Sistemi operativi

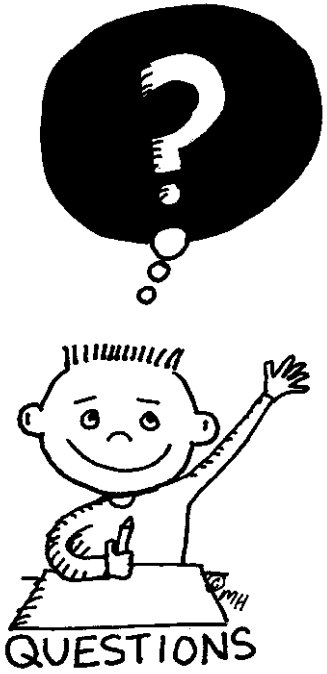
Corso di Laurea Triennale in Ingegneria Informatica

Lezione 10

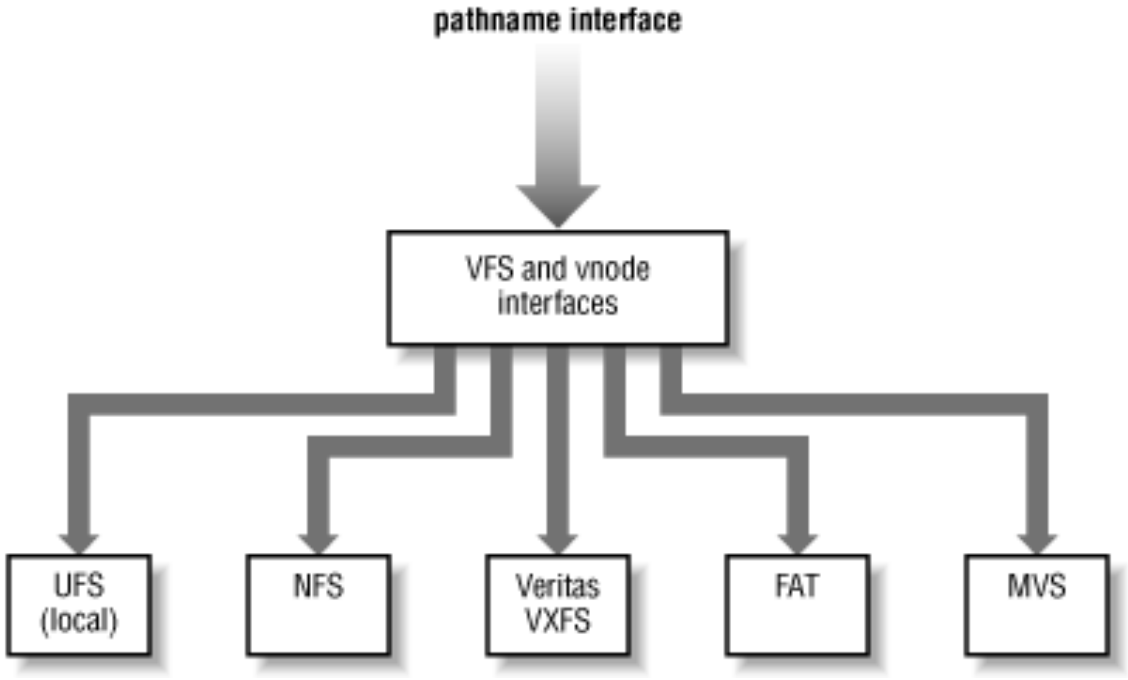
- Virtual Filesystem
- mount, umount
- I/O, Unix I/O,
- Standard I/O
- Pipe e Fifo

Fabio Del Vigna - fabio.delvigna@for.unipi.it

Domande sulle lezioni passate?

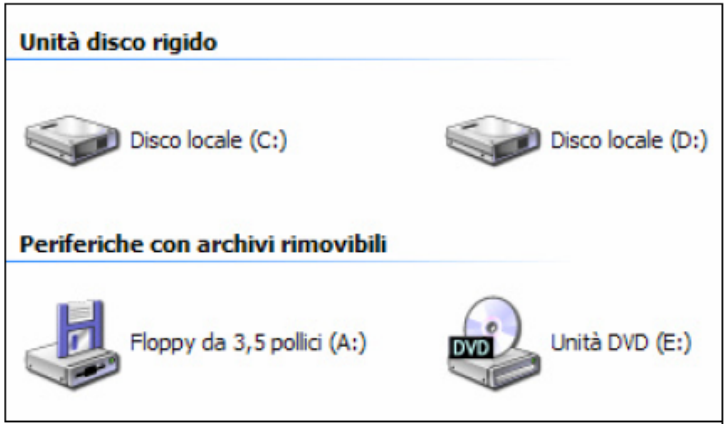


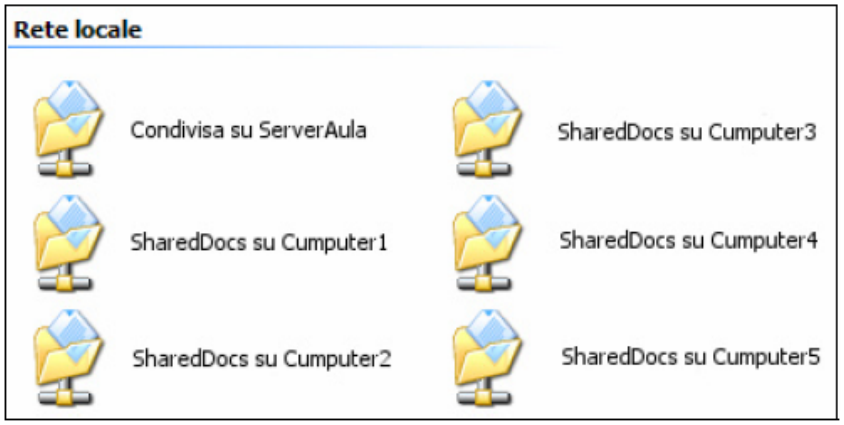
Virtual File System



Accesso ai file in Windows

- Accesso locale
  - unità individuate da una lettera seguita da “ : ”
  - esempio `C:\Windows\Programmi\`
- Unità remote (unità condivise)
  - individuate da un nome preceduto da \\
  - esempio `\\serveraula\condivisa`





Accesso ai file in UNIX

- **everything is a file**
- virtual file system (**VFS**)
  - interfaccia per l'accesso a file system differenti
  - trasparente all'utente.
- Individuazione dei dispositivi
  - periferiche sono riferite come file speciali in `/dev`
  - unità viste come parte di un unico file system globale con radice in root (/).

