

System call: preliminari di compilazione C, tipi primitivi POSIX, gestione limiti, gestione errori

Roberto De Virgilio

Sistemi operativi - 29 Novembre 2017

Gestione dei file

- **rm**

- ReMove (delete) files

- **cp**

- CoPy files

- **mv**

- MoVe (or rename) file

- **ln**

- LiNk creation (symbolic or not)

- **more, less**

- page through a text file

df [options] [directory]

mostra lo spazio libero nei dischi

% df -Tm

du [options] [directory]

% du

% du directory

% du -s directory

% du -k directory

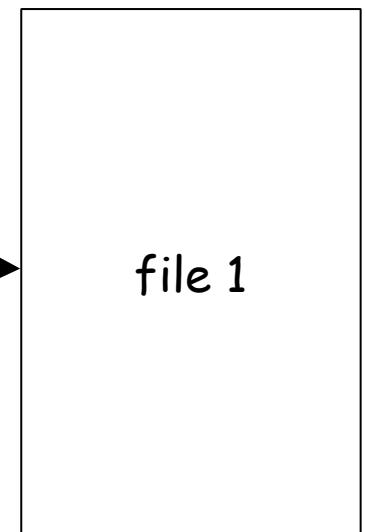
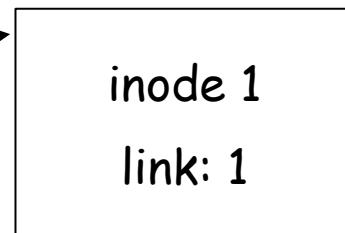
Link

- **`ln nome_file nome_hlink`**
 - E' un hard-link (collegamento fisico)
 - Crea una entry nella directory corrente chiamata *nome_hlink* con lo stesso inode number di *nome_file*
 - Il **link number** dell'inode di *nome_file* viene incrementato di 1
- **`ln -s nome_file nome_slink`**
 - E' un link simbolico (collegamento fittizio)
 - Crea un file speciale nella directory corrente chiamato *nome_slink* che "punta" alla directory entry *nome_file*
 - Il **link number** dell'inode di *nome_file* non viene incrementato
- **Se cancello file:**
 - hard-link: il link number dell'inode viene decrementato
 - link simbolico: il link diviene "stale", ovvero punta ad un file non esistente

Esempio - Situazione iniziale

Directory:

name inode n.

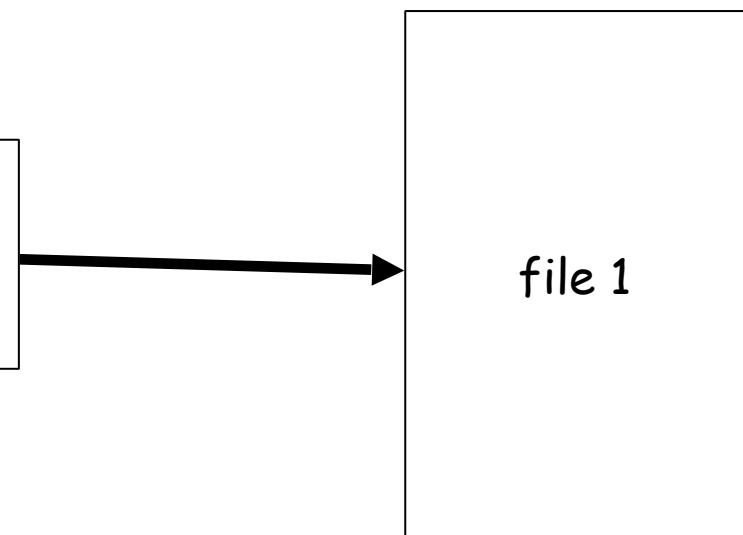
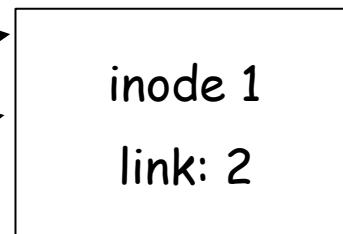


Esempio - Creazione di un hard-link

Directory:

name inode n.

file	inode n.
file	
hlink	

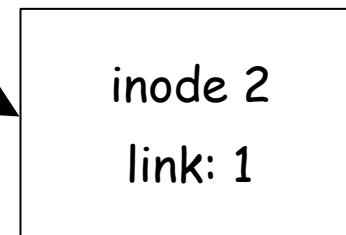
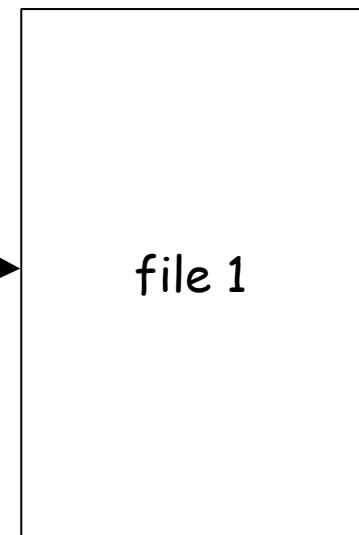
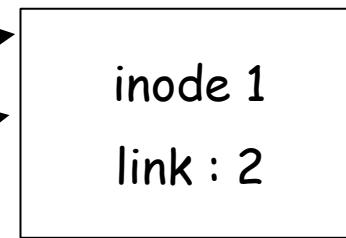


Esempio - Creazione di un link simbolico

Directory:

name inode n.

name	inode n.
file	
hlink	
slink	

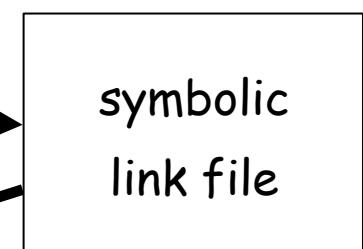
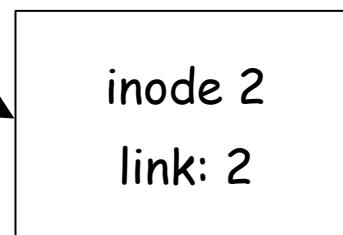
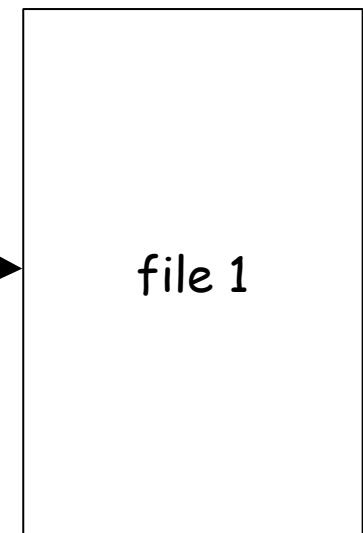
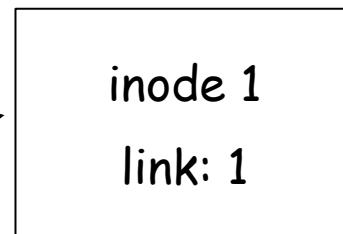


Esempio - Rimozione del file originale

Directory:

name inode n.

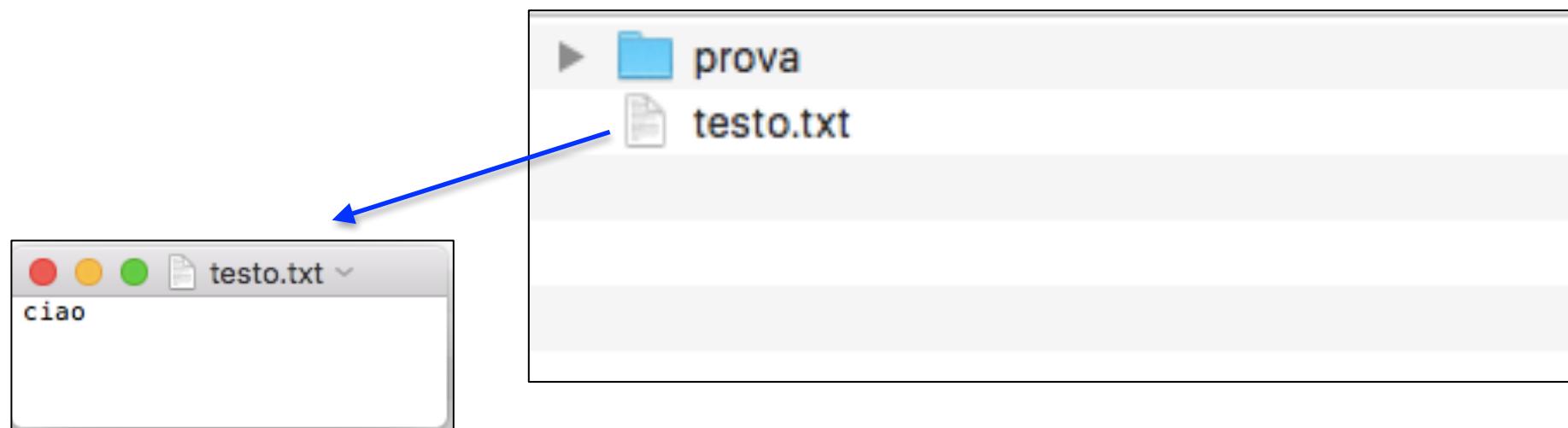
hlink	
slink	



?

