
- Restituisce -1 in caso di insuccesso, un numero positivo rappresentante il socket descriptor altrimenti.

Un programma che usa socket deve includere: <sys/types.h> e <sys/socket.h>.
Inoltre se usa socket del domino AF_LOCAL o AF_UNIX deve anche includere <sys/un.h>; mentre, se usa socket del domino AF_INET allora vanno inclusi <netinet/in.h>, <arpa/inet.h> e <arpa/netdb.h>.

Funzione bin:

```
#include <sys/socket.h>

int bind(int sd, const struct sockaddr *my_addr, int addrlen);
```

- Assegna un indirizzo locale alla socket (o all'insieme di socket) individuato dal socket descriptor sd
- sd è un socket descriptor ottenuto da una precedente chiamata a socket
- my_addr è l'indirizzo locale, specificato secondo il formato caratteristico della famiglia di protocolli cui sd è riferito;
- $addrlet$ è la lunghezza del suddetto indirizzo;
- Restituisce -1 in caso di errore, 0 altrimenti.

Leggere da un socket:

- Si può usare *read*
- Se non ci sono dati da leggere, *read* blocca il processo in attesa di dati (come per una pipe)
- È normale ottenere meno bytes di quelli richiesti (come per una pipe)
- Ottenere 0 bytes significa che il socket è vuoto ed inoltre è stato chiuso

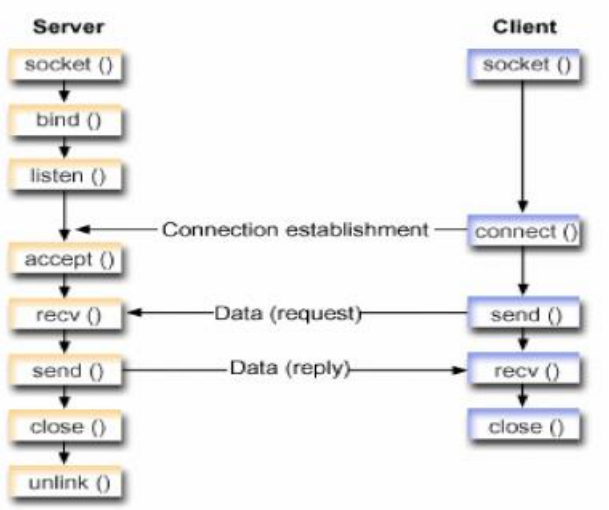
Scrivere su un socket:

- Si può usare *write*
- È normale riuscire a scrivere meno bytes di quelli richiesti

- Se il socket è stato chiuso, il processo riceve il segnale SIGPIPE
    - Di default, questo segnale termina il processo
    - Se si ignora questo segnale ($signal(\text{SIGPIPE,SIG\_IGN})$) write restituisce -1 e imposta $errno = \text{EPIPE}$ oppure, si può catturare il segnale.

13.2 Socket Server-Client

13.2.1 Flusso Socket Server-Client



13.2.2 Lato Server

Crea il socket	Funzione *socket*
Gli assegna un indirizzo	Funzione *bind*
Si mette in ascolto	Funzione *listen*
Accetta nuove connessioni	Funzione *accept*
Chiude il socket	Funzione *close*
Cancella il file se socket locale	Funzione *unlink*

Panoramica: il server

Funzione listen:

```
int listen(int sd, int backlog);
```

- Utilizzata da un server in caso di comunicazione orientata alla connessione;
- Pone il socket specificato da sd in modalità passiva, ossia in ascolto di eventuali connessioni, predisponendo per esso una coda per le connessioni in arrivo di lunghezza pari a $backlog$;
- $backlog$ rappresenta il numero massimo di connessioni pendenti accettate. Se tale numero è superato, il client riceverà un errore, oppure (nel caso di protocolli come TCP che prevedono la ritrasmissione) la richiesta del client verrà ignorata in modo da poter essere ritentata;
- Restituisce 0 in caso di successo, -1 altrimenti.