File in Unix (1 / 3)

- **everything is a file**
  - Garantisce omegeneità del sistema

- Tre categorie di file:
  - File ordinari
  - Direttori (*directory*)
  - Dispositivi (*file speciali*)

File in Unix (2 / 3)

- **File: nome, i-node, i-number** (considerare le spiegazioni teoriche)
  - Garantisce omegeneità del sistema

File in Unix (3 / 3)

- Ad ogni file possono essere associati uno o più nomi simbolici

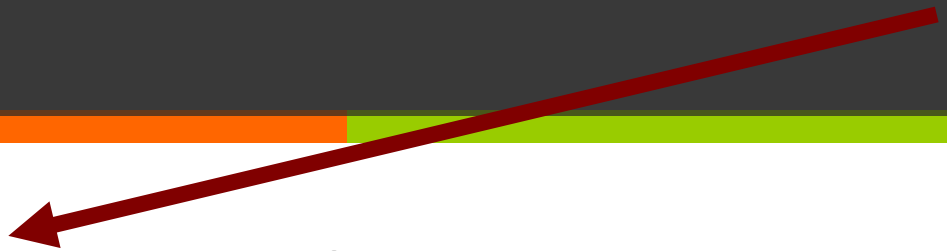
MA

- Ad ogni file è associato **uno e un solo descrittore (i-node)**, univocamente identificato da un intero (i-number)

Link

- Si possono creare **link** (collegamenti) a file contenuti nel filesystem
- Vedere l'uso del comando **ln**: `man ln`

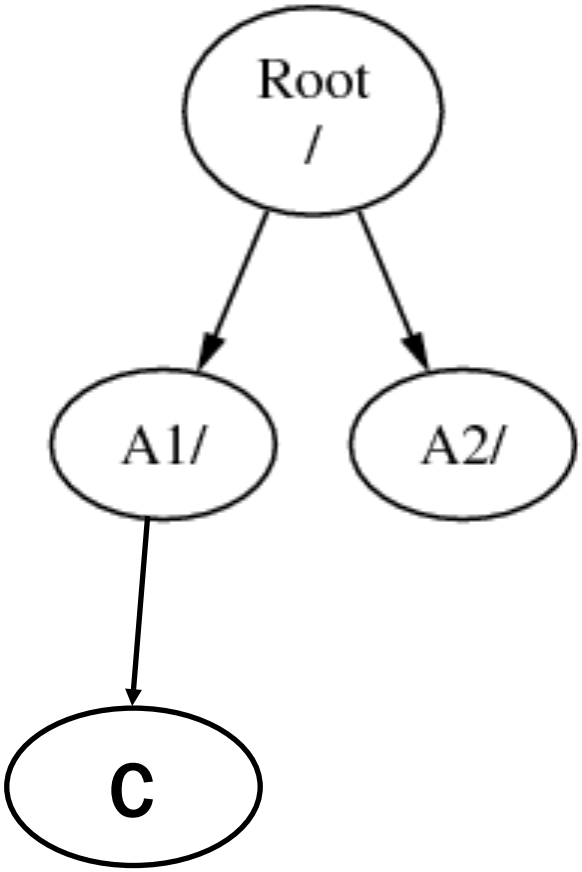
I-node



Descrittore del file

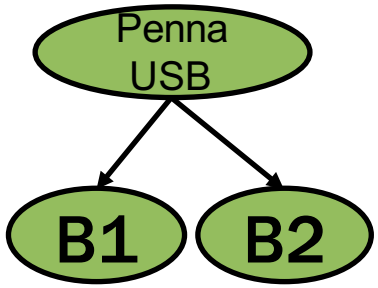
- Tra gli attributi contenuti nell'i-node vi sono:
  1. **tipo di file**: *Ordinario – Direttorio – file speciale*
  2. **proprietario, gruppo** (user-id, group-id)
  3. dimensione
  4. data
  5. **numero di links**

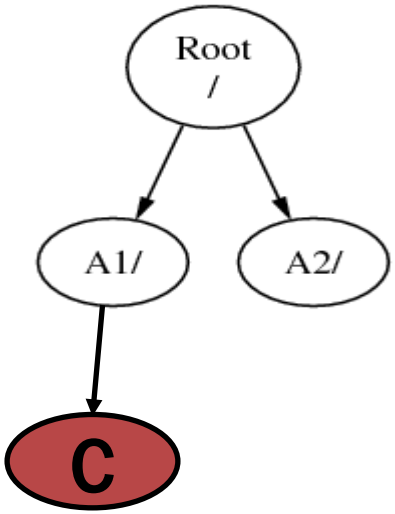
Montare un file system: (1/ 3)



- File system (**F**)
- Supponiamo di avere una penna USB (identificata come file system **B**).
- Per accedere al filesystem **B**, è necessario ***montare*** la penna nel file system (**F**).