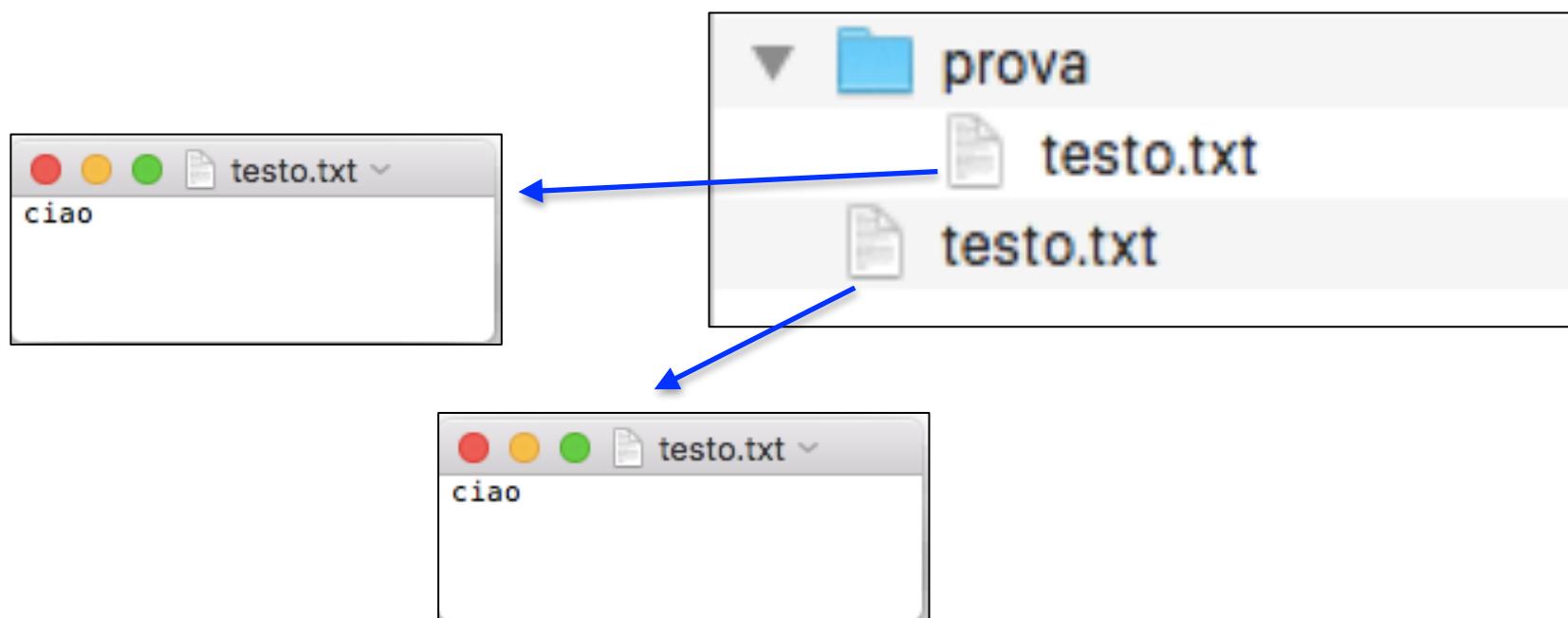
Uso di Hard Link

```
~: mkdir prova  
~: echo "ciao" > testo.txt
```



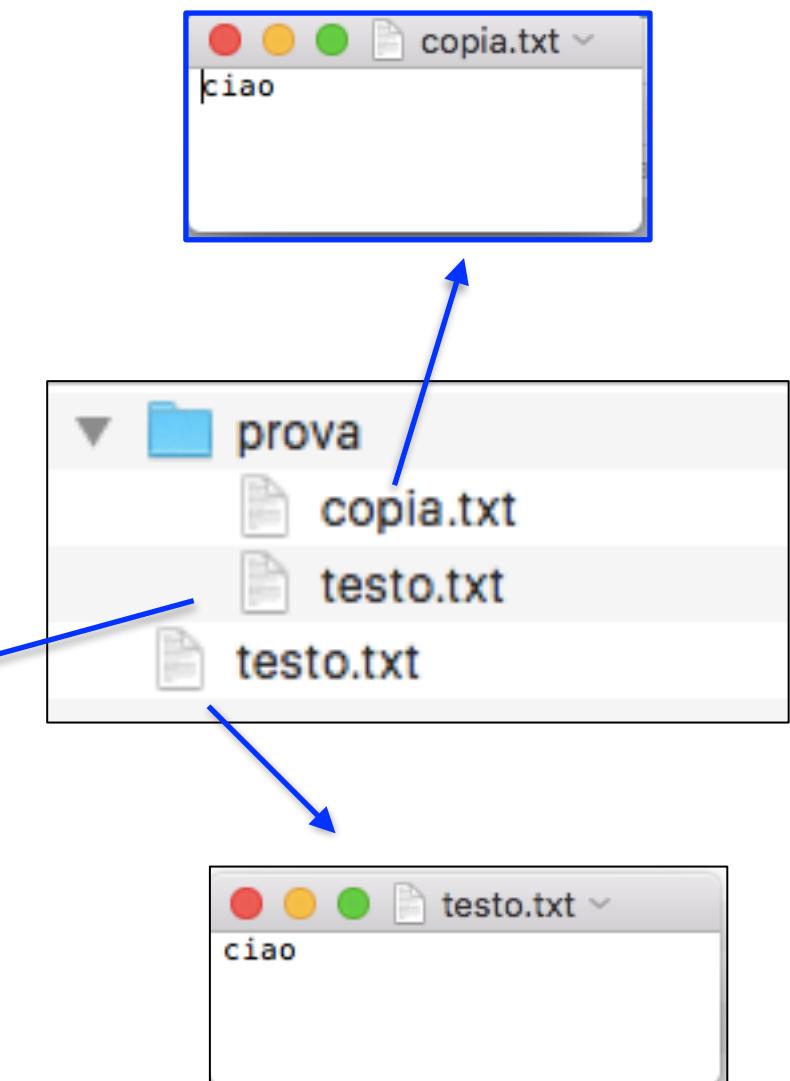
Uso di Hard Link

```
~: mkdir prova  
~: echo "ciao" > testo.txt  
~: cd prova  
~prova: ln ../../testo.txt
```



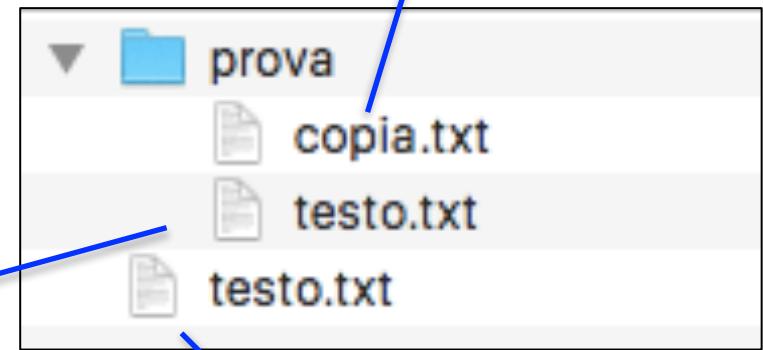
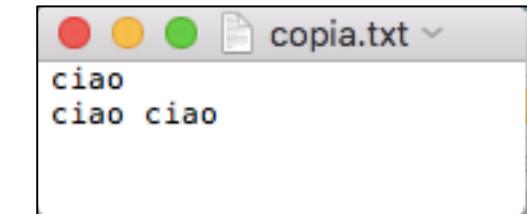
Uso di Hard Link

```
~: mkdir prova  
~: echo "ciao" > testo.txt  
~: cd prova  
~prova: ln ../../testo.txt copia.txt
```



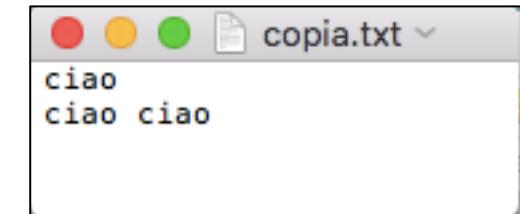
Uso di Hard Link: file dipendenti

```
~prova: echo "ciao ciao" >> testo.txt
```