Accettare nuove connessioni:

```
int accept(int sd, const struct sockaddr *addr, int addrlen);
```

- Utilizzata da un server in caso di comunicazione orientata alla connessione, restituisce un nuovo socket descriptor su cui si potrà operare per effettuare la comunicazione con un client;
- Estrae la prima connessione relativa al socket descriptor sd in attesa sulla coda delle connessioni, definita grazie ad una precedente chiamata a $listen$ per sd;
- Nella struttura $addr$ e nella variabile $addrlen$ vengono restituiti, rispettivamente, l'indirizzo e la lunghezza di tale indirizzo per il client che si è connesso;

- Restituisce un nuovo socket descriptor in caso di successo, -1 altrimenti. Il nuovo socket eredita le caratteristiche di sd.

Struttura di un server:

```
int fd1, fd2;
struct sockaddr_un my_addr;

my_addr.sun_family = AF_LOCAL;
strcpy(my_addr.sun_path, "/tmp/my_socket");

fd1 = socket(PF_LOCAL, SOCK_STREAM, 0); // crea socket locale

bind(fd1, (struct sockaddr*)&my_addr, sizeof(my_addr));

listen(fd1, 5); // 5 è la dimensione della coda di attesa

fd2 = accept(fd1, NULL, NULL);
...
close(fd2);
close(fd1);
unlink("/tmp/my_socket");
```

13.2.2 Lato Client

Crea il socket	Funzione *socket* (uguale a quella server)
Si connette ad un server	Funzione *connect*
Chiude il socket	Funzione *close*

Panoramica: il client

Funzione connect:

```
int connect(int sd, const struct sockaddr *serv_addr, int addrlen);
```

- Utilizzata dal lato client, ha due funzionalità differenti a seconda che la comunicazione sia orientata o meno alla connessione;
- In entrambi i casi collega il socket locale di identificativo sd al socket remoto di indirizzo $serv\_addr$
- Nel caso di comunicazioni con connessione attiva la procedura di avvio della connessione (il *three-way handshake* per il TCP) e ritorna solo quando la connessione è stabilita o si è verificato un errore
- Restituisce 0 in caso di successo, -1 altrimenti.

Funzione close:

```
int close(int sock_fd);
```

- Restituisce 0 in caso di successo, -1 in caso di errore
- Serve per dichiarare che non si vuole più utilizzare il socket
- Più processi dello stesso host possono condividere la stessa socket
    - Solo se tutti avranno eseguito una $close()$ il sistema operativo provvederà a chiudere la connessione (necessariamente di tipo SOCK_STREAM) concludendo il protocollo TCP
- La chiusura è simmetrica: la connessione sarà effettivamente chiusa quando sarà stata chiusa sia sul server che sul client.

Struttura di un client:

```
int fd;
struct sockaddr_un addr;

addr.sun_family = AF_LOCAL;
strcpy(addr.sun_path, "/tmp/my_socket");

fd = socket(PF_LOCAL, SOCK_STREAM, 0); // crea socket locale
connect(fd, (struct sockaddr*)&addr, sizeof(addr));

...
close(fd);
```

13.3 Socket TCP/IP

13.3.1 Specificare indirizzi IP

Usando la notazione *dotted* (puntata):

```
struct sockaddr_in addr;

if (inet_aton("143.225.5.2", &addr.sin_addr) == 0) {
    perror("inet_aton");
    exit(1);
}
```

- $inet\_aton$: ascii to network
- Riempie direttamente una struttura in_addr
- Restituisce 0 in caso di errore

Impostare l'indirizzo:

```
struct sockaddr_in addr;

// host byte order
addr.sin_family = AF_INET;

// short, network byte order
addr.sin_port =  htons(MYPORT);

// long, network byte order
inet_aton("10.12.110.57", &(addr.sin_addr));

// a zero tutto il resto
memset(&(addr.sin_zero), '\0', 8);
```

Indirizzi TCP/IP per il server (bind):

```
struct sockaddr_in addr;

addr.sin_family = AF_INET;
addr.sin_port = htons(5200);
addr.sin_addr.s_addr = htonl(INADDR_ANY);

bind(fd, (struct sockaddr*) &addr, sizeof(addr));
```

- Il server chiama bind per stabilire su quale indirizzo mettersi in ascolto.
- Di solito, il server sceglie solo la porta
- Come indirizzo IP, sceglie INADDR_ANY, così accetta connessioni dirette a qualunque indirizzo (uno stesso host può avere più indirizzi IP)