Montare un file system: (2 / 3)



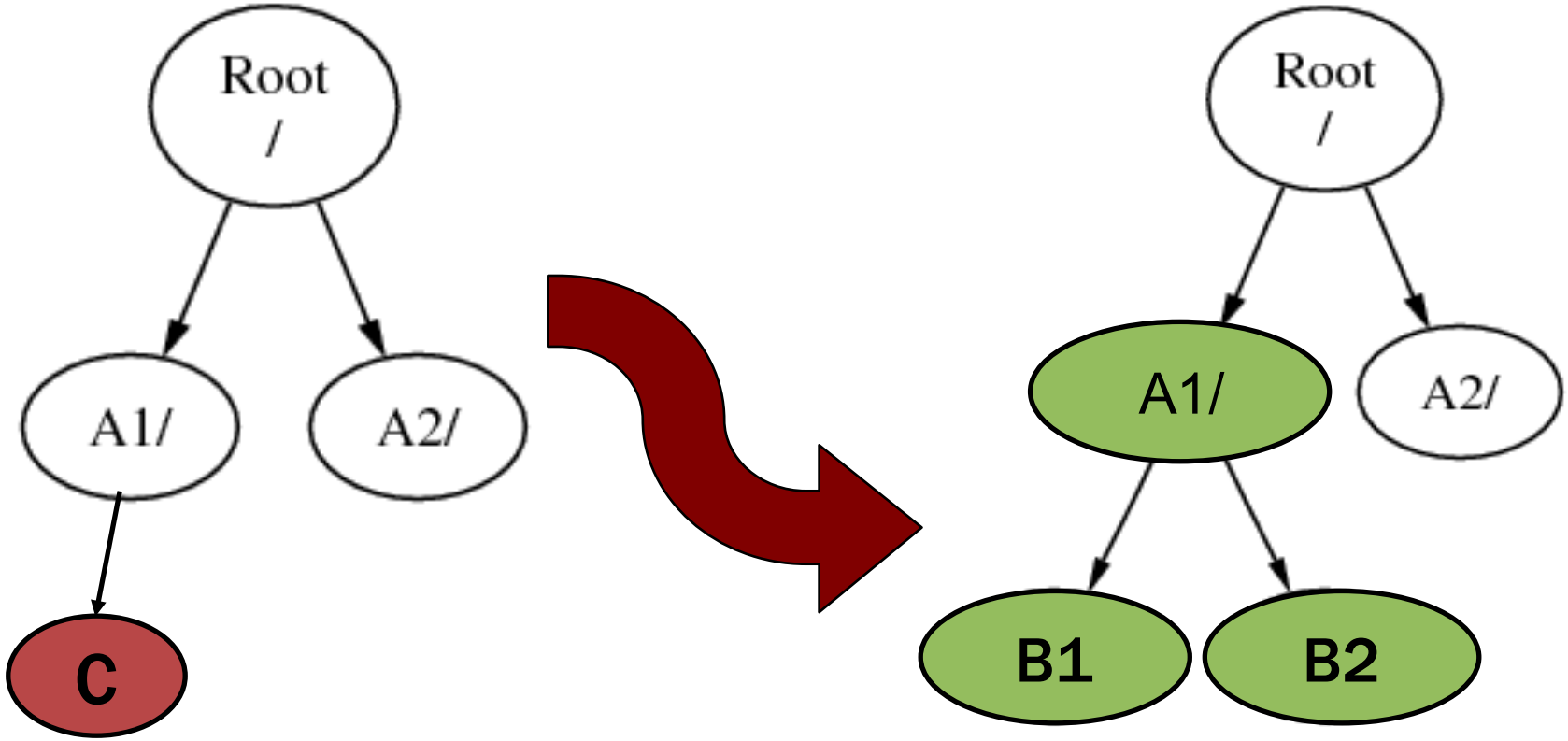


- Scegliere dove “mettere” il file system **B**, es. **A1**
- Montare **B** sulla directory **A1**
- **A1** è “sostituito” dalla directory root del file system **B** contenente le due directory **B1** e **B2**
- I file che erano in **/A1** sono nascosti, tornano visibili quando **B** è smontato da **A1**

Montare un file system: (3 / 3)

Prima di montare il file system B

Dopo aver montato il file system B



Individuazione dei dispositivi in FreeBSD

Dispositivo	Nome
Hard disk IDE	`/dev/adX`
Unità ottiche IDE	`/dev/acdX`
Dischi SCSI e USB mass storage	`/dev/daX`
Unità ottiche SCSI	`/dev/cdX`
Lettori floppy	`/dev/fdX`

Tipi di file system

Tipo di filesystem	Nome
Unix filesystem	`ufs`
Network filesystem	`nfs`
MSDOS (include FAT)	`msdos`
NTFS (Windows NT, 2000, XP, 2003)	`ntfs`
Partizione di scambio	`swap`
ISO9660 (CD-ROM)	`cd9660`
UDF (DVD-ROM)	`udf`

Comando `mount` (1 / 3)

`mount` permette di montare un file system su una directory, in modo da rendere accessibili i file e le directory all'interno del file system.

`mount [op.] [file system] [mount point]`

- `op.` (opzioni)
  - –t → il tipo del file system da montare
  - -o rw → (lettura/scrittura)
  - -o ro → (solo lettura)
  - …

Comando `mount` (2 / 3)

- **`file system`**, dispositivo su cui risiede il file system da montare
- **`mount point`**, directory in cui montare il file system

`mount`, *operazioni preliminari:*

- creare una cartella in cui montare:

`mkdir /mnt/usb`

Comando `mount` (3 / 3)

- **Montare un floppy disk**
  - `mount -t msdos /dev/fd0 /mnt/floppy`
- **Montare un CD**
  - `mount -t cd9660 /dev/acd0 /mnt/cdrom`
- **Montare una penna USB**
  - `mount -t msdos /dev/da0 /mnt/usb`

Comando `umount` (1 / 2)

`umount` smonta i filesytem, cioè esegue l'operazione inversa di `mount`.