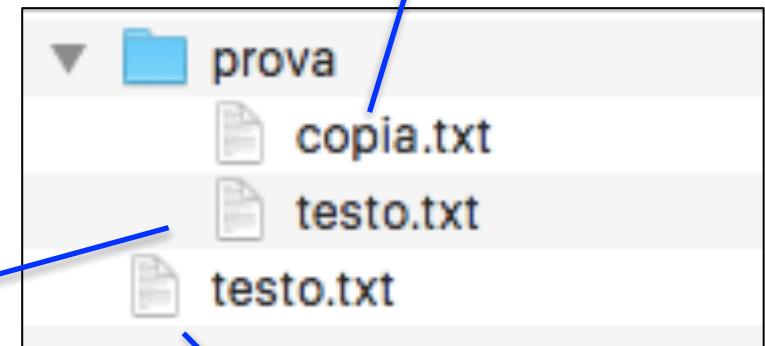


Uso di Hard Link: directory non ammesse

```
~prova: cd ..  
~: ln prova clone  
ln: prova: Is a directory
```



```
copia.txt ~  
ciao  
ciao ciao
```



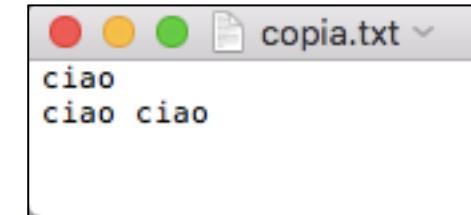
```
testo.txt ~  
ciao  
ciao ciao
```



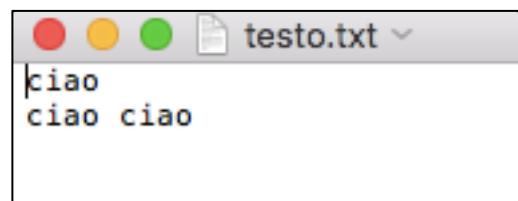
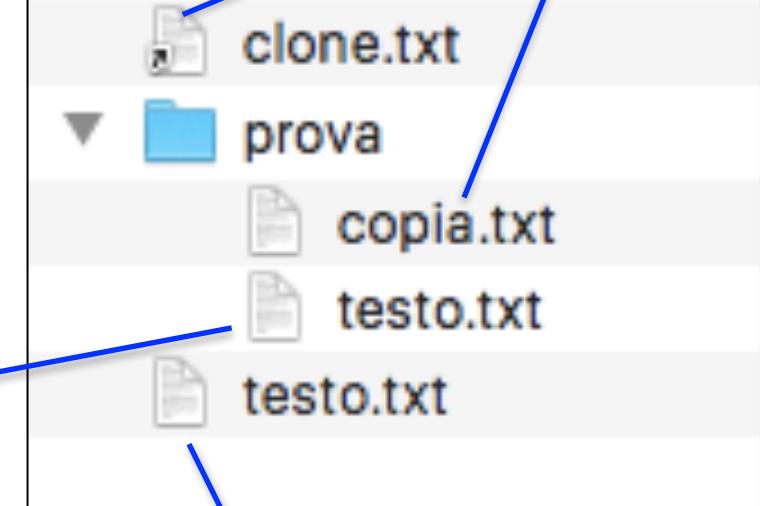
```
testo.txt ~  
ciao  
ciao ciao
```

Uso di Symbolic Link

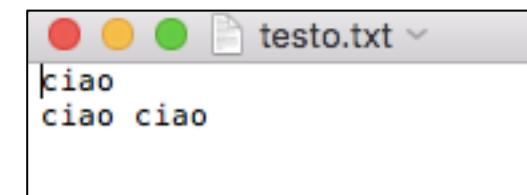
```
~: ln -s prova/copia.txt clone.txt
```



```
ciao  
ciao ciao
```



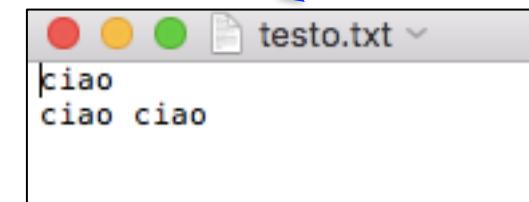
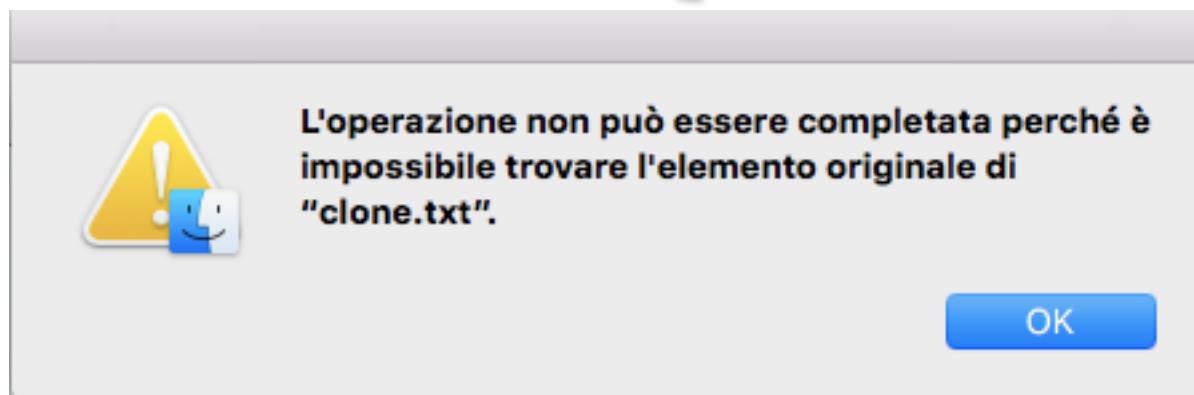
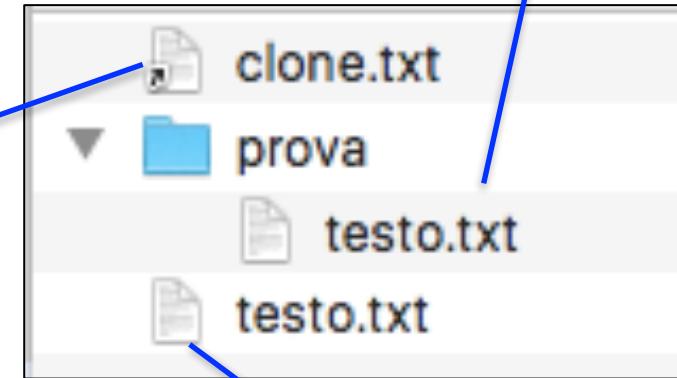
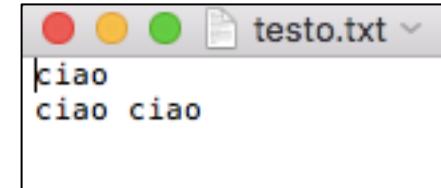
```
ciao  
ciao ciao
```



```
ciao  
ciao ciao
```

Uso di Symbolic Link: no side effect

```
~: ln -s prova/copia.txt clone.txt  
~: cd prova  
~prova: rm copia.txt
```



Uso di Symbolic Link: directory ammesse

```
~: ln -s prova/copia.txt clone.txt
```

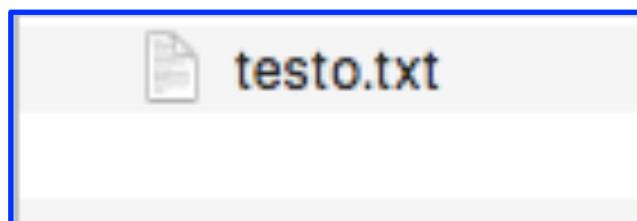
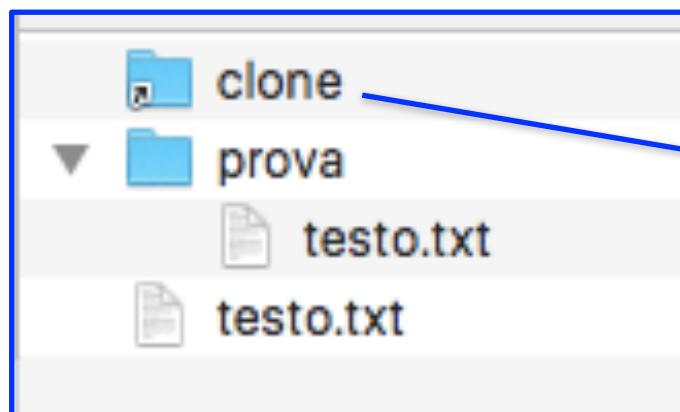
```
~: cd prova
```

```
~prova: rm copia.txt
```

```
~prova: cd ..
```

```
~: rm clone.txt
```

```
~: ln -s prova clone
```



Gestione dei processi

■ Attributi associati ai processi

- **pid**
Identificatore del processo
- **ppid**
Parent pid (identificatore del processo padre)
- **nice number**
Priorità statica del processo; può essere cambiata con il comando **nice**
- **TTY**
Terminal device associata al processo
- **real, effective user id**
real, effective group id
Identificatore del owner e del group owner del processo
- altro:
memoria utilizzata, cpu utilizzata, etc.