Indirizzi TCP/IP per il client (connect):

```
struct sockaddr_in addr;

addr.sin_family = AF_INET;
addr.sin_port = htons(5200);
inet_aton("142.225.5.3", &addr.sin_addr);

connect(fd, (struct sockaddr*) &addr, sizeof(addr));
```

13.3.2 Leggere e Scrivere su Socket

Trasmissione dati:

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

- Invia il contenuto del buffer buf al socket specificato
- Si usa esclusivamente con SOCK_STREAM
- Restituisce il numero di byte inviati oppure -1 in caso di errore
- È la stessa funzione che consente la scrittura su file

Ricezione:

```
ssize_t read(int fd, const void *buf, size_t count);
```

- Solo per socket connessi (SOCK_STREAM)
- Legge un messaggio di lunghezza massima len dal socket
- Se non c'è alcun messaggio, il programma rimane sospeso (chiamata bloccante)
- La funzione ritorna il numero id byte letti, -1 in caso di errore
- È la stessa funzione che consente la lettura da un file

Esistono altre funzioni specifiche per leggere e scrivere su socket:

- 3 modalità di **send**:
  - send è come write (richiede connessione stabilita), ma ha flag per specificare modalità di scrittura
  - sendto permette la scrittura su connectionless socket
  - sendmsg per specificare buffer multipli
- 3 modalità di **receive**
  - recv come read con flag
  - recvfrom per ottenere indirizzo della fonte
  - rcvmsg per ricevere da buffer multipli

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

int send(int s, const void *msg, int len, unsigned int flags);

int sendto(int s, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, int tolen)
```

- $send()$ può essere utilizzata sole se s è stato connesso
- $sendto()$ può essere sempre utilizzata perché richiede di specificare l'indirizzo di destinazione
- Restituiscono -1 in caso di errore oppure il numero di byte effettivamente trasmessi
- $flags$ si può lasciare a zero

```c
#include <sys/types.h>
#include <sys/socket.h>

int recv(int s, void *buf, int len, unsigned int flags);

