CARATTERISTICHE DEL C COME LINGUAGGIO DI SISTEMA

- visibilità della **rappresentazione** delle variabili

  operatore **sizeof**
  per ogni variabile o tipo fornisce la dimensione

- possibilità di agire sugli **indirizzi**

  operatore **&**
  per ogni variabile o funzione fornisce l'indirizzo dell'area di memoria

  puntatori come indirizzi → **aritmetica** sugli indirizzi

- operatori a **basso livello**

  operatori **bit a bit**

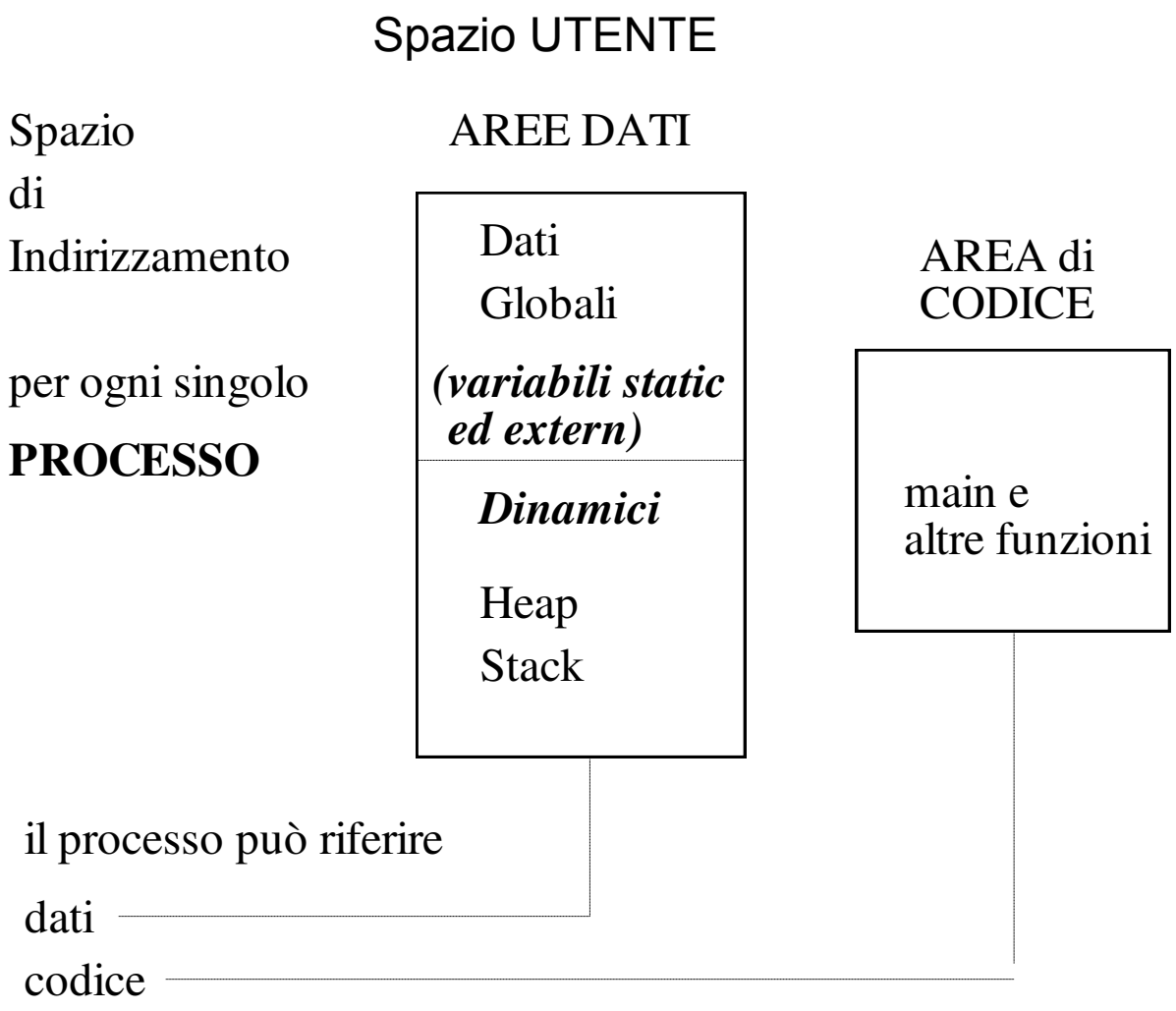
Per le altre funzioni ci si appoggia al
sistema operativo **UNIX** *oppure*
al supporto a tempo di esecuzione

Tipiche funzioni:
⇨ I/O da standard input/output
(*getchar, gets, scanf, putchar, puts, printf*)
⇨ interazione con il S.O. (*exit*)
etc.

Inoltre:
accesso al FILE system
gestione processi concorrenti
e loro comunicazione/sincronizzazione
(**solo UNIX**)
...

SPAZIO DI INDIRIZZAMENTO DI UN PROCESSO

(*punto di vista interno* ➔ lo approfondiremo quando parleremo di processi)



Un processo può riferire dati aggiuntivi in **AREA di KERNEL**

argc
argv
envp char ** envp ➔ stringhe composte **nome = valore**
tabella file aperti ➔ interazione con i file
etc.