Gestione dei processi

■ Comando ps

- Riporta lo stato dei processi attivi nel sistema

```
$ ps
```

```
PID TTY TIME CMD
```

```
648 pts/2 00:00:00 bash
```

```
$ ps alx
```

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
0	0	1	0	15	0	500	244	1207b7	S	?	0:05	init
0	0	2	1	15	0	0	0	124a05	SW	?	0:00	[keventd]
0	0	3	1	34	19	0	0	11d0be	SWN	?	0:00	[ksoftirqd_CPU0]
0	0	4	1	25	0	0	0	135409	SW	?	0:00	[kswapd]
0	0	5	1	25	0	0	0	140f23	SW	?	0:00	[bdflush]
0	0	6	1	15	0	0	0	1207b7	SW	?	0:00	[kupdated]
0	0	7	1	25	0	0	0	15115f	SW	?	0:00	[kinoded]
0	0	9	1	19	0	0	0	23469f	SW	?	0:00	[mdrecoveryd]
0	0	12	1	15	0	0	0	1207b7	SW	?	0:00	[kreiserfsd]
0	0	150	1	0	-20	0	0	107713	SW<	?	0:00	[lvm-mpd]

Gestione dei processi

■ Comando nice

- esegue un comando con una priorità statica diversa

```
nice -n 19 command
```

■ Comando renice

- cambia la priorità di un processo in esecuzione

```
renice [+-]value -p pid
```

■ Comando kill

- termina un processo

```
kill pid
```

```
kill -9 pid
```

Gestione dei processi

■ Processi in foreground:

- Processi che "controllano" il terminale da cui sono stati lanciati
- In ogni istante, un solo processo è in foreground

■ Processi in background

- Vengono eseguiti senza "controllare" il terminale a cui sono "attaccati"

■ Job control

- Permette di portare i processi da background a foreground e viceversa

& Lancia un processo

direttamente in background

Esempio: `long_cmd &`

`^Z` Ferma (stop) il processo in foreground

`jobs`

Lista i processi in background

`%n`

Si riferisce al processo in background n

Esempio: `kill %1`

`fg` Porta un processo in

background in foreground

Esempio: `fg %1`

`bg` Fa ripartire in background i processi fermati

Piccola parentesi: compilare un programma in C

Il programma più noto:

```
#include <stdio.h>
int main ( )
{
    printf ("Hello, World!\n");
    return 0;
}
```

Per compilare il vostro programma

Per compilare da codice sorgente a codice oggetto

```
gcc -c hello.c
```

crea il file `hello.o`

Per fare il linking in un file eseguibile

```
gcc hello.o -o hello
```

crea il file eseguibile `hello`

Per fare i due passi assieme (senza creare un file .o):

```
gcc hello.c -o hello
```

Per eseguire il programma:

```
./hello
```

Hello World!

Messaggi di errore

Inseriamo un errore:

supponiamo di dimenticare il ; dopo il **printf**

```
gcc hello.c -o hello
```

```
"hello.c", line 5: syntax error before or at: return  
gcc: acomp failed for hello.c
```

Nota:

il compilatore rileva l'errore al primo token non appropriato

in questo caso, lo statement **return**

provate sempre a correggere gli errori a partire dal primo; gli altri possono sparire

Variabili esterne

Keyword **extern**

E' possibile dichiarare una variabile globale in qualsiasi parte di un programma, anche in un file separato

In questo caso, bisogna distinguere fra:

dichiarazione "reale" (alloca memoria per la variabile):

la sintassi è la solita

dichiarazione "extern" (rende noto al compilatore l'esistenza della variabile):

la sintassi prevede il qualificatore **extern** per informare il compilatore che ci si riferisce ad una variabile dichiarata in un altro file

Esempio

```
extern int var;
```

Esempio - Variabili esterne

```
/* test_main.c */  
  
int a=4;  
  
extern int b;  
  
int test( );  
  
int main( )  
{  
  
    printf("a=%d,b=%d\n", a,b);  
  
    a = b = 5;  
  
    test();  
  
    return 0;  
}
```

```
/* test.c */  
  
extern int a;  
  
int b=3;  
  
int test( )  
{  
  
    printf("a=%d,b=%d\n", a,b);  
}
```

Cosa succede in compilazione/esecuzione?

Esempio - Variabili esterne

```
/* main.c */                                /* test.c */  
int a=4;                                     extern int a;  
extern int b;                               int b=3;  
int test( );                                 int test( )  
int main( )                                {  
{                                           printf("a=%d,b=%d\n",a,b);  
    printf("a=%d,b=%d\n", a,b); }  
    a = b = 5;  
    test();  
    return 0;  
}
```

Cosa succede in compilazione/esecuzione?

```
gcc main.c test.c -o test
```

```
./test
```

```
a=4, b=3
```

```
a=5, b=5
```

Esempio - Variabili esterne

```
/* main.c */                                /* test.c */  
int a=4;                                    int a=3;  
int b=2;                                    int b=3;  
int test( );                                int test( )  
int main( )                                {  
{                                            printf("a=%d,b=%d\n",a,b);  
    printf("a=%d,b=%d\n", a,b); }  
    a = b = 5;  
    test();  
    return 0;  
}
```

Cosa succede in compilazione/esecuzione?

Esempio - Variabili esterne

```
/* main.c */                                /* test.c */  
int a=4;                                     int a=3;  
int b=2;                                     int b=3;  
int test( );                                 int test( )  
int main( )  
{                                         {  
    printf("a=%d,b=%d\n", a, b);  
    printf("a=%d,b=%d\n", a, b); }  
    a = b = 5;  
    test();  
    return 0;  
}
```

Cosa succede in compilazione/esecuzione?

```
gcc main.c test.c -o test  
/tmp/cc8W0ciu.o(.data+0x0): multiple definition of `a'  
/tmp/ccUvzasl.o(.data+0x0): first defined here  
/tmp/cc8W0ciu.o(.data+0x4): multiple definition of `b'  
/tmp/ccUvzasl.o(.data+0x4): first defined here
```

Variabili automatiche

Le variabili locali sono *automatiche*:

Esistono solo mentre l'esecuzione è nel range del blocco in cui sono definite

Quando il processo entra nel blocco in cui la variabile è definita:

la variabile viene creata

Quando il processo esce dal blocco in cui la variabile è definita:

la variabile viene distrutta

Nel caso il processo esca da un blocco, per poi rientrare, la variabile è da considerare come una nuova istanza

Nota:

Le variabili automatiche vengono allocate nello stack

Esiste una keyword **auto**, che però non è necessaria

Variabili automatiche

```
/* main.c */
#include <stdio.h>
int main()
{
    int x = 1;
    int y = 10;
{
    int x = 2;
    printf("%d, %d\n" ,x,y);
}
printf("%d, %d\n" ,x,y);
}
```

Cosa succede in compilazione/esecuzione?

Variabili automatiche

```
/* main.c */
#include <stdio.h>
int main()
{
    int x = 1;
    int y = 10;

    int x = 2;
    printf("%d, %d\n" ,x,y);
}

printf("%d, %d\n" ,x,y);
```

Cosa succede in compilazione/esecuzione?

gcc main.c -o test

./test

2 10

1 10

Variabili automatiche

```
/* main.c */
#include <stdio.h>
int main()
{
    int x = 1;
    int y = 10;

    {
        int x = 2;
        int z = 20;
        printf("%d, %d\n", x, y);

    }
    printf("%d, %d, %d\n", x, y, z);
}
```

Cosa succede in compilazione/esecuzione?

Variabili automatiche

```
/* main.c */
#include <stdio.h>
int main()
{
    int x = 1;
    int y = 10;
{
    int x = 2;
    int z = 20;
    printf("%d, %d\n",x,y);
}
printf("%d, %d, %d\n",x,y,z);
}
```

Cosa succede in compilazione/esecuzione?

```
gcc main.c -o test
```

```
main.c:12:31: error: use of undeclared identifier 'z'
        printf("%d, %d, %d\n",x,y,z);
                                         ^
1 error generated.
```

Variabili statiche

Keyword static

Una variabile marcata **static** non è **auto**, ovvero non viene distrutta dopo l'uso

Variabili statiche – Globali

Sono visibili solo nel file in cui sono dichiarate

Variabili statiche - Locali

A volte è necessario prolungare la vita di una variabile locale

Per esempio, per contare il numero di invocazioni di una funzione

Variabili statiche - Locali

Variabili statiche – Globali

Sono visibili solo nel file in cui sono dichiarate

Variabili statiche - Locali

A volte è necessario prolungare la vita di una variabile locale

Per esempio, per contare il numero di invocazioni di una funzione

Così non va (x è automatica):

```
int count( ) {  
    int x=0;  
    return (++x);  
}
```

Keyword static

Una variabile marcata **static** non è **auto**, ovvero non viene distrutta dopo l'uso

```
int count( ) {  
    static int x=0;  
    return (++x);  
}  
  
int main() {  
    printf("%d\n", count());  
    printf("%d\n", count());  
    printf("%d\n", count());  
}
```

Esempio – Variabili static locali

```
/* main.c */
int count( ) {
    int x=0;
    return (++x);
}

int main() {
    printf("%d\n", count());
    printf("%d\n", count());
    printf("%d\n", count());
}
```

Cosa succede in compilazione/esecuzione?