`umount [opzioni] [file system] [mount point]`

- `file system`, **dispositivo su cui risiede il file system da smontare**
- `mount point`, **directory da cui smontare il file system**

Comando umount (2 / 2)

- `umount`

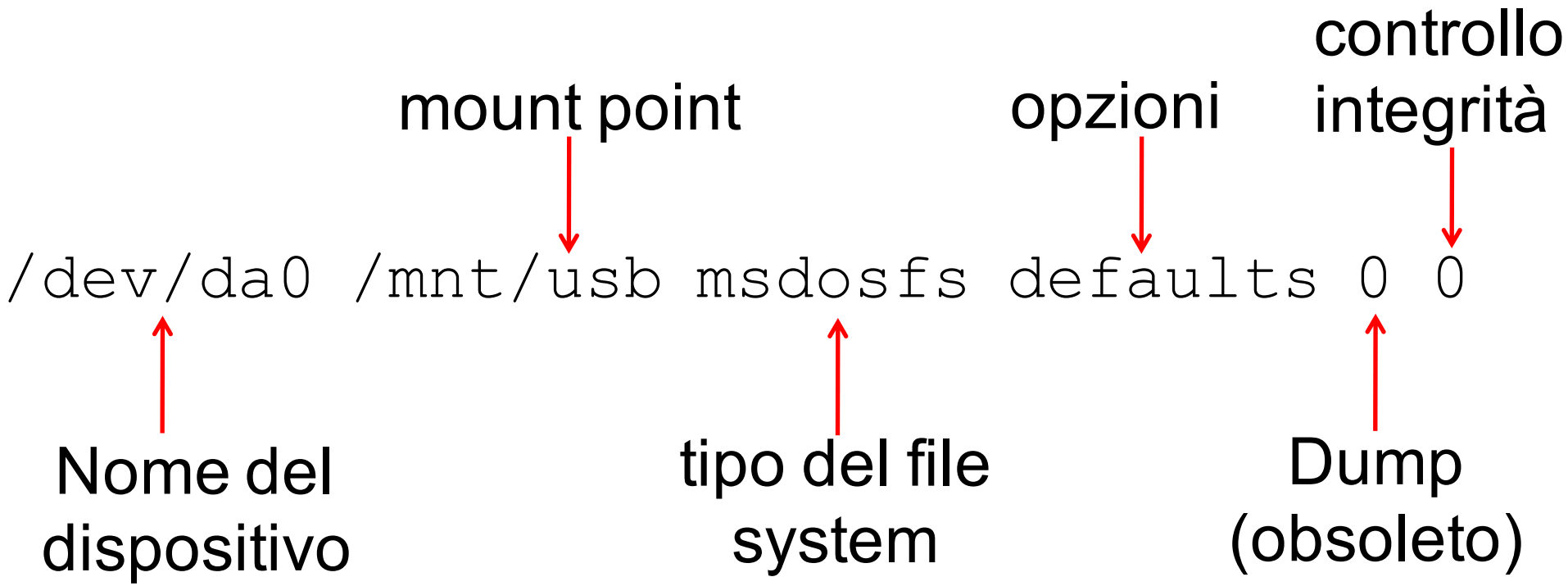
- Smontare un floppy disk
  - `umount /dev/fd0`
  - `umount /mnt/floppy`
- Smontare un CD
  - `umount /dev/acd0`
  - `umount /mnt/cdrom`
- Smontare una penna USB
  - `umount /dev/da0`
  - `umount /mnt/usb`

/etc/fstab(1/4)

/etc/fstab

- definisce le caratteristiche e le directory di innesto dei vari file system
- letto solo dai programmi
- aggiornamento fatto manualmente dall'amministratore del sistema

/etc/fstab (2 / 4)


mount point
opzioni
controllo integrità
/dev/da0 /mnt/usb msdosfs defaults 0 0
Nome del dispositivo
tipo del file system
Dump (obsoleto)

/etc/fstab (3 / 4)

Campo	Significato
Tipo dispositivo	Tipo di dispositivo da montare
Mount point	Punto di innesto per il filesystem
Tipo del filesystem	Tipo del filesystem
Opzioni	Opzioni per il montaggio

/etc/fstab (4 / 4)

Opzioni di montaggio:

- `default`: **Impostazioni predefinite:** `rw, exec, auto, async`
- `sync | async`: **I/O sul file system in modo *sincrono* o *asincrono***
- `auto | noauto`: ***Permette o impedisce* il montaggio automatico**
- `exec | noexec`: ***Permette o impedisce* l'esecuzione di file binari**

Comando `mount` (opzioni aggiuntive)

- `mount -a`
  - **monta tutti i file system elencati nel file `/etc/fstab`, con l'eccezione dei file specificati come `noauto`, quelli specificati dall'opzione `-t` o quei file system che sono già montati.**
- `mount -ta`
  - **monta i file system definiti in `/etc/fstab` ma solo se il loro tipo corrisponde a quello specificato dall'opzione `-t`.**

Comando `umount` (opzioni aggiuntive)

- `umount -a`
  - **smonta tutti i file system (sconsigliato) definiti in** `/etc/fstab`
- `umount -at`
  - **smonta tutti i file system del tipo corrispondente a quello specificato dall'opzione** `-t` **definiti in** `/etc/fstab`