L'area di KERNEL è generalmente NON visibile direttamente, ma solo tramite **primitive** ➔ **accessibile direttamente solo argc e argv**

PRIMITIVE

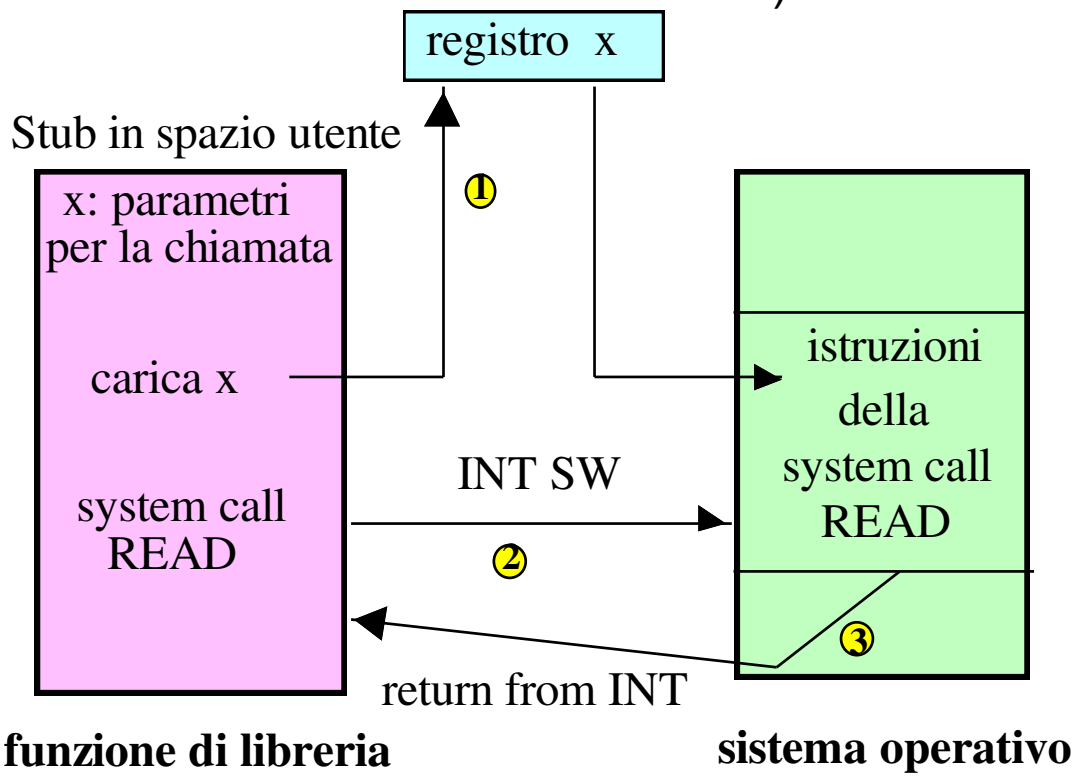
si dicono **primitive** le azioni elementari della macchina virtuale UNIX con **proprietà**

- operazioni di **base** (con cui formare tutte le altre)
- operazioni **atomiche** (eseguite senza interruzione)
- operazioni **protette** (eseguite in ambiente di kernel)

*Le **primitive** sono visibili come **normali procedure/funzioni**,*
ad esempio, invocabili da C
*Ma sono chiamate eseguite dal **sistema operativo***

Implementativamente

Una chiamata ad una primitiva
richiede il **cambiamento di visibilità** (allo stato kernel)
e il **trasferimento** dei parametri
viene espansa con un intermediario (**stub**)
che tratta i **parametri** e chiede il passaggio al kernel
con un trasferimento di controllo dinamico detto **TRAP**
(operazione di **basso livello**: Assembler)



FILE IN C

I file in C sono visti in modo molto semplice tramite un insieme di operazioni primitive (invocabili, come detto, come semplici funzioni di **libreria**)

CARATTERISTICHE DI BASE:
Organizzazione a **BYTE**
ACCESSO **sequenziale**
I/O pointer associato all'apertura del file

Per interagire con i file, il Sistema Operativo mette a disposizione un **TIPO DI DATO ASTRATTO**: il S.O. (il File System) agisce come ***gestore*** delle istanze di questo tipo

L'interazione con un file deve quindi essere autorizzata dal gestore e quindi dal File System ➔ ***operazioni di richiesta risorsa (prologo) e rilascio (epilogo)***

L'operazione di **prologo** consente di ottenere una **chiave** (**file descriptor**, *UNIX* o **file handle**, *MS-DOS*) che serve per poter richiedere le operazioni di lettura, scrittura, posizionamento del file pointer

L'operazione di epilogo consente di restituire la chiave quando la sessione di interazione con il file è terminata

OPERAZIONI di Basso LIVELLO sui FILE

create*, *open*, *close*, *read* / *write*, *lseek

OPERAZIONI sul FILE SYSTEM: prologo, epilogo

===> operazioni di RICHIESTA e RILASCIO risorse

CREATE

```
fd = creat(name, mode);
        char *name;   /* nome del file */
        int mode;     /* attributi del file */
        int fd;       /* chiave */
```

mode ⇨ in UNIX sono i diritti del file (di solito in ottale)

La primitiva **creat** crea un file di nome *name* e diritti *mode*

===> se il file esiste già lo azzera (mode non ha effetto)
NOTA: si deve avere il diritto di scrittura

===> il file viene APERTO in scrittura

OPEN

```
fd = open(name, flag);
    int flag; /* modalità di apertura */
```

flag ⇨ in UNIX `#include <fcntl.h>`
sono definite le costanti
`O_RDONLY`, `O_WRONLY`, `O_RDWR`
(sola lettura, sola scrittura, entrambe)

La primitiva **open** apre il file di nome *name* con modalità *flag*

Sia la **creat** che la **open** ritornano un FILE DESCRIPTOR se hanno successo; altrimenti ritornano -1
⇨ controllare sempre