int recvfrom(int s, void *buf, int len, unsigned int flags
             struct sockaddr *from, int *fromlen);
```

- Ricevono in buf non più di len byte. Se $from$ non è NULL, la struttura $sockaddr$ verrà riempita con l'indirizzo del mittente.
- Restituiscono -1 in caso di errore oppure il numero di byte effettivamente ricevuti
- $flags$ si può lasciare a zero
- Se con $recvfrom$ non arriva alcun messaggio, il programma rimane sospeso (la chiamata è bloccante)

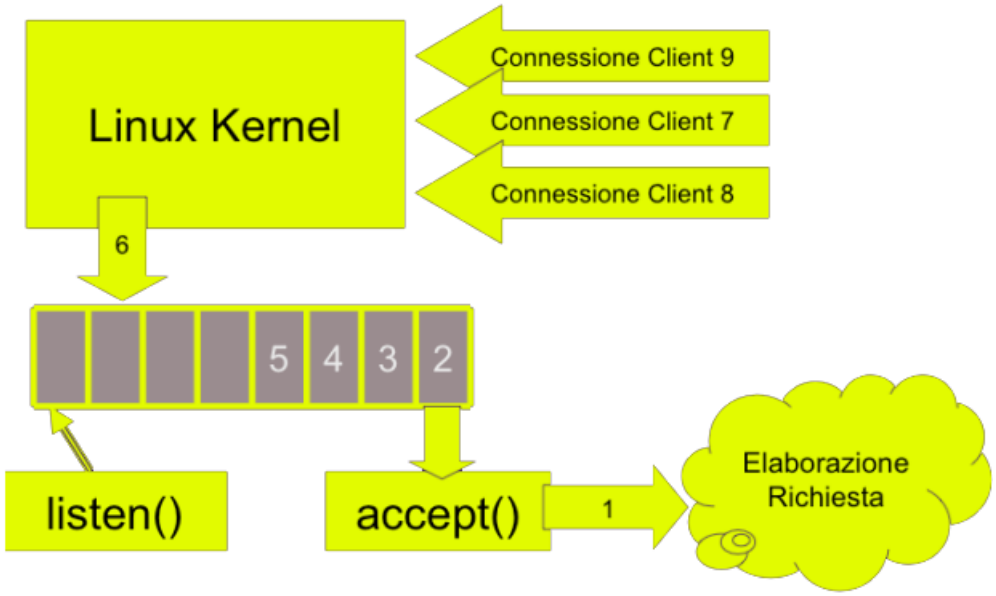
13.3.3 Server a Programmazione Concorrente

Un generico server attende le richieste di connessione su una determinata porta.
Possono arrivare richieste concorrenti e da molteplici client.

Una soluzione senza programmazione concorrente:

- Client serviti uno alla volta: finché una connessione non è terminata, non vengono serviti altri client interessati al servizio.
- N.B.: comunque è il SO che gestisce le richieste concorrenti ed in effetti le serializza

La coda di connessione:

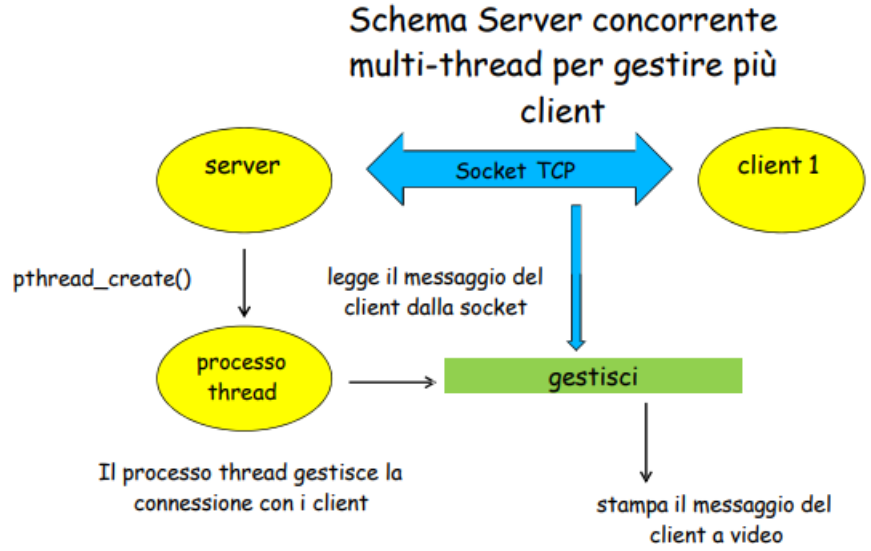


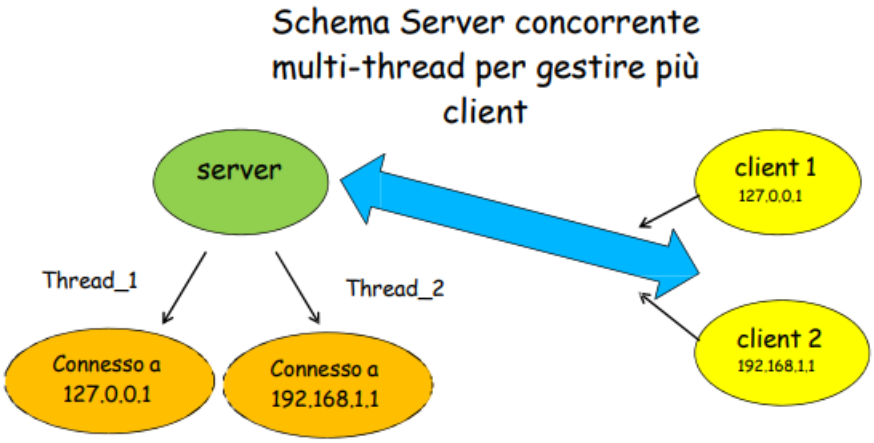
Un server che gestisce sequenzialmente i vari client è insoddisfacente:

- Il tempo di attesa di ogni client potrebbe risultare eccessivo poiché dipende dalle durate delle altre connessioni
- Se le connessioni durassero molto, il server ben presto diverrebbe indisponibile anche solo ad accordare le nuove richieste di connessioni

Nei server reali le richieste di vari client devono essere gestite concorrentemente.

13.3.4 Comunicazione Client/Server con Socket TCP





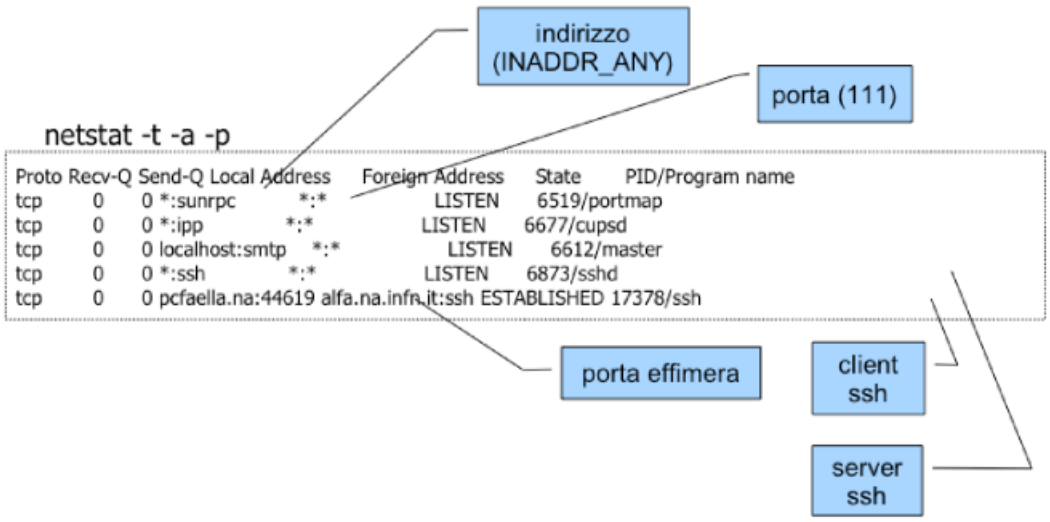
13.3.5 Comandi utili

Risoluzione indirizzi:

- $gethostbyname()$: dato un hostname, restituisce una struttura dati che specifica anche i suoi indirizzi IP
- $gethostbyaddr()$: dato un indirizzo IP, restituisce una struttura dati che specifica anche il suo hostname
- $getservbyname()$: dato un nome di servizio e di protocollo, restituisce una struttura dati che specifica i suoi nomi e l'indirizzo di porta
- $gethostname()$ e $getdomainname()$: restituiscono l'hostname della macchina
    - $herror()$ stampa un messaggio di errore per $gethostname()$

netstat [-t] [-all] [-p] [-n]

- elenca tutti i socket di rete del sistema (non riguarda i socket locali)
- -t mostra solo i socket TCP, cioè quelli con famiglia = PF_INET e tipo = SOCK_STREAM
- -all (oppure -a) mostra anche i socket in ascolto
- -p specifica il pid del processo che ha creato ciascun socket
- -n mostra gli indirizzi e le porte in formato numerico (invece di simbolico)



Altri comandi utili:

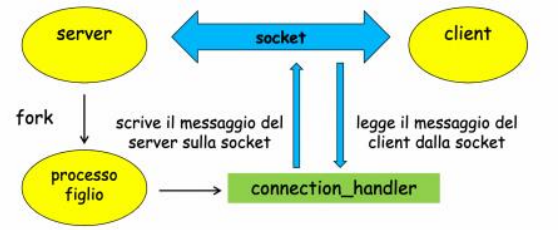
- /sbin/ifconfig
  - Mostra l'indirizzo IP della macchina corrente
- nslookup $< nome\ di\ dominio >$
  - fornisce l'indirizzo IP di un host del quale conosciamo il nome
  - esempio: nslookup $www.unina.it$

13.4 Esercizi

13.4.1 Esercizio 1

Scrivere due programmi C, $server.c$ e $client.c$, che comunicano tramite socker.

Il $server.c$ crea una socket locale, ed ogni volta che instaura una connessione con un client, mediante fork crea un nuovo processo. Tale processo, dovrà leggere sul socket il messaggio scritto dal client e scrivere il messaggio che il server manda al client.



Il $client.c$ dovrà connettersi al socket locale su cui scriverà il messaggio da mandare al server.

N.B.: lanciare l'esecuzione dei due programmi su due shell distinte della stessa macchina.