Esempio – Variabili static locali

```
/* main.c */
int count( ) {
    int x=0;
    return (++x);
}

int main() {
    printf("%d\n", count());
    printf("%d\n", count());
    printf("%d\n", count());
}
```

Cosa succede in compilazione/esecuzione?

```
gcc main.c -o test
```

```
./test
```

```
1
```

```
1
```

```
1
```

Esempio – Variabili static locali

```
/* main.c */
int count( ){
    static int x=0;
    return (++x);
}

int main() {
    printf("%d\n", count());
    printf("%d\n", count());
    printf("%d\n", count());
}
```

Cosa succede in compilazione/esecuzione?

Esempio – Variabili static locali

```
/* main.c */
int count( ){
    static int x=0;
    return (++x);
}

int main() {
    printf("%d\n", count());
    printf("%d\n", count());
    printf("%d\n", count());
}
```

Cosa succede in compilazione/esecuzione?

gcc main.c -o test

./test

1

2

3

Esempio – Variabili static globali

```
/* main.c */                                /* test.c */  
int a=4;                                     static int a = 2;  
static int b = 1;                           int b=3;  
int test( );                                 int test( )  
int main( )                                {  
{                                           printf("a=%d,b=%d\n",a,b);  
    printf("a=%d,b=%d\n", a,b); }  
    a = b = 5;  
    test();  
    return 0;  
}
```

Cosa succede in compilazione/esecuzione?

Esempio – Variabili static globali

```
/* main.c */                                /* test.c */  
int a=4;                                     static int a = 2;  
static int b = 1;                           int b=3;  
int test( );                                 int test( )  
int main( ) {  
    printf("a=%d,b=%d\n", a,b); }  
    a = b = 5;  
    test();  
    return 0;  
}
```

Cosa succede in compilazione/esecuzione?

```
gcc main.c test.c -o test
```

```
./test
```

```
a=4, b=1
```

```
a=2, b=3
```

Riassunto

Le variabili locali automatiche:

Esistono solo quando l'esecuzione è nel blocco in cui le variabili sono definite; il loro contenuto non persiste dopo che l'esecuzione lascia il blocco

Le variabili locali statiche:

Esistono per l'intera vita di un processo, ma è possibile fare riferimenti ad esse solo quando l'esecuzione è nel blocco in cui sono definite

Le variabili globali

Esistono sempre, sono visibili ovunque (salvo essere nascoste da variabili locali)

Le variabili globali statiche

Sono visibili solo nel file in cui sono dichiarate

Variabili "registro"

La keyword **register**

Suggerisce al compilatore che una variabile verrà usata pesantemente e che sarebbe meglio mantenerla direttamente in un registro del processore per aumentare l'efficienza

Esempio:

```
register int i;  
for (i=0; i < 1000000; i++) ...
```

Note:

E' possibile dichiarare **register** solo variabili automatiche o parametri formali di una funzione

Il numero delle variabili dichiarabili **register** è limitato

Solo certi tipi di dati possono essere mantenuti nei registri

Il compilatore può comunque fare di testa sua...

Librerie ANSI C

Le librerie incluse in ANSI C contengono molte funzioni di utilità:

`<assert.h>`: asserzioni

`<ctype.h>`: test sui caratteri

`<errno.h>`: codici di errore

`<float.h>`: limiti per tipi floating point

`<limits.h>`: limiti

`<locale.h>`: informazioni "locali" (lingua, punteggiatura, etc)

`<math.h>`: funzioni matematiche

`<setjmp.h>`: salti non-locali

`<signal.h>`: segnali

Librerie ANSI C

Elenco (continua...)

`<stdarg.h>`: gestione argomenti

`<stddef.h>`: definizioni di uso generale

`<stdio.h>`: input e output

`<stdlib.h>`: funzioni varie

`<string.h>`: funzioni di stringa

`<time.h>`: funzioni relative a ora e data

Documentazione:

Elenco breve di tutte le funzioni in ANSI C nella pagina di documentazione

Documentazione singole funzioni tramite:

`man 3 nomefunzione`

Esempio: **`man 3 printf`**

Argomenti di linea di comando

E' possibile passare agli argomenti di linea di comando ai programmi eseguibili

Esempio:

ls -l /etc

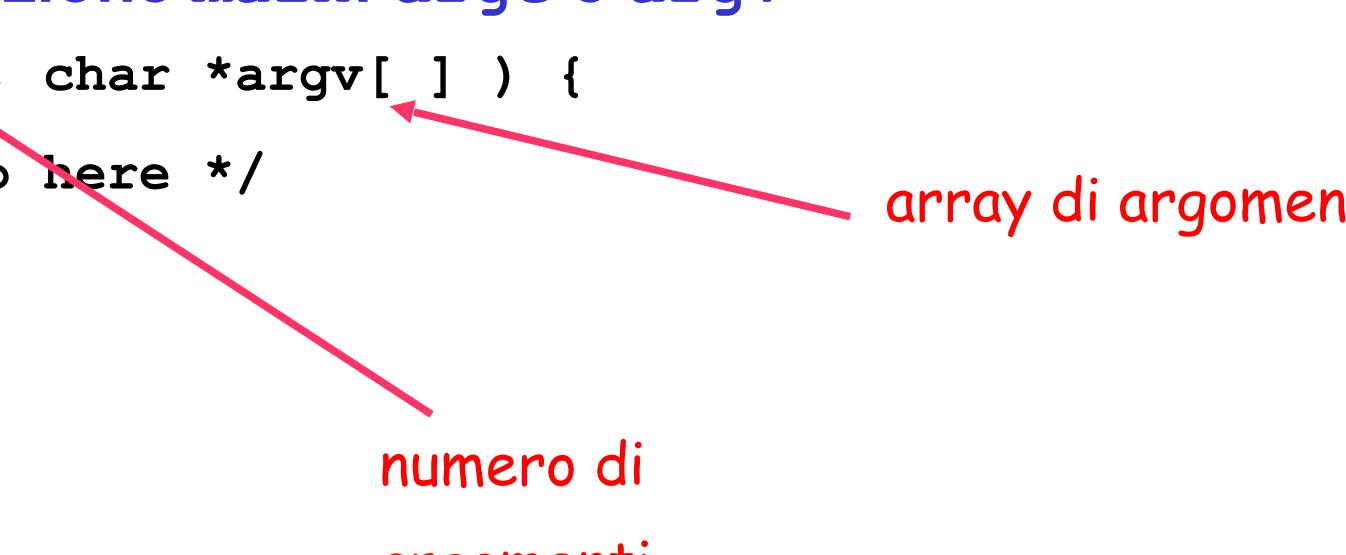
-l e /etc sono argomenti della linea di comando

Parametri della funzione main: argc e argv

```
int main (int argc, char *argv[ ] ) {  
    /* Statements go here */  
}
```

array di argomenti

numero di
argomenti



Esempio 1

```
/* myargs.c */
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf ("Program name: %s\n", argv [0]);
    if (argc > 1) {
        for (i=1; i<argc; i++)
            printf ("Argument %d: %s\n", i, argv[i]);
    }
    else
        printf ("No command line arguments entered.\n");
    return 0;
}
```

Esempio 1 – Esecuzione

Supponiamo di aver compilato il programma precedente come
myargs

```
roberto:~$ ./myargs first "second arg" 3 4
```

Program name: ./myargs

Argument 1: first

Argument 2: second arg

Argument 3: 3

Argument 4: 4

Esempio 2: informazioni di help

```
/* showfile.c */
#include <stdio.h>
int main(int argc, char *argv[ ]) {
    FILE *fp;    int k;
    if(argc !=2) {
        printf("Usage: %s file\n", argv[0]);
        return 0;
    }
    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("Cannot open file!\n");
        return 1;
    }

    while((k=fgetc(fp))!=EOF)    fputc(k, stdout);
    fclose(fp);
    return 0;
}
```

Generalmente, main controlla se gli argomenti sono validi; in caso contrario stampa informazione di help

Esempio 2: informazioni di help

Cosa succede in compilazione/esecuzione?

```
gcc showfile.c -o test  
./test
```