CLOSE

```
retval = close(fd);
   int fd, retval;
```

La primitiva **close** chiude il file corrispondente a fd

===> chiusura automatica dei file alla terminazione del processo

FILE DESCRIPTOR

===> riferimento a *istanze* del **tipo di dato astratto** che consentono ad un processo di accedere ai file

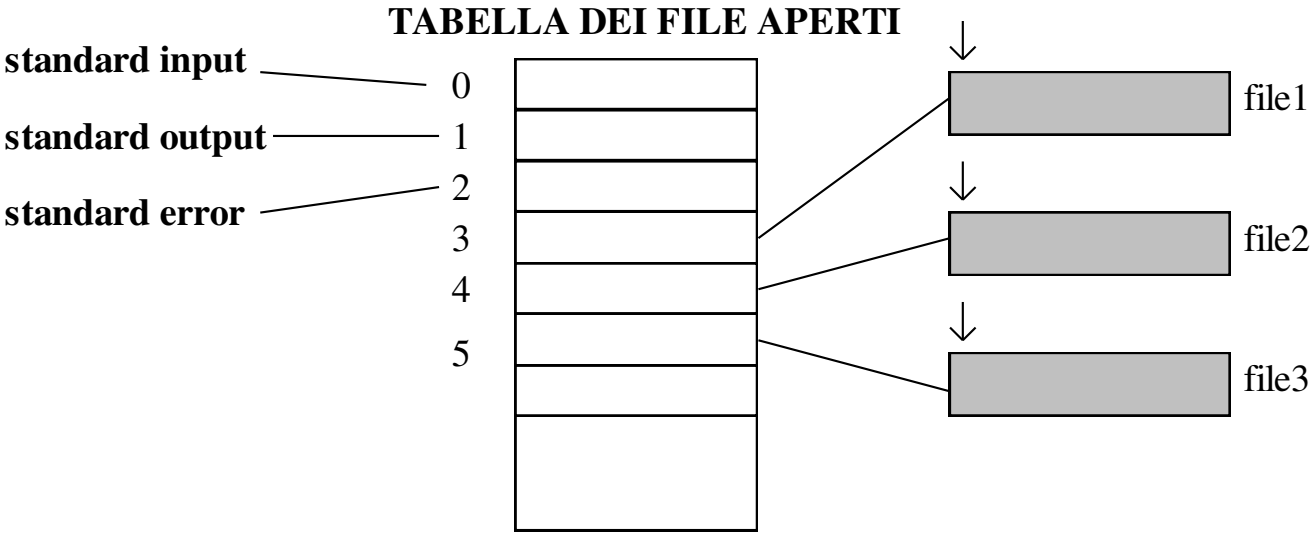
===> corrispondono ad un **INDICE** nella **TABELLA dei FILE APERTI** (parte dell'area di kernel associata ad ogni processo)

Avere delle tabelle implica: a) LIMITE al numero massimo di file aperti per ogni processo e b) LIMITE sul numero massimo di file aperti in tutto il sistema

I concetti di
- standard input
- standard output
- standard error
sono associati ai file descriptor **0**, **1**, **2** (N.B. ragione in particolare per notazione 2> per ridirigere lo standard error!)

Nuove operazioni di RICHIESTA di una risorsa file producono nuovi file descriptor per un processo

Nell'area di Kernel di ogni processo:



In realtà lo schema è un po' più complicato ===> vedi dopo

Chiave ⇨ indice (**NUMERI** successivi) tabella dei file aperti

NOTE SU CLOSE E OPEN/CREAT

1) Dimensione della tabella dei file aperti ➔
   fissata dal Sistema Operativo
   la funzione close() risulta necessaria per liberare degli elementi quando si tenta di aprire un file avendo esaurito lo spazio nella tabella dei file aperti

2) Ogni volta che viene aperto o creato un file, il Sistema Operativo cerca nella tabella dei file aperti il **primo elemento libero**:
   quindi, se le aperture/creazioni e le chiusure vengono mescolate, l'effetto è che verranno riutilizzati elementi precedentemente occupati e perciò verranno ritornati file descriptor già utilizzati

Questo consente la realizzazione del meccanismo di ridirezione

Completa Omogeneità dei file con i dispositivi

Anche per i dispositivi usiamo le stesse primitive
open, read, write, close

OPEN (approfondimento)

```
OPEN    fd = open(name, flag, mode);
          char *name;
          int flag;
          int mode;     /* attributi del file */
          int fd;       /* file descriptor  */
```

Nel file header **fcntl.h** altre costanti oltre a
O_RDONLY, O_WRONLY, O_RDWR

- ⇨ **O_APPEND** aggiunge in fondo al file,
  ➔ **N.B.** Bisogna usarlo insieme con `O_WRONLY, O_RDWR`
- ⇨ **O_TRUNC** distrugge il contenuto, se il file esiste
- ⇨ **O_CREAT** crea il file, se non esiste
  ➔ **N.B.** Bisogna specificare il terzo parametro **mode**
- ⇨ **O_EXCL** fallisce se il file esiste già
  ➔ **N.B.** Bisogna usarlo insieme con `O_CREAT`

ESEMPI:

```
#include <fcntl.h>

fd1 = open("pippo", O_CREAT, 0644);
fd2 = open("paperino", O_CREAT | O_EXCL, 0644);
fd3 = open("paperina", O_TRUNC);
fd4 = open("pluto", O_WRONLY | O_APPEND);
```

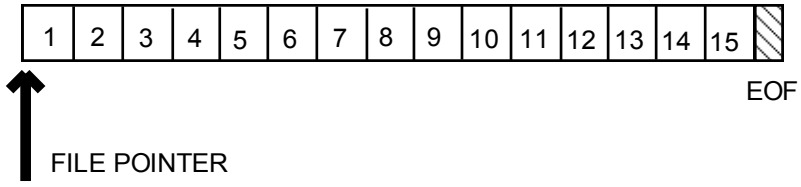
Se fd2 negativo, cosa vuol dire?