```
Esecuzione
$ ./server  (shell1)  MESSAGGIO DA CLIENT: Saluti da client
$ ./client  (shell2)  MESSAGGIO DA SERVER: Saluti dal server
```

13.4.2 Esercizio 2

Ad ogni nuova connessione, il server scrive sul socket l'ora corrente, poi chiude la connessione e si rimette in attesa di nuove connessioni.

Implementare anche un client che riceve l'orda dal server e la stampa sul terminale.
Implementare un server che fornisce ai client l'ora esatta, usando un socket locale.

Suggerimento: una stringa che rappresenta l'ora esatta si può ottenere come segue:

```c
#include <time.h>

char buffer[26];
time_t ora;
time(&ora);
printf("Ora esatta: %s\n", ctime_r(&ora, buffer));
```

- La funzione $time$ restituisce l'ora in un formato interno (time_t)
- La funzione $ctime\_r$ trasforma il formato interno in stringa; ha bisogno di un buffer di almeno 26 caratteri

13.4.3 Esercizio 3

Scrivere due programmi C, $chef.c$ (server) e $cook.c$ (client).

Il server crea un socket chiamato $ricetta$ e tramite essa fornisce una ricetta a tutti i client che la richiedono. La ricetta è formata da una sequenza di stringhe terminate dal caratter $'\backslash 0'$.

Il client si connette al socket $ricetta$ e legge la ricetta fornita dal server. Man mano che il client legge la ricetta la mostra sullo standard output e quindi termina.

Il server crea un processo figlio che evade la richiesta del client.

Lanciare i due programmi su due shell distinte della stessa macchina:

```
Esecuzione:
    $ ./chef.out   (server)  (shell1)
    $ ./cook .out  (client)  (shell2)