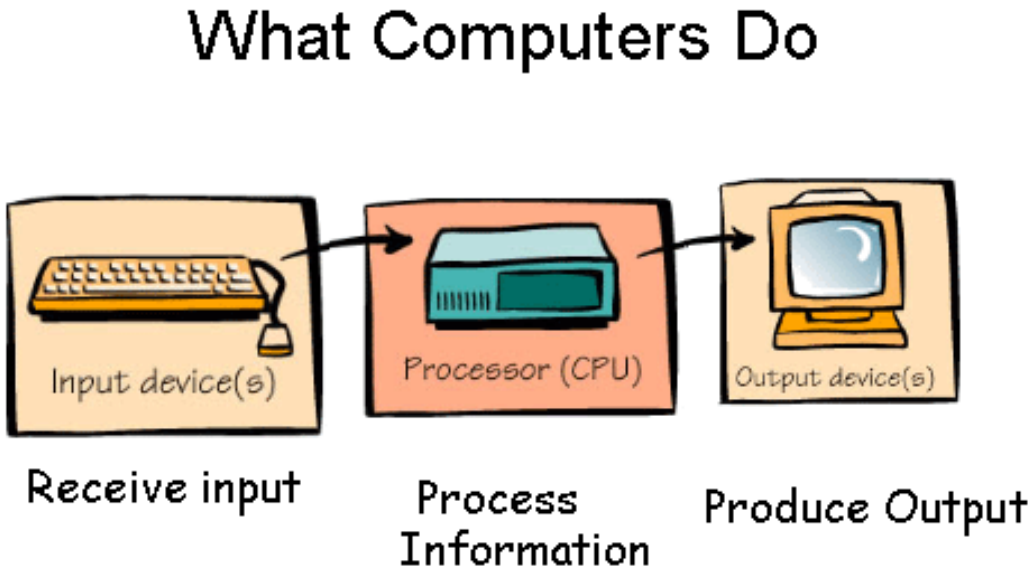
Organizzazione del filesystem

- Il filesystem (a livello logico) di Debian, come quello di altri sistemi Unix, ha un'*organizzazione prefissata*

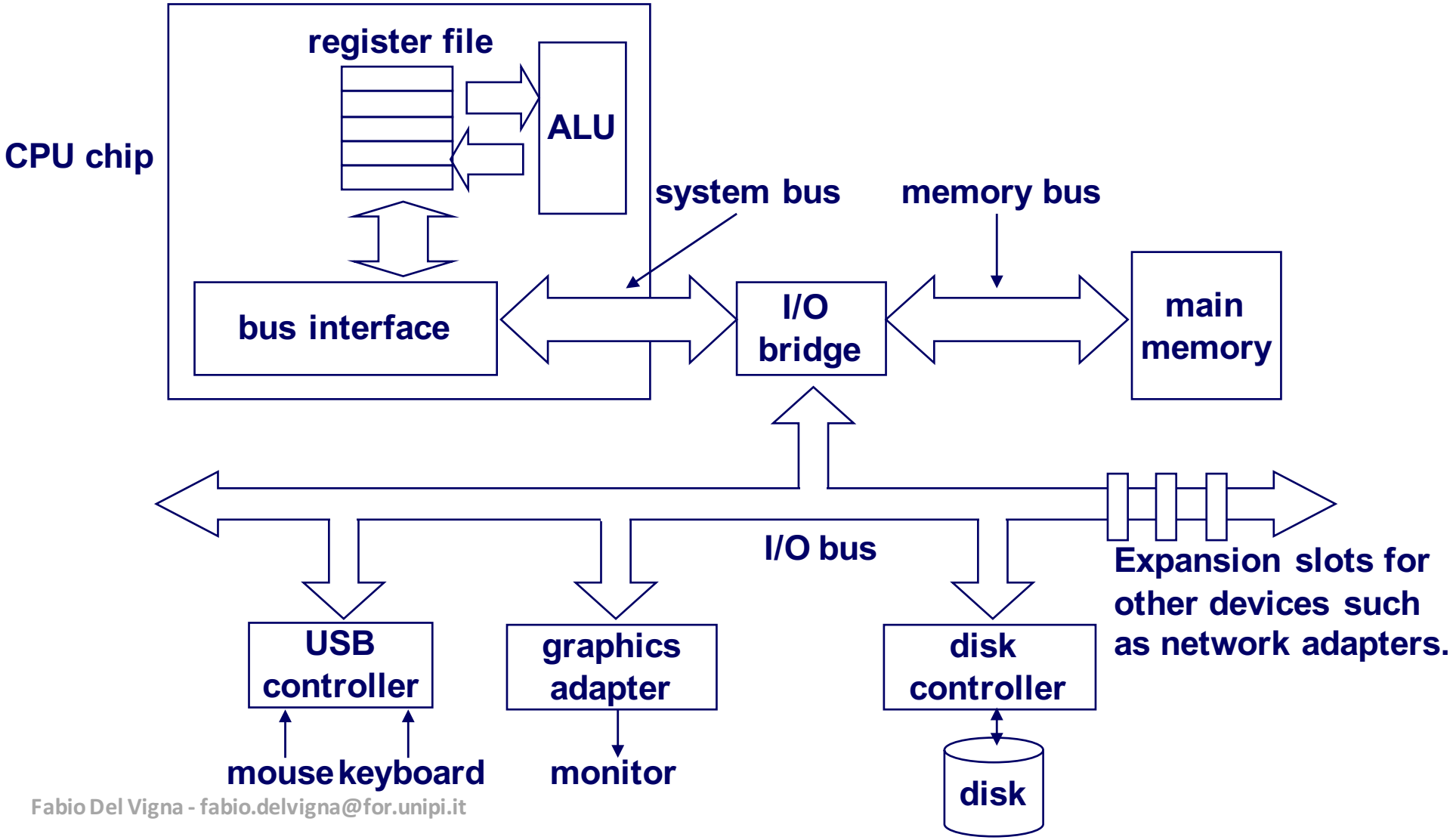
- Tale organizzazione è detta *filesystem hierarchy*
  - **`man hier`**

I/O

Introduzione



Sistema Hardware



Caratteristiche dell'I/O

I/O Unix/Linux Classico

- I/O su stream di byte
  - Può riposizionare il punto di inserimento ed estendere i file alla fine
- I/O sincrono
  - Legge o scrive blocchi di dati fino al completamento dell'operazione
- Granularità fine
  - Scritture passo-passo
  - Ogni operazione di I/O è gestita dal kernel e un processo appropriato