FILE POINTER

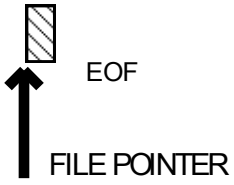
Dopo che un file è stato aperto (o creato), tramite il file descriptor appositamente ottenuto è possibile operare sul file in modo **sequenziale**: la posizione corrente all'interno del file consente, in ogni istante, di sapere su quale parte del file avrà effetto la prossima operazione richiesta

Tale posizione viene indicata dal **file pointer** (detto anche I/O pointer) associato al file descriptor utilizzato

NOTA BENE: Il file pointer non è associato al file effettivo, ma al file descriptor e quindi fa parte dell'istanza del tipo di dato astratto *file*: quindi, aperture effettuate sullo stesso file, definiranno file pointer separati



(a) Situazione del file pointer dopo l'apertura del file.



(b) Situazione del file pointer dopo la creazione del file.

OSSERVAZIONE:

La marca di END-OF-FILE è un concetto astratto: i diversi S.O. la possono implementare in modi differenti

☺ **UNIX** non inserisce nessun carattere particolare nel file, ma basa il concetto di end-of-file sulla lunghezza del file

☹ **MS-DOS** inserisce un carattere particolare nel file

OPERAZIONI di LETTURA E SCRITTURA

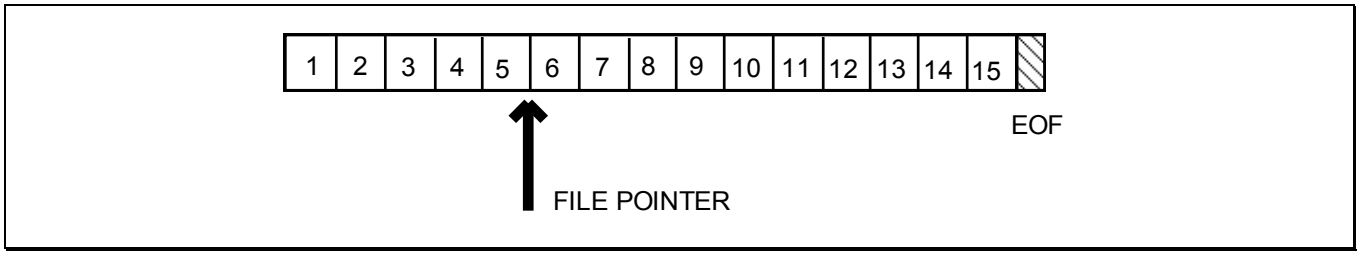
READ `nread = read(fd, buf, n);`

WRITE `nwrite = write(fd, buf, n);`

```
int nread, nwrite, n, fd;
char *buf;
```

lettura e scrittura di un file avvengono a partire dalla **posizione corrente** del file ed avanzano il file pointer all'interno del file

→ restituiscono il **numero dei byte** su cui hanno lavorato
Nel caso di read: se il file pointer è sull'end-of-file viene restituito come valore di ritorno 0



La **READ** tenta di leggere un numero di byte pari a **n**: i caratteri letti si trovano nella memoria puntata da **buf**
nread ===> numero di byte effettivamente letti
Se ci sono dei problemi o se si è incontrato l'END OF FILE, nread sarà diverso da n

La **WRITE** tenta di scrivere un numero di byte pari a **n** presi dalla memoria puntata da **buf**
nwrite ===> numero di byte effettivamente scritti
Se ci sono dei problemi, nwrite sarà diverso da n

LETTURA E SCRITTURA (segue)

NOTA:
buf può essere una porzione di memoria definita tramite un array
esempio: char buf[100];
oppure può essere allocata dall'HEAP
esempio: char *buf;
buf = (char *) malloc(100);
in ogni modo la sua dimensione deve essere >= n

Ogni utente ha la **propria visione** dei file aperti

Nel caso di più utenti che aprono lo stesso file
ogni processo utente ha un proprio
I/O pointer separato

SE un utente legge o scrive, modifica il proprio pointer
gli altri utenti non non subiscono modifiche del proprio I/O pointer

SE un utente chiude un file, non c'è impatto sugli altri utenti

FILE SYSTEM CONDIVISO
Un utente non ha visibilità delle azioni di un altro utente
se non attraverso la **modifica dei dati**

ESEMPI DI LETTURA/SCRITTURA

ESEMPIO 1. COPIA

a. COPIA da un FILE ad un ALTRO

===> EFFICIENZA in base al numero di byte letti

```
#include <fcntl.h>      /* File CoNTroL */
#include <stdio.h>
/* definisce la costante BUFSIZ: tipico valore 512 */
#define PERM 0644
/*  DIRITTI  di  lettura  e  scrittura  al  proprietario,
lettura al gruppo ed agli altri */

main  ()
{   char f1 [20]= "file1",
    f2 [40]= "file2";
    int nread,
        infile, outfile;  /* file descriptor */
    char buffer [BUFSIZ];

infile = open (f1, O_RDONLY);
/* apertura in lettura del file "file1" */
outfile = creat (f2, PERM);
/* creazione del file "file2" */

while ( (nread = read (infile, buffer, BUFSIZ) )> 0 )
/* lettura dal primo file fino a che ci sono caratteri */
       write (outfile, buffer, nread );
/* scrittura sul secondo file dei caratteri letti */

close (infile);  close (outfile);
/* chiusura dei file */
}
```