Usage: ./test file

Generalmente, main controlla se gli argomenti sono validi; in caso contrario stampa informazione di help

Esempio 2: informazioni di help

Cosa succede in compilazione/esecuzione?

```
gcc showfile.c -o test  
./test myargs.c
```

```
/* myargs.c */  
#include <stdio.h>  
int main(int argc, char *argv[]) {  
    int i;  
    printf ("Program name: %s\n", argv [0]);  
    if (argc > 1) {  
        for (i=1; i<argc; i++)  
            printf ("Argument %d: %s\n", i, argv[i]);  
    }  
    else  
        printf ("No command line arguments entered.\n");  
    return 0;  
}
```

Generalmente, main controlla se gli argomenti sono validi; in caso contrario stampa informazione di help

Come organizzare un progetto

- **Ovviamente:**
 - Programmi di grandi dimensioni non possono essere contenuti in un file singolo
- **Nel C**
 - vi sono tecniche per semplificare gestione di file multipli
 - non sono "enforced": sono lasciate al programmatore
- **Un programma grande è diviso in moduli:**
 - i file .h contengono le funzioni prototipo del modulo ed eventuali costanti
 - i file .c contengono le definizioni di funzioni del modulo
 - i moduli sono compilati separatamente
 - viene generato un eseguibile tramite linking dei moduli

Premessa

- **Come gestire progetti di programmazione molto grandi**
 - Come organizzarli
 - Come compilarli
 - Come lavorare in cooperazione
- **Nota:**
 - Con un'enfasi sul C, ma i concetti valgono anche per altri linguaggi di programmazione

Esempio: sample.c

```
#include <stdio.h>
#include "my_math.h"
int main()
{
    int a, b, c;
    puts("Input three numbers:");
    scanf("%d %d %d", &a, &b, &c);
    printf("The average of %d %d %d is %f.\n",
           a,b,c,average(a,b,c));
    return 0;
}
```

Esempio - Modulo my_math

```
/* my_math.h */
#define PI 3.1415926
float average(int x,
    int y, int z);
float sum(int x,
    int y, int z);
```

```
/* my_math.c */
#include "my_math.h"
float average(int x, int y,
    int z)
{
    return sum(x,y,z)/3;
}

float sum(int x,
    int y, int z)
{
    return x+y+z;
}
```

Compilare questo semplice programma

- Per generare `my_math.o`
 - Abbiamo bisogno di `my_math.c` e `my_math.h`
 - `gcc -c my_math.c`
- Per generare `sample.o`
 - Abbiamo bisogno di `sample.c` e `my_math.h`
 - `gcc -c sample.c`
- Per generare l'eseguibile
 - Abbiamo bisogno di `my_math.o` e `sample.o`
 - `gcc -o sample sample.o my_math.o`

Come utilizzare make

- I programmi che consistono di molti moduli sono impossibili da mantenere manualmente
- Si crea un Makefile

Target

Indentazioni
con tab, non
spazi

```
# Makefile for the sample
sample: sample.o my_math.o
    cc -o sample sample.o my_math.o
sample.o: sample.c my_math.h
    cc -c sample.c
my_math.o: my_math.c my_math.h
    cc -c my_math.c
clean:
    rm sample *.o core
```

Dipendenze

Comandi

Makefile

- Un makefile consiste in un insieme di regole del tipo:

```
target ... : prerequisites ...
            command
            ...
            ...
```

- Target:

- il nome di un file da generare
- un'azione da svolgere

Prerequisiti:

uno o più file utilizzati come input per creare il file target

Regole

una o più azioni da eseguire

Nota:

la prima regola è quella di "default", o radice

Come utilizzare make

■ Come?

- Si salva il file con il nome **Makefile** o **makefile** nella stessa directory contenente i file
- Si lancia l'utility **make**

■ Make

- trova il **Makefile**
- verifica le regole e le dipendenze e rigenera i file per cui è necessario un update

■ Esempio:

- Se solo **sample.c** è stato modificato:
 - **cc -c sample.c**
 - **cc -o sample sample.o my_math.o**

Come utilizzare make

- Per rimuovere tutti i file generati
 - `make clean`
- Per ricompilare, è possibile:
 - Rimuovere tutti i file generati e ricompilare
 - `make clean ; make`
 - Oppure è possibile cambiare la data di ultima modifica:
 - `touch my_math.h ; make`
 - La data di ultima modifica di `my_math.h` prende l'ora corrente
 - Nota: questo perchè tutti i file dipendono da `my_math.h`

Introduzione

■ System call

- per utilizzare i servizi del s.o., il programmatore ha a disposizione una serie di “entry point” per il kernel chiamate system calls
 - in UNIX, circa 70-200, a seconda della versione
- nei sistemi UNIX, ogni system call corrisponde ad un funzione di libreria C
 - il programmatore chiama la funzione, utilizzando la sequenza di chiamata standard di C
 - la funzione invoca il servizio del sistema operativo nel modo più opportuno

Introduzione

■ Funzioni di libreria:

- funzioni di utilità che forniscono servizi general purpose al programmatore
- queste funzioni non sono entry point del kernel, sebbene alcune di esse possano fare uso delle system call per realizzare i propri servizi
- esempi:
 - La funzione **printf** può invocare la system call **write** per stampare
 - La funzione **strcpy** (string copy) e la funzione **atoi** (convert ASCII to integer) non coinvolgono il sistema operativo

Introduzione

■ Distinzione:

- Dal punto di vista del programmatore, non vi sono grosse distinzioni fra le funzioni di libreria e le system call
 - Entrambe sono funzioni
- Dal punto di vista di chi implementa un sistema operativo, la differenza è ovviamente più importante

■ Nota:

- E' possibile sostituire le funzioni di libreria con altre funzioni che realizzino lo stesso compito, magari in modo diverso
- In generale, non è possibile sostituire le system call, che dipendono dal sistema operativo

Introduzione

- **Le funzioni malloc, free**

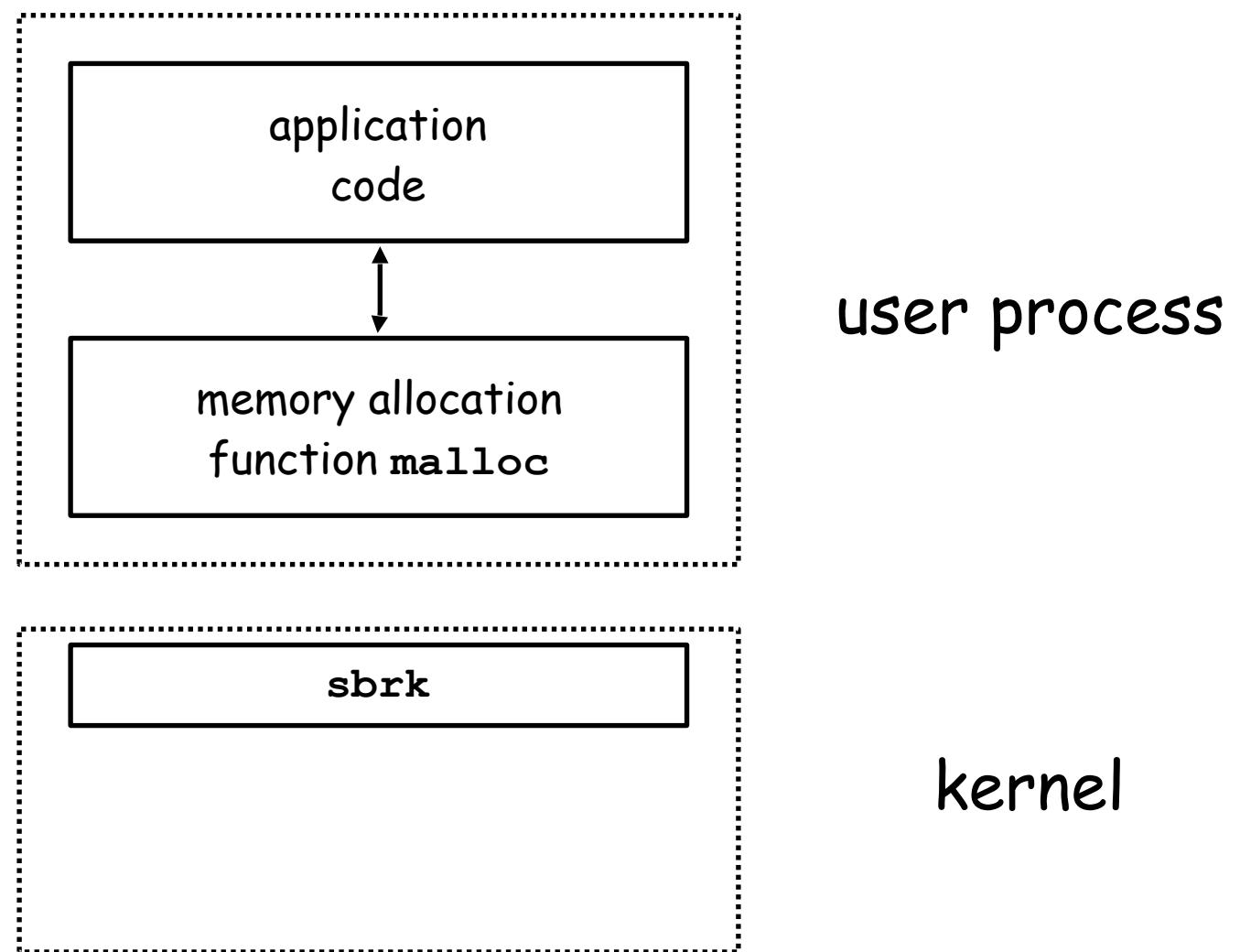
- gestiscono l'allocazione della memoria e le procedure associate di garbage collection
- esistono diversi modi per implementare la `malloc` e la `free` (best fit, first fit, worst fit, etc)
- è possibile sostituire la `malloc` con il nostro meccanismo di allocazione