I/O Mainframe

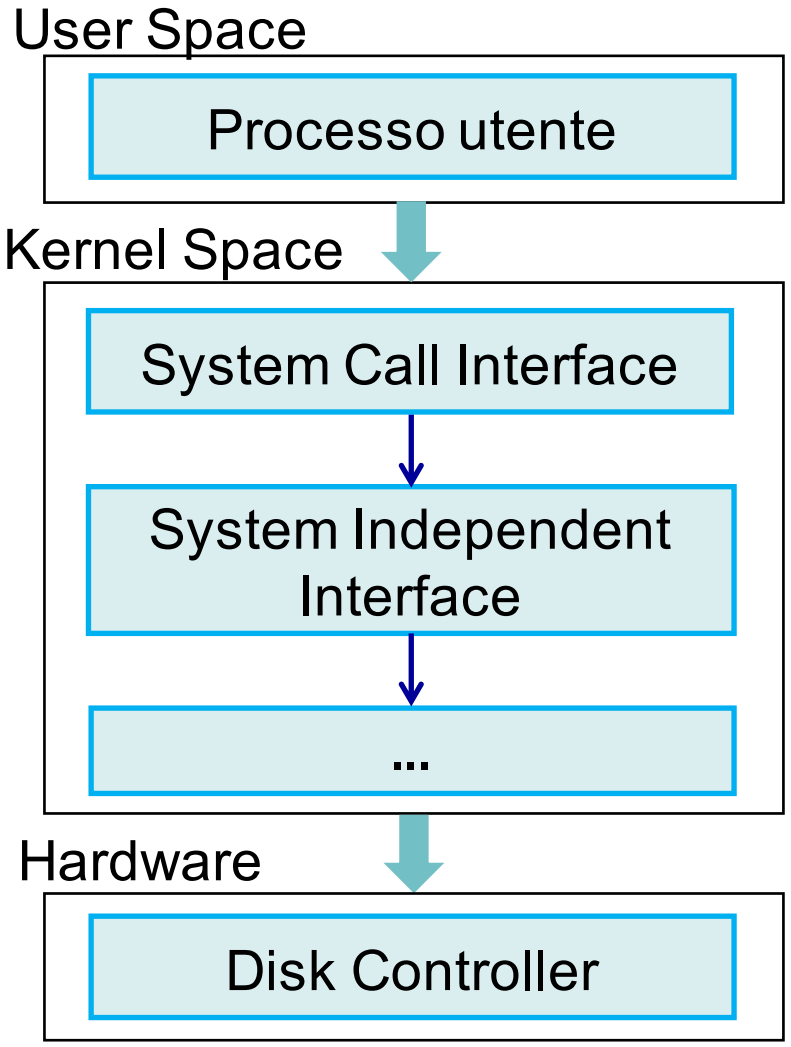
- I/O su record strutturati
  - Utilizza funzioni per inserire, rimuovere e aggiornare i record
- I/O asincrono
  - Può sovrapporre esecuzione e I/O all'interno di un processo
- Granularità bassa
  - Il processo scrive su canali che sono eseguiti dallo hardware per I/O
  - Le operazioni sono in genere eseguite autonomamente con un solo interrupt al termine dell'esecuzione

I/O

- L'I/O nei sistemi UNIX è gestito tramite diverse interfacce:
  - UNIX I/O
  - Standard I/O

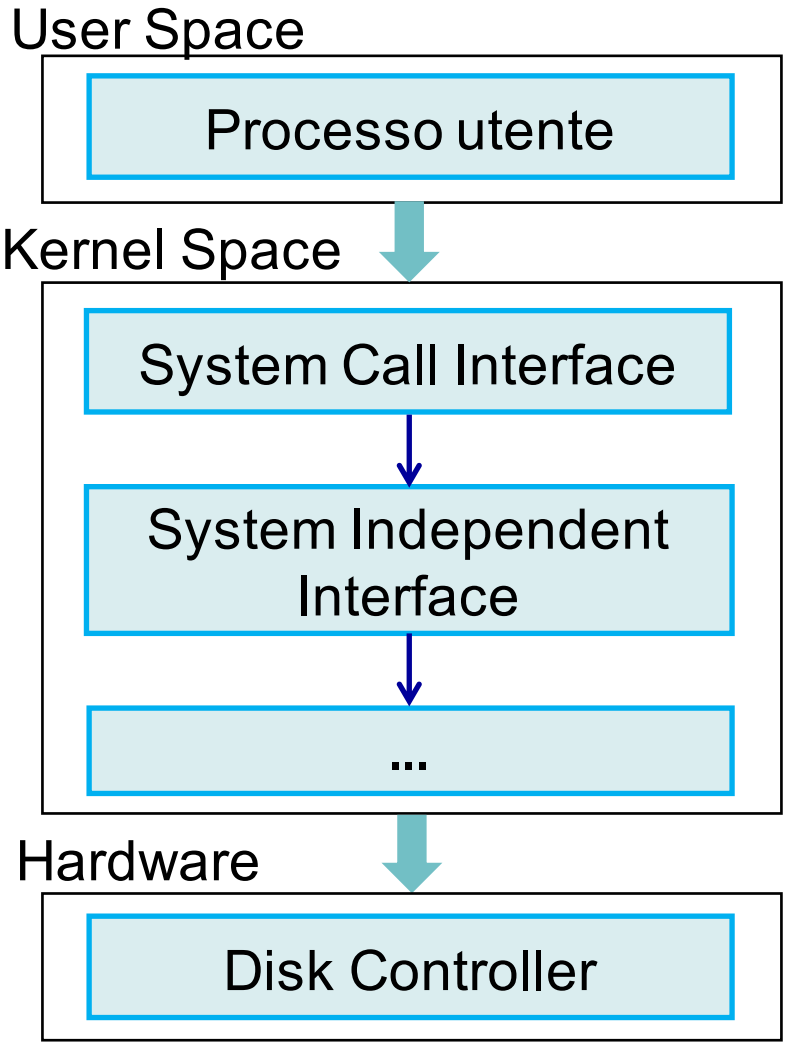
- Si possono sviluppare altre interfacce a partire da quelle esistenti