Legge dal file *file1* e scrive su *file2*

FILE ESEGUIBILE ===> copia1

INVOCAZIONE ===> copia1

N.B.: Il file file1 ***DEVE*** esistere già

Segue COPIA da un FILE ad un ALTRO

ATTENZIONE: Se dopo la lettura si devono analizzare i caratteri conviene leggere un carattere alla volta

```
#include <fcntl.h>
#include <stdio.h>
#define PERM 0644
/*  DIRITTI  di  lettura  e  scrittura  al  proprietario,
lettura al gruppo ed agli altri */

main  ()
{   char f1 [20]= "file1",
    f2 [40]= "file2";
    int nread,
        infile, outfile;  /* file descriptor */
    char ch;
/* definiamo un buffer di solo un carattere */

infile = open (f1, O_RDONLY);
/* apertura in lettura del file "file1" */
outfile = creat (f2, PERM);
/* creazione del file "file2" */

while ( (nread = read (infile, &ch, 1) )> 0 )
/* nella read bisogna passare il puntatore ad una zona di
memoria e quindi in questo caso, l'indirizzo di ch */
{ /* eventuale fase di analisi del carattere letto ch */
       write (outfile, &ch, nread );
/* anche nella write bisogna passare il puntatore ad una
zona  di  memoria  e  quindi  in  questo  caso,  l'indirizzo  di
ch */
}

close (infile);  close (outfile);
/* chiusura dei file */
}
```

b. COPIA da un FILE ad un ALTRO (uso argomenti)

```
#include <stdio.h>
#include <fcntl.h>
#define PERM 0644

main (int argc, char **argv)
{   int infile, outfile, nread;
    char buffer [BUFSIZ];

infile  = open (argv [1], O_RDONLY);
/* apertura in lettura del file il cui nome è in argv[1] */
outfile = creat (argv [2], PERM);
/* creazione del file il cui nome è in argv[2]*/

while (( nread = read (infile, buffer, BUFSIZ)) > 0 )
/* lettura dal primo file fino a che ci sono caratteri */
       write (outfile, buffer, nread);
/* scrittura sul secondo file dei caratteri letti */

close (infile);
close (outfile);
/* chiusura dei file */
}
```

Si passano i nomi dei file come argomenti

FILE ESEGUIBILE ===> copia2
INVOCAZIONI ===> copia2 file1 file2
copia2 copia1.c temp

c. ANCORA ESERCIZIO DI COPIA: PIÙ CONTROLLI

```
#include <fcntl.h>
#include <stdio.h>
#define PERM 0644       /* in UNIX */

int copyfile (char *f1, char * f2)
{   int infile, outfile, nread;
    char buffer [BUFSIZ];  /* usato per i caratteri */

if (( infile = open (f1, O_RDONLY)) < 0) return 2;
/* ERRORE se non si riesce ad aprire in LETTURA il primo
file */

if (( outfile = creat (f2, PERM)) <0 )
/* ERRORE se non si riesce a creare il secondo file */
          {close (infile); return 3; }

while (( nread = read (infile, buffer, BUFSIZ)) > 0 )
{ if ( write (outfile , buffer, nread ) < nread )
/* ERRORE se non si riesce a SCRIVERE */
     { close (infile); close (outfile); return 4; }
    }
close (infile); close (outfile);
/* se arriviamo qui, vuol dire che tutto è andato bene */
return 0;
}

main (int argc, char **argv)
{   int status;
if (argc != 3)  /* controllo sul numero di argomenti */
    { printf ("Errore: numero di argomenti sbagliato\n");
      exit (1); }
status = copyfile (argv[1], argv[2]);
if (status != 0)
    printf ("Ci sono stati degli errori nella copia\n");
exit (status);
}
```

FILE ESEGUIBILE ===> copia
INVOCAZIONI ===> copia file1 file2
copia copia1.c temp

Dopo la esecuzione controllare il return code
⇨ in UNIX, **echo $?**

d. Con RIDIREZIONE ⇨ FILTRO

```
#include <stdio.h>

main ()
{   char buffer [BUFSIZ];
    int nread;

while ( (nread = read (0, buffer, BUFSIZ)) > 0 )
/*  lettura  dallo  standard  input  fino  a  che  ci  sono
caratteri */
       write (1 , buffer, nread);
/* scrittura sullo standard output dei caratteri letti */
}
```

Il sistema esegue automaticamente i collegamenti con lo STANDARD INPUT e lo STANDARD OUTPUT
➔ non c'è bisogno di aperture o creazioni e di chiusure

La lettura da fd=0 legge dallo **standard input**
la scrittura su fd=1 scrive su **standard output**
Questi due **file descriptor** sono aperti *automaticamente* dal **sistema** (shell) e collegati all'I/O

In Unix, **CTRL-D** rappresenta la fine file dello STANDARD INPUT (in MS-DOS, invece è CTRL-Z)

Completa Omogeneità dei file con i dispositivi
fd = open ("/dev/printer", O_WRONLY);
Anche per i dispositivi usiamo le stesse primitive
open, read, write, close

FILE ESEGUIBILE ===> copiarid

INVOCAZIONI ===> copiarid
copiarid **<** copia1.c
copiarid **<** copia1.c **>** temp