Output:
lato server: La ricetta è stata scritta nel socket
lato client: prova, prova, prova, prova,
             prova, & prova.
```

13.4.4 Soluzioni

Per le soluzioni andare al link seguente:
https://drive.google.com/file/d/1ThFIEWgr5rwZ5BcRE1oLhxGJujYQToYT/view?usp=sharing

14. Comandi UNIX

I comandi sono nell'ordine in cui sono stati presentati a lezione. Ma di seguito ho fornito un elenco in ordine alfabetico con i collegamenti ipertestuali dei comandi:

A awk

C cat chgrp chmod chown cp

G grep

E echo

H head

L ln ls

M mkdir mv

P ps

R rm rmdir

S sed sort

T tail touch

W wc

ls $[options]$ $[directory1\ file2\ \ldots]$

- Lista (in ordine alfabetico) il contenuto della o delle directories indicate
- Accetta anche nomi di file
- Senza parametri, elenca il contenuto della working directory
- Possiede numerose opzioni, alcune sono:
  - -l (**l**ong) formato esteso con informazioni aggiuntive
  - -a (**a**ll) mostra anche i file "nascosti" (dotfiles)
  - -R (**R**ecursive) visita ricorsivamente le sottodirectory
  - -i mostra l'**i**-number
  - -t (**t**ime) lista nell'ordine di modifica (prima il file modificato per ultimo)

touch $[opzioni]$ … $file$ …

- Aggiorna il tempo di accesso e modifica di $file$ al tempo corrente
- Se il file non esiste lo crea

chmod $[permission]$ $[filename1\ \ldots]$

- Change modality: attribuisce le *permissions* a *filename* (modifica i permessi dei file elencati)
- Può farlo solo il proprietario del file (o root)
- *Permissions* può essere espresso in forma ottale o **simbolica**
  - $[ugoa]$: **u**ser (proprietario), **g**roup (gruppo), **o**ther (altri utenti), **a**ll (tutti)
  - $[+-=]$: $+$ aggiungi, $-$ togli, $=$ assegna
  - $[rwx]$: **r**ead, **w**rite, e**x**ecute

chown $[options][user][:[group]]$ $file$ …

- Change owner: cambia proprietario e/o gruppo primario per uno o più file
- Se dopo ":" non segue il nome del gruppo, viene attribuito il gruppo principale cui appartiene *user*

- Se prima di :*group* non viene indicato il nome dell'utente, viene cambiato solo il gruppo primario (chgrp)

chgrp $newgroupid\ file$ …

- Change group: *newgroupid* diventa il nuovo gruppo dei *file…*
- Il comando può essere eseguito solo dal proprietario (o dal superuser)

mkdir $directory$ …

- Make directory: crea la/le directory

rmdir $directory$ …

- Remove directory: Rimuove la/le directory (deve essere vuota)

ln $name1\ name2$

- Link: associa il nuovo nome (link) name2 al file (esistente) name1, che non può essere una directory
- Tutti i link allo stesso file hanno identico status e caratteristiche
- Non è possibile distinguere la entry originaria dai nuovi link
- I link di questo tipo non possono essere fatti con file che stanno su file system diversi
- Se *name2* è una directory, il nuovo nome è *name2/name1*
- Numero links è un attributo gestito dal sistema
- Link simbolici: ln -s $name1\ name2$
    - Permette di creare link a directory
    - Permette di creare link fra file o directory che stanno su file system diversi
    - Viene creato un file *name2* che contiene il link simbolico (i.e. il path di *name1*)

mv $[options]\ name\ \ldots\ target$

- Move: muove il file o directory *name* sotto la directory target
- Se *name* e *target* non sono directories, il contenuto di *target* viene sostituito dal contenuto di *name*

cp $[options][name\ \ldots]\ target$

- Copy: come mv, ma *name* viene copiato

rm $[-r]\ name$ …

- Rimuove i files indicati
- Se un file indicato è una directory: messaggio di errore, a meno che non sia specificata l'opzione -r, nel qual caso, rimuove ricorsivamente il contenuto della directory

echo $[argomenti]$

- Visualizza gli argomenti in ordine, separati da singoli blank

cat $file$ …

- Concatenate: concatena i $file$ e li scrive sullo standard output
- Se mancano gli argomenti scrive lo standard input sullo standard output

wc $[options]\ [file\ \ldots]$

- Word count: fornisce il numero dei codici di interruzione di riga (in pratica il numero delle righe), delle parole o dei caratteri contenuti in $file$.
- Senza opzioni fornisce, nell'ordine suddetto, ciascuna delle precedenti informazioni.
- Alcune opzioni:
    - -c emette solo il numero complessivo di caratteri di $file$
    - -w emette solo il numero complessivo di parole di $file$
    - -l emette solo il numero di righe in $file$

sort [$options$] [$file$ …]

- Permette di (ri)ordinare o fondere insieme il contenuto dei file passati come parametri, oppure di (ri)ordinare le linee passategli in input.
- In assenza di opzioni che definiscano diversi criteri di ordinamento, quest'ultimo avviene in base al primo campo ed è alfabetico.
- Alcune opzioni:
  - -f ignora le differenze tra lettere minuscole e maiuscole
  - -n considera la chiave di ordinamento numerica anziché testuale
  - -r ordina in senso decrescente anziché crescente
  - -o $fileout$ invia l'output a $fileout$ anziché sull'output standard
  - -t s usa s come separatore di campo
  - -k $s1, s2$ usa i campi da $s1$ a $s2 - 1$ come chiavi di ordinamento

head [-$numero$] $file$

- Visualizza le prime 10 (o -$numero$) linee di un file

tail [-$numero$] $file$

- Visualizza le ultime 10 (o -$numero$) linee di un file

grep [$opzioni$] $pattern$ [$nomefile$]

- Stampa le righe del file che corrispondono al $pattern$
- Il $pattern$ è una espressione regolare
- Nel caso più semplice, il pattern può essere una stringa senza caratteri speciali:
  - $grep\ a\ pippo.txt$: stampa le righe di $pippo.txt$ che contengono una a
- Se $nomefile$ non è specificato, legge da standard input
- Questo consente la concatenazione in pipe:
  - ls -l | grep 2006: elenca i file che sono stati modificati l'ultima volta nel 2006 (ma non solo, poiché magari potrebbe esserci un file contenente "2006"; basta che ci sia il pattern)
  - ls -l | grep rwx: elenca i file per cui almeno una categoria di utenti ha tutti i permessi
- Con opzione -v, stampa le righe che non corrispondono al pattern
  - ls -l | grep -v doc: elenca i file che non contengono "doc" nel nome
- Con -c, visualizza solo il numero di occorrenze della stringa nel file;
- -i: case insensitive
- -n: restituisce il numero di linee

ps [$selezione$] [$formato$]

- Selezione:
  - Niente: processi lanciati dalla shell corrente
  - -u $pippo$: i processi dell'utente $pippo$
  - -a: (All) tutti i processi
- Formato:
  - Niente: PID, terminale, ora di esecuzione, comando
  - -f: (full) anche UID, PPID, argomenti
  - -F: (Full) anche altro
  - -o $elenco\_campi$: visualizza i campi specificati