Schema di funzionamento del VFS



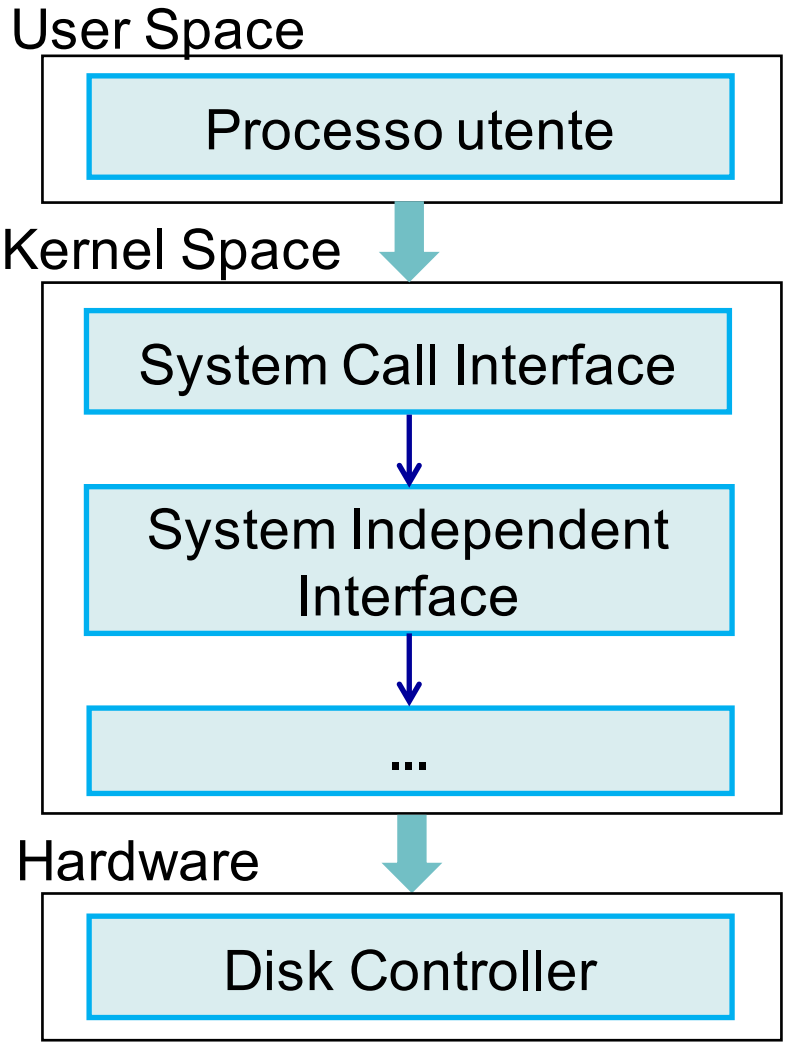
- Virtual file system
  - permette la coesistenza di file system diversi nello stesso albero delle cartelle,
  - fornisce delle operazioni di input/output **indipendenti** dai dispositivi,
  - i programmi utente possono usare la **stessa interfaccia** per la manipolazione dei file.

Accesso al Filesystem 1/2



- fornisce delle operazioni di input/output **indipendenti** dai dispositivi
- i programmi utente possono usare la **stessa interfaccia** per la manipolazione dei file.

Accesso al Filesystem 2/2

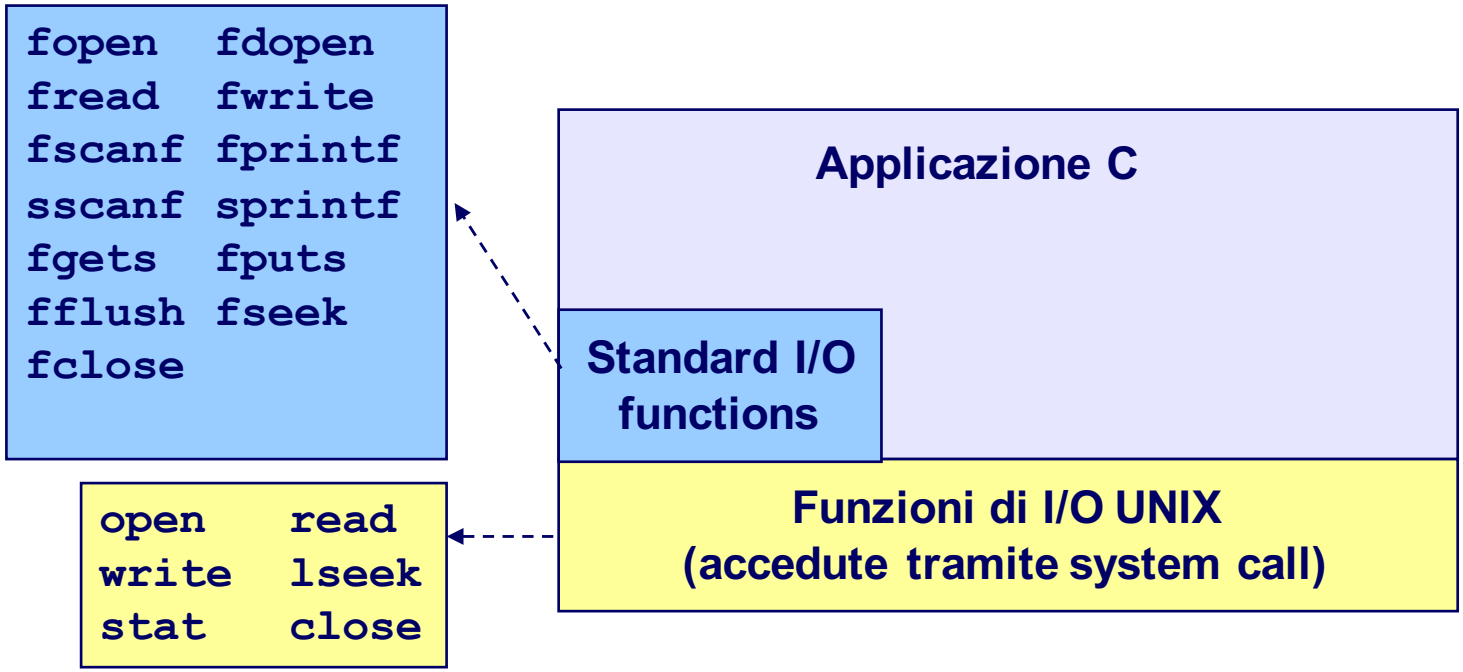


Standard I/O

UNIX I/O

Unix I/O vs Standard I/O

Standard I/O è implementato usando l'I/O UNIX (basso livello)



Quale usare?