ESEMPIO 2. INSERIMENTO DI CARATTERI IN UN FILE

```
#include <stdio.h>
#include <fcntl.h>
#define PERM 0644       /* in UNIX */

main (int argc, char **argv)
{   int fd;
    char *buff;
    int nr;

printf("il nome del file su cui inserire
       i caratteri è %s\n", argv[1]);

buff=(char *)malloc(80);
/* bisogna ALLOCARE memoria per il BUFFER */

if ((fd = open(argv[1], O_WRONLY)) < 0)
    fd = creat(argv[1], perm);
/* se il file esiste, viene aperto in lettura, altrimenti
viene creato */

printf("Aperto o creato con fd = %d\n", fd);

while ((nr=read(0, buff,80)) > 0)
    write(fd, buff, nr);

close(fd);
}
```

OSSERVAZIONE:

Leggere BUFSIZ caratteri alla volta, oppure 80 oppure 1 alla volta in questi due primi esempi è completamente indifferente a livello logico

⇨ potrà incidere eventualmente solo a livello di prestazioni

Diventa importante leggere un carattere alla volta se dobbiamo fare una analisi/trasformazione di un carattere alla volta

FILE E MULTIUTENZA

Ogni utente ha un identificatore detto **uid** (user id) e appartiene a un gruppo **gid** (group id), come specificato nel file /etc/passwd.

Esempio:

```
letizia:ITZ7b:250:100:Leonardi:/home/letizia:/bin/sh
```

Un processo acquisisce **uid** e **gid** dell'utente che lo lancia

Il kernel memorizza per ogni file **user id** ed **group id** del processo creatore

Un processo può accedere a un file se:

- uid processo == 0 (⇨ superutente)
- uid processo == uid proprietario file e diritti OK
- uid processo != uid proprietario file ma
- gid processo == gid proprietario file e diritti OK
- uid e gid proc != uid e gid file, ma diritti other OK

Attenzione: in realtà il kernel guarda **effective uid** e **gid** del processo che accede al file

IMPLEMENTAZIONE RIDIREZIONE

Supponiamo, quindi, di aver richiesto al Sistema Operativo (MS-DOS o UNIX che sia) di eseguire il programma *prova* ridirezionando sia lo standard input che lo standard output:

```
prova < dati > risultati
```

dove *dati* sia il nome del file da cui si devono leggere i dati e *risultati* sia il nome del file su cui devono essere scritti i risultati

Il Sistema Operativo, prima di mandare in esecuzione il programma *prova*, effettua le seguenti operazioni a basso livello:

```
close(0);
open("dati", O_RDONLY);
close(1);
creat("risultati", modo);
```

prima chiusura
⇨ libera l'elemento di posto 0 della tabella dei file aperti;
apertura seguente
⇨ occupa proprio quell'elemento e quindi il valore ritornato come file descriptor è 0
seconda chiusura
⇨ libera l'elemento di posto 1 della tabella dei file aperti
creazione seguente
⇨ va ad occupare proprio quell'elemento e quindi il valore ritornato come file descriptor è 1

Quindi, il Sistema Operativo manda in esecuzione il programma *prova* che, tutte le volte che leggerà dallo standard input e scriverà sullo standard output, in realtà effettuerà letture dal file *dati* e scritture sul file *risultati*

OPERAZIONI non SEQUENZIALI

LSEEK

```
newpos = lseek(fd, offset, origin);
long int newpos, offset;   int fd;
int origin;
/* 0 dall'inizio, 1 dal corrente, 2 dalla fine*/
```

Questa funzione sposta il file pointer all'interno del file, identificato dal file descriptor *fd*, di *offset* caratteri (byte) a partire dalla posizione data da *origin* ➔ *origin* può essere specificato in modo simbolico includendo il file <unistd.h> e usando le costanti **SEEK_SET** (inizio), **SEEK_CUR** (corrente) o **SEEK_END** (fine)

Il valore di offset può essere un long integer positivo o negativo

Il valore ritornato rappresenta la posizione corrente del file pointer a partire dall'inizio del file cioè il numero di byte a partire dall'inizio del file ove si trova il file pointer

Le successive operazioni di lettura/scrittura saranno applicate a partire dalla nuova posizione

Ad esempio, per tornare all'inizio del file basterà scrivere:

```
lseek(fd, 0L, 0);
```

dove 0L rappresenta il valore zero espresso come costante "lunga"

Nel caso che ci si sposti alla fine del file ===>
newpos rappresenta la lunghezza (in byte) del file

ESERCIZIO:
Calcolare la lunghezza di un file il cui nome viene passato come argomento: **NON si utilizzi la funzione lseek**

NOTA BENE: La funzione lseek() non restituisce errore se con il file pointer si "esce" dal file in avanti o indietro

ESEMPI DI USO DI LSEEK:

ESEMPIO 1: Vengono inserite in un file (***senza distruggerne il contenuto***) delle stringhe, lette da input, solo se soddisfano una certa condizione. Il nome del file è un parametro del programma.

```
#include <stdio.h>
#include <fcntl.h>
#define PERM 0644       /* in UNIX */

int pattern (s)
char *s;
{ /* restituisce 1 solo se il secondo carattere è uguale
a 's' e se il penultimo è una cifra */
    return (  s[1] == 's' &&
              s[strlen(s)-2] >= '0' &&
                 s[strlen(s)-2] <= '9'   ? 1 : 0);
}

main (int argc, char **argv)
{   int fd;
    char stringa [80], answer [3], eol = '\n';
    long int pos = 0;
printf("il nome del file su cui inserire le stringhe è
%s\n", argv[1]);
if ((fd = open(argv[1], O_WRONLY)) < 0)
       fd  = creat(argv[1], PERM);
/* apertura in scrittura, se non esiste, creazione */
else   pos = lseek(fd, 0L, 2);
/* se il file esiste, ci si posiziona alla fine */
printf ("il file contiene %ld byte\n", pos);
while (   printf("Vuoi finire?(si/no)\n"),
              scanf("%s", answer), strcmp (answer,"si")    )
    {   printf("fornisci la stringa da inserire\n");
        scanf("%s", stringa);
        if (pattern(stringa)) {
/* se si soddisfa il pattern, si inserisce nel file */
                    write(fd, stringa, strlen(stringa));
                    write(fd, &eol, 1);
                          }   };
close (fd);
}
```

ESEMPIO 2: Viene appeso ad un file (parametro del programma) il contenuto di un altro file. Quest'ultimo è lo standard input: **possibilità di ridirezione**

```
#include <fcntl.h>
#include <stdio.h>
#define PERM 0644       /* in UNIX */

int appendfile (char *f1)
{  int outfile, nread;  char buffer [BUFSIZ];
if ( (outfile = open ( f1, O_WRONLY)) < 0 )
    /* apertura in scrittura */
    { if (( outfile = creat ( f1, PERM)) <0 )
        /* se il file non esiste, viene creato */
          return (-1);  }
else lseek (outfile, 0L, 2);
    /* se il file esiste, ci si posiziona alla fine */
while (( nread = read (0, buffer, BUFSIZ)) > 0 )
    /* si legge dallo standard input */
{   if ( write (outfile, buffer, nread ) < nread )
    { close (outfile); return (-2); /* errore */ }
}/* fine del file di input */
close (outfile); return (0);
}

main (int argc, char ** argv)
{ int integi;
if (argc <= 1) /* controllo sul numero di argomenti */
{ printf ("ERRORE: almeno un argomento \n"); exit (-3); }
integi = appendfile (argv[1]);
exit (integi);
}
```

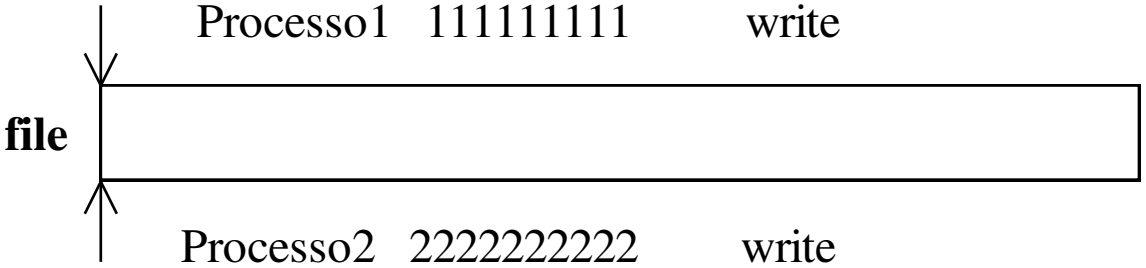
POSSIBILI INVOCAZIONI:

- `append  fff`
  abc
  def
  <fine-file> ⇨ si appende al file fff ciò che c'è sullo stdinput
- `append  fff  <  aaa`
  ⇨ si appende al file fff tutto ciò che c'è nel file aaa

OSSERVAZIONI:

* Operazioni sui dispositivi e file **solo sincrone**
  cioè con attesa del completamento dell'operazione

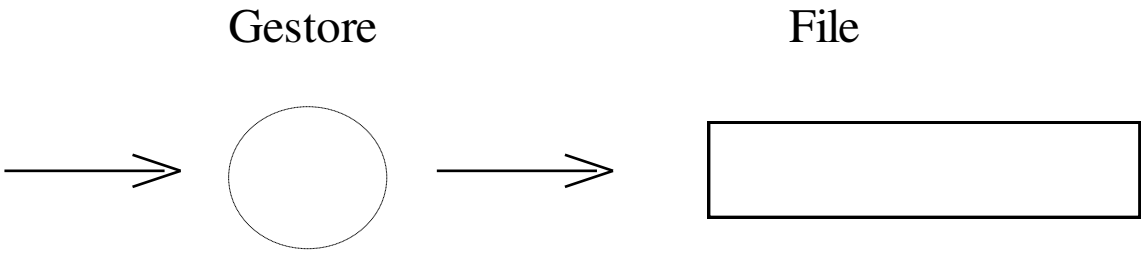
* **ATOMICITÀ della SINGOLA OPERAZIONE**
  di lettura/scrittura e di azione su un file
  ===> **operazioni primitive** cioè azioni elementari e non interrompibili della macchina virtuale UNIX



NON è garantita la **atomicità** delle sequenze di operazioni

Per esempio, ***se più processi mandano file sulla stampante*** si possono mescolare le linee inviate alla stampante!!!!