- **La system call sbrk:**

- non è una funzione di allocazione della memoria general-purpose;
- il suo unico compito è di aumentare o diminuire lo spazio di memoria associato ad un processo

Introduzione

- La funzione `malloc` (e qualunque funzione alternativa per la gestione della memoria) è basata sulla `sbrk`:



Introduzione - Standard

▪ **POSIX (Portable Operating System Interface)**

- E' una famiglia di standard sviluppato dall'IEEE
 - **IEEE 1003.1 (POSIX.1)**
Definizione delle system call (operating system interface)
 - **IEEE 1003.2 (POSIX.2)**
Shell e utility
 - **IEEE 1003.7 (POSIX.7)**
System administration
- Noi siamo interessati allo standard IEEE 1003.1 (POSIX.1)
 - **POSIX.1** 1988 Original standard
 - **POSIX.1** 1990 Revised text
 - **POSIX.1a** 1993 Addendum
 - **POSIX.1b** 1993 Real-time extensions
 - **POSIX.1c** 1996 Thread extensions

Introduzione - Standard

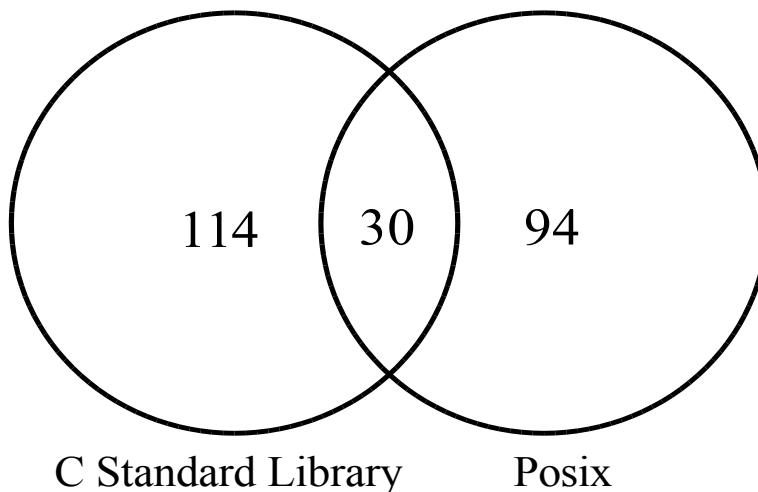
■ POSIX.1

- **cpio.h** valori di archivio x cpio
- **dirent.h** gestione directory
- **fcntl.h** controllo dei file
- **grp.h** gestione del file group
- **pwd.h** gestione del file password
- **tar.h** valori di archivio x tar
- **termios.h** I/O terminale
- **unistd.h** costanti simboliche
- **utime.h** informazioni sul tempo / file
- **sys/stat.h** informazioni sui file
- **sys/times.h** informazioni sul tempo / processi
- **sys/types.h** tipi di dato di sistema
- **sys/wait.h** controllo processi

Introduzione - Standard

- **POSIX.1 include anche alcune primitive della C Standard Library**

Numero di funzioni:



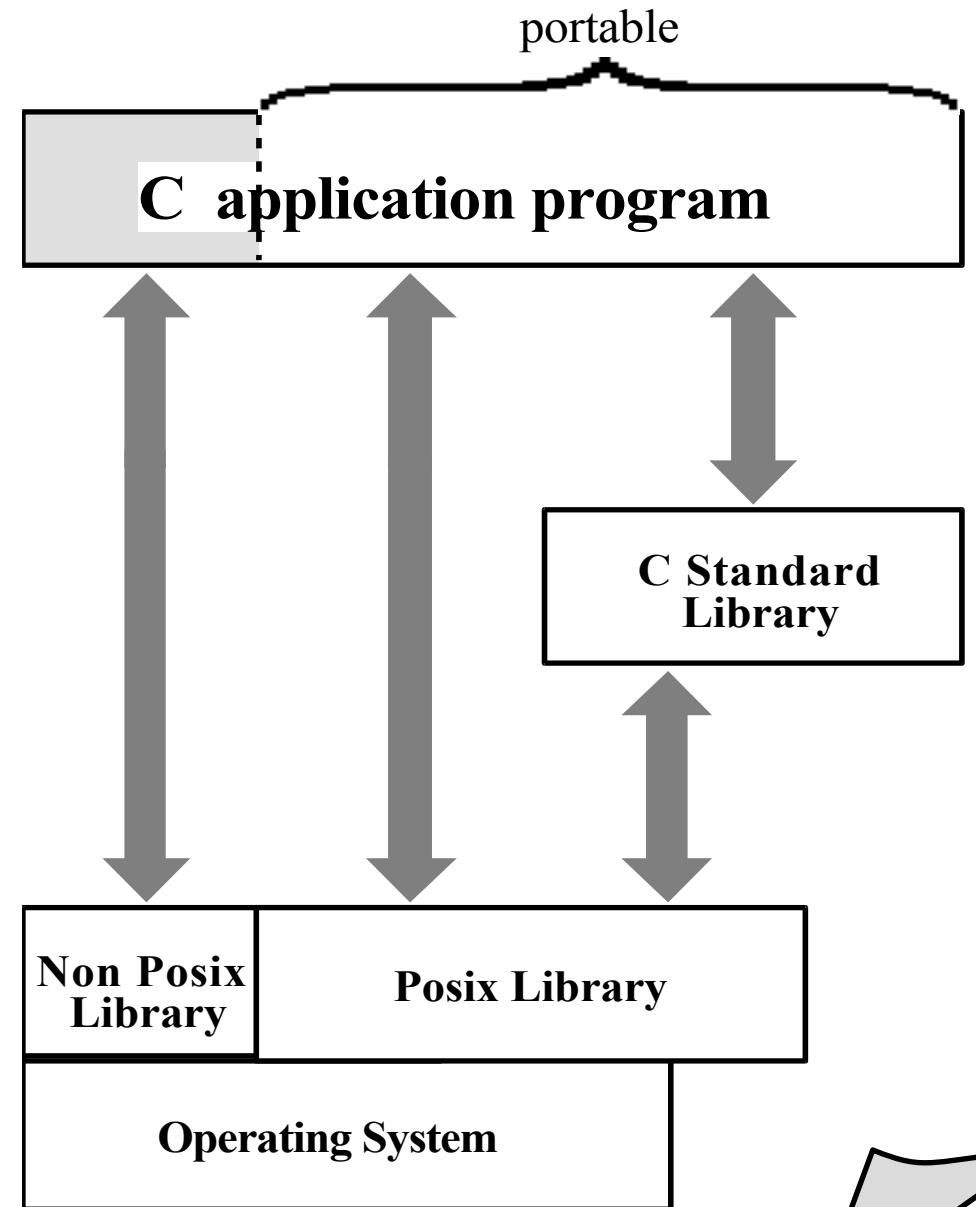
- **Esempio:**

- `open()` è POSIX
- `fopen()` è Standard C e POSIX
- `sin()` è Standard C

Introduzione - Standard

- **Portabilità:**

Un sorgente C Standard conforme a POSIX può essere eseguito, una volta ricompilato, su qualunque sistema POSIX dotato di un ambiente di programmazione C Standard



Introduzione - Standard

- Per scrivere un programma conforme a POSIX, bisogna includere gli header files richiesti dalle varie primitive usate, scelti fra:
 - header files della C Standard Library (contenenti opportune modifiche POSIX)
 - header files specifici POSIX
- Nel seguito, gli header files saranno omessi dalle slide;
 - per ottenere la lista completa degli header necessari, utilizzate `man`
- Inoltre:
 - Se vogliamo compilare un programma in modo tale che dipenda solo dallo standard POSIX, dobbiamo inserire prima degli `#include`:

```
#define _POSIX_SOURCE 1
```

Introduzione – Tipi Primitivi

▪ Tipi di dato

- Storicamente, le variabili UNIX sono state associate a certi tipi di dato C
 - ad esempio: nel passato, i major/minor device number sono stati collocati in un variabile a 16 bit (8 major, 8 minor)
 - sistemi più grandi possono richiedere una dimensione maggiore (e.g. SVR4 usa 32 bit, 14 major, 18 minor)
- Lo header `sys/types.h` definisce i tipi dei dati di sistema primitivi (dipendenti dall'implementazione):
 - ad esempio, in linux `sys/types.h` definisce il tipo `dev_t` come una quantità a 16 bit
 - questi tipi sono definiti per garantire portabilità
 - sono definiti tramite `typedef`

Introduzione – Tipi Primitivi

- **Alcuni dei primitive system data types**

- `clock_t` counter of clock ticks
- `dev_t` device numbers
- `fd_set` file descriptor set
- `fpos_t` file position
- `gid_t` group id
- `ino_t` inode number
- `mode_t` file types, file creation mode
- `nlink_t` number of hard links
- `off_t` offset
- `pid_t` process id
- `size_t` dimensioni (unsigned)
- `ssize_t` count of bytes (signed) (read, write)
- `time_t` counter of seconds since the Epoch
- `uid_t` user id

Introduzione - Limiti

■ Limiti delle implementazioni UNIX

- esistono un gran numero di valori costanti e magic number che dipendono dall'implementazione di UNIX
- nel passato, questi valori erano “hard-coded” nei programmi (i valori venivano inseriti direttamente)
- per garantire una maggior portabilità, i vari standard hanno definito dei metodi più portabili per accedere alle costanti specifiche di un'implementazione

Introduzione - Limiti

- **Si considerano tre classi di valori costanti:**
 - opzioni compile-time (il sistema supporta job control?)
 - limiti compile-time (quant'è il valore massimo per short?)
 - limiti run-time (quanti caratteri in un nome di file?)
- **Note:**
 - le prime due classi sono determinate a tempo di compilazione, e quindi i loro valori possono essere definiti come costanti negli header di POSIX/ANSI C
 - la terza classe richiede che il valore sia ottenuto chiamando un'opportuna funzione

Introduzione - Limiti

■ Limiti ANSI-C

- tutti i limiti ANSI C sono limiti compile-time
- sono definiti nello header `limits.h`
`(/usr/include/limits.h)`
- contiene informazioni sui tipi primitivi, come ad esempio
 - numero di bit in un `char`
 - valore massimi/minimi di `char`, `signed char`, `unsigned char`,
`int`, `unsigned int`, `short`, `unsigned short`, `long`,
`unsigned long`

Introduzione - Limiti (estratto di limits.h)

```
/* Number of bits in a `char'. */
#define CHAR_BIT      8

/* Minimum and maximum values a `signed char' can hold. */
#define SCHAR_MIN     (-128)
#define SCHAR_MAX      127

/* Maximum value an `unsigned char' can hold. (Minimum is 0.) */
#define UCHAR_MAX     255

/* Minimum and maximum values a `char' can hold. */
#ifndef __CHAR_UNSIGNED__
#define CHAR_MIN      0
#define CHAR_MAX      UCHAR_MAX
#else
#define CHAR_MIN      SCHAR_MIN
#define CHAR_MAX      SCHAR_MAX
#endif

/* Minimum and maximum values a `signed short int' can hold. */
#define SHRT_MIN      (-32768)
#define SHRT_MAX      32767
```

Introduzione - Limiti

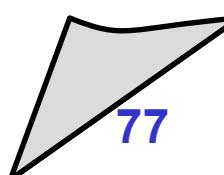
■ **Limiti POSIX.1**

- POSIX.1 definisce un gran numero di costanti che definiscono limiti implementativi del sistema operativo
- questo è uno degli aspetti più complessi di POSIX.1

■ **Limiti runtime**

- alcuni limiti runtime possono essere fissi in un'implementazione
- altri possono variare all'interno della stessa implementazione
- esempio:
 - la dimensione massima di nome di file dipende dal particolare file system considerato

Introduzione - Limiti



Introduzione - Limiti

- **33 limiti e costanti, suddivisi in 7 categorie:**
 - *Limiti invarianti run-time* (eventualmente indeterminati):
 - **ARG_MAX** max. number of arguments for exec
 - **CHILD_MAX** max. number of processes per user ID
 - **CLK_TCK** number of clocks ticks per second
 - **OPEN_MAX** max number of open files per process
 - **PASS_MAX** max. number of significant char. in password
 - **STREAM_MAX** max. number of standard I/O streams
 - **TZNAME_MAX** max. number of byte for names of a time zone
 - *Costanti simboliche a tempo di compilazione:*
 - **_POSIX_SAVED_IDS** if implementations supports saved ids
 - **_POSIX_VERSION** posix version
 - **_POSIX_JOB_CONTROL** if implementations supports job control

Introduzione - Limiti

- **33 limiti e costanti, suddivisi in 7 categorie:**
 - **limiti variabili per i pathname** (eventualmente indeterm.):
 - **LINK_MAX** max value of link's count
 - **MAX_CANON** terminal-related
 - **MAX_INPUT** terminal-related
 - **NAME_MAX** max number of chars for a filename (no-null)
 - **PATH_MAX** max number of chars for a pathname (no-null)
 - **PIPE_BUF** max. number of bytes atomic. written in a pipe
 - **flag variabili:**
 - **_POSIX_NO_TRUNC** if filenames longer than NAME_MAX are truncated
 - **_POSIX_CHOWN_RESTRICTED** if use of chown is restricted

Introduzione - Limiti

- **Funzioni di lettura limiti:**
 - `long sysconf(int name);`
 - `long pathconf(const char *path, int name);`
 - `long fpathconf(int filedes, int name)`
- **La prima legge i valori invarianti**
 - esempio di nome: `_SC_nome_costante`
- **La seconda e la terza leggono i valori che possono variare a seconda del file a cui sono applicati**
 - esempio di nome: `_PC_nome_costante`
- **Vedi codice contenuto in `conf.c`**

Introduzione - Limiti

- **Limiti indeterminati**

- se non è definito nello header `limits.h`...
- se `sysconfig()` o `pathconfig()` ritornano `-1`...
- ...allora il valore non è definito

- **Esempio:**

- molti programmi hanno necessità di allocare spazio per i pathname; la domanda è: quanto?
- `pathalloc.c` cerca di determinare `PATH_MAX`
- in caso il valore “di default” non sia sufficiente, in alcuni casi è possibile aumentare lo spazio previsto e ritentare
- esempio: `getcwd()` (get current working directory)

Introduzione - Limiti

■ Esempio

- una sequenza di codice comune in un processo daemon (che esegue in background, non connesso ad un terminale) è quella di chiudere tutti i file aperti
- come?

```
#include <sys/param.h>  
  
for (i=0; i < NOFILE; i++) close(i);
```

Introduzione – Gestione Errore

- **La maggior parte delle system call:**

- restituisce il valore -1 in caso di errore ed assegna lo specifico codice di errore alla variabile globale

```
extern int errno;
```

- **Nota:**

- se la system call ha successo, **errno** non viene resettato

- **Lo header file `errno.h` contiene la definizione dei nomi simbolici dei codici di errore**

```
# define EPERM 1 /* Not owner */  
# define ENOENT 2 /* No such file or dir */  
# define ESRCH 3 /* No such process */  
# define EINTR 4 /* Interr. system call */  
# define EIO 5 /* I/O error */
```

Introduzione – Gestione Errore

- La primitiva `void perror (const char *str)`
 - converte il codice in `errno` in un messaggio in inglese, e lo stampa anteponendogli il messaggio di errore `str`
- Esempio:

...

```
fd=open ("nonexist.txt", O_RDONLY);  
if (fd== -1) perror ("main");  
...  
--> main: No such file or directory
```



Current conditions at Pescara, Italy (LIP) 42.26N 014.12E 11M (LIP)
Last updated Feb 10, 2012 - 02:50 PM EST / 2012-02-10 1950 UTC
Temperature: 1 C
Relative Humidity: 80%
Wind: from the W (270 degrees) at 15 MPH (13 KT) gusting to 45 KPH
Weather: light snow grains
Sky conditions: overcast

Su	Mo	Tu	We	Th	Fr	Sa
feb 29	30	31	01	02	03	04
05	06	07	08	09	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
mar 26	27	28	29	01	02	03

Su	Mo	Tu	We	Th	Fr	Sa
mar 04	05	06	07	08	09	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

silvio@Stain ~ \$ cd Video
silvio@Stain:~/Video\$ movgrab http://vimeo.com/27998081

Formats available for this Movie: flv
Selected format: flv
Progress: 61.47% 15.4M of 25.1M 693.6K/s