Unix I/O

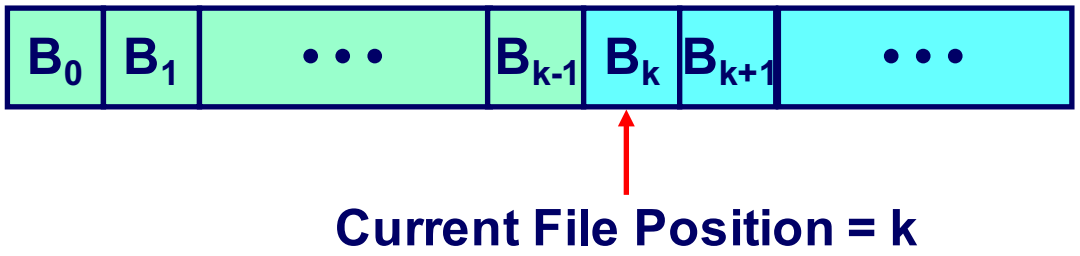
Caratteristiche principali

- Device come file → interfaccia Unix I/O
- L' I/O è gestito in maniera uniforme

Unix I/O – System Call

System call di I/O

- Apertura / chiusura di file
  - ⇨ `open()` e `close()`
- Lettura e scrittura di file
  - ⇨ `read()` e `write()`
- Cambiare la *current file position* (seek)
  - ⇨ Indica la posizione da cui leggere o su cui scrivere
  - ⇨ `lseek()`



Apertura di file

Informare il kernel che si vuole accedere ai contenuti di un file

```
int fd;    /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

Ritorna un intero, il *file descriptor*

`fd == -1` indica errore

File aperti di default

Ogni processo creato da una shell Unix inizia la propria esecuzione con tre file aperti associati a un terminale:

- **0: standard input**
- **1: standard output**
- **2: standard error**

Chiudere file

Informare il kernel che si è finito di accedere a quel file

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

Non chiudere un file già chiuso!

Morale: controllare sempre i valori di ritorno delle funzioni, anche `close()`

Leggere file 1/2

Copiare byte dalla posizione corrente nel file alla memoria e poi aggiornare la posizione nel file

```
char buf[512];
int fd;        /* file descriptor */
int nbytes;    /* numero di byte letti */

/* Aprire il file fd ...  */
/* Poi leggere fino a 512 byte dal file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

Leggere file 2/2

```
[...]
int nbytes;   /* numero di byte letti */
[...]
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

Ritorna il numero di byte letti dal file `fd` in `buf`

- `nbytes < 0` indica un errore
- *short counts* (`nbytes < sizeof(buf)`) sono possibili (e non sono errori)

Scrivere su file 1/2

Copiare byte dalla memoria alla posizione attuale nel file, e poi aggiorna la posizione attuale nel file

```
char buf[512];
int fd;         /* file descriptor */
int nbytes;     /* numero di byte letti */

/* Aprire il file fd ... */
/* Scrivere fino a 512 byte da buf al file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

Scrivere su file 2/2

```
[...]
int nbytes;   /* numero di byte scritti */

/* Aprire il file fd ... */
/* Poi scrivere fino a 512 byte da buf al file fd */
if ((nbytes = write(fd, buf, sizeof(buf)) < 0) {
   perror("write");
   exit(1);
}
```

Ritorna il numero di byte scritti da `buf` al file `fd`

- `nbytes < 0` indica errore
- Come con la read, i short count sono possibili, e non sono errori

Unix I/O – un esempio

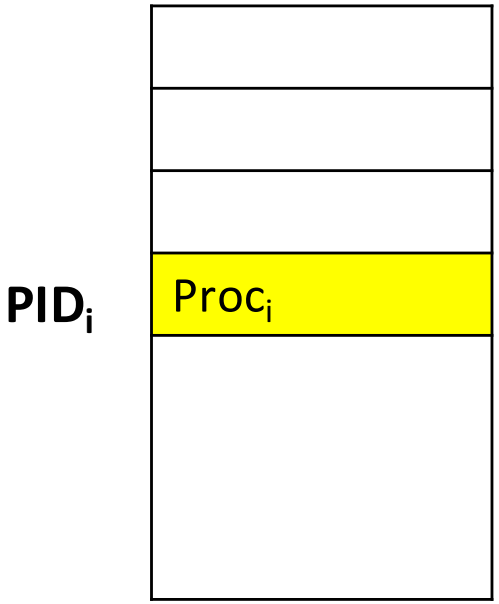
Copiare dallo stdin allo stdout, un byte alla volta

```
int main(void){
  char c;
  int len;

  while ((len = read(0 /*stdin*/, &c, 1)) == 1) {
    if (write(1 /*stdout*/, &c, 1) != 1) {
      exit(20);
    }
  }
  if (len < 0) {
    printf ("read from stdin failed");
    exit (10);
  }
  exit(0);
}
```

Process Structure

process identifier (PID)
stato del processo
puntatori a aree dati e stack
riferimento alla Text Structure
informazioni di scheduling (es: priorità, tempo di CPU, etc.)
riferimento al processo padre (PID del padre)
informazioni relative alla gestione di segnali (segnali inviati ma non ancora gestiti, maschere)
puntatori a processi successivi in code di scheduling (ad esempio, ready queue)
puntatore alla User Structure