===> Definizione di un **gestore** che incapsula la risorsa



Tipi di file

- file **ordinari**
- file **directory**
- file **speciali** (dispositivi fisici) in /dev

speciali orientati al **carattere** (sono tipicamente stampanti, terminali, linee telefoniche)

speciali orientati al **blocco** (dischi,nastri)

OPERAZIONI di LINK e UNLINK

LINK

```
retval= link(name1, name2);
    char *name1, name2;
    int retval;
```

Questa primitiva consente in UNIX di creare un nuovo NOME (un link) per un file esistente
⇨ viene incrementato il numero di link

UNLINK

```
retval= unlink(name);
    char *name;
    int retval;
```

Questa primitiva consente di cancellare un file

In UNIX, in realtà, come dice il suo nome, il suo compito è cancellare un link ⇨ nel caso in numero di link arrivi a ZERO allora si opera anche la DISTRUZIONE del file cioè la liberazione dello spazio su disco

Tramite l'uso di queste due primitive viene realizzato, ad esempio, il comando **mv** di UNIX

ESEMPIO: IMPLEMENTAZIONE DEL COMANDO UNIX MV (versione semplificata)

```
#include <stdio.h>
#include <stdlib.h>
main (int argc, char **argv)
{ if (argc != 3) { printf ("Errore\n"); exit(-1); }
/* controllo del numero di parametri */
if (link(argv[1], argv[2]) < 0)
    { printf ("Errore\n"); exit(-2);}
/* controllo sulla operazione di link */
if (unlink(argv[1]) < 0)
    { printf ("Errore\n"); exit(-3);}
/* controllo sulla operazione di unlink */
printf ("Ok\n"); exit(0);
}
```

Si considerino eventuali estensioni

ALTRE PRIMITIVE SUI FILE

1) Verifica sui diritti di accesso ad un file

```
ACCESS      retval = access (pathname, amode);
                     char * pathname;
                     int amode;
                     int retval;
```

La primitiva ACCESS consente di verificare il tipo di accesso consentito su un file

Il parametro ***amode*** può essere:
04 read access; 02 write access
01 execute access; 06 read e write access; 00 existence

access restituisce il valore 0 in caso di successo, altrimenti un valore negativo

2) Verifica dello stato di un file

```
STAT        retval = stat (pathname, &buff);
                     char * pathname;
                     struct stat buff;
/* struttura che rappresenta il descrittore del file */
                     int retval;

FSTAT       retval = fstat (fd, &buff);
                     int fd;
```

FSTAT può essere usato solo se il file è già aperto

Entrambe le primitive, tornano il valore 0 in caso di successo, altrimenti un valore negativo

Vediamo quali possono essere i campi della ***struct stat***:

```
struct stat  {  ushort  st_mode;        /* modo del file */
    ino_t   st_ino;         /* I_node  number */
    dev_t   st_dev;         /* ID del dispositivo */
    dev_t   st_rdev;        /* solo per file speciali */
    short   st_nlink;       /* numero di link */
    ushort  st_uid;         /* User ID del proprietario */
    ushort  st_gid;         /* Group ID del proprietario */
    off_t   st_size;        /* Lunghezza del file in byte */
    time_t  st_atime;       /* tempo dell'ultimo accesso */
    time_t  st_mtime;       /* tempo dell'ultima modifica*/
    time_t  st_ctime; /* tempo dell'ultimo cambiamento di stato */}
```

ESEMPIO di uso di ACCESS

```
#include <unistd.h>

main(int argc, char **argv)
{
    if (argc < 2)
       { printf("ERRORE!\n"); exit(-1);}

    if (access(argv[1], F_OK) == 0)
              printf("OK file\n");
    else      printf("NO file\n");

    if (access(argv[1], R_OK) == 0)
              printf("OK lettura\n");
    else      printf("NO lettura\n");

    if (access(argv[1], W_OK) == 0)
              printf("OK scrittura\n");
    else      printf("NO scrittura\n");

    if (access(argv[1], X_OK) == 0)
              printf("OK esecuzione\n");
    else      printf("NO esecuzione\n");
}
```

NOTA BENE: **access** verifica i diritti dell'utente, cioè fa uso del **real** uid del processo (e non dell'effective uid)

ESEMPIO di uso di STAT

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

main(int argc,   char **argv)
{ struct stat b;
    if (argc < 2) { printf("ERRORE!\n"); exit(-1);}
    if (stat(argv[1], &b) != 0)
        { printf("NO!!!\n"); exit(-2); }
    printf("st-dev=%ld\n", b.st_dev);
    printf("st-ino=%ld\n", b.st_ino);
    printf("st-mode=%ld\n", b.st_mode);
    printf("st-nlink=%ld\n", b.st_nlink);
    printf("st-uid=%ld\n", b.st_uid);
    printf("st-gid=%ld\n", b.st_gid);
    printf("st-rdev=%ld\n", b.st_rdev);
    printf("st-size=%ld\n", b.st_size);
    printf("st-atime=%ld\n", b.st_atime);
    printf("st-mtime=%ld\n", b.st_mtime);
    printf("st-ctime=%ld\n", b.st_ctime);
}
```

ESEMPIO di uso di FSTAT

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

main(int argc, char *argv)
{ int fd; struct stat b;
    if (argc < 2) { printf("ERRORE!\n"); exit(-1);}
    if ((fd=open(argv[1], 0)) <= 0)
           { printf("NO OPEN!!!\n"); exit(-2);}
    if (fstat(fd, &b) != 0)
           { printf("NO!!!\n"); exit(-3); }
    printf("st-dev=%ld\n", b.st_dev);
    printf("st-ino=%ld\n", b.st_ino);
    printf("st-mode=%ld\n", b.st_mode);
    printf("st-nlink=%ld\n", b.st_nlink);
    printf("st-uid=%ld\n", b.st_uid);
    printf("st-gid=%ld\n", b.st_gid);
    printf("st-rdev=%ld\n", b.st_rdev);
    printf("st-size=%ld\n", b.st_size);
    printf("st-atime=%ld\n", b.st_atime);
    printf("st-mtime=%ld\n", b.st_mtime);
    printf("st-ctime=%ld\n", b.st_ctime);
}
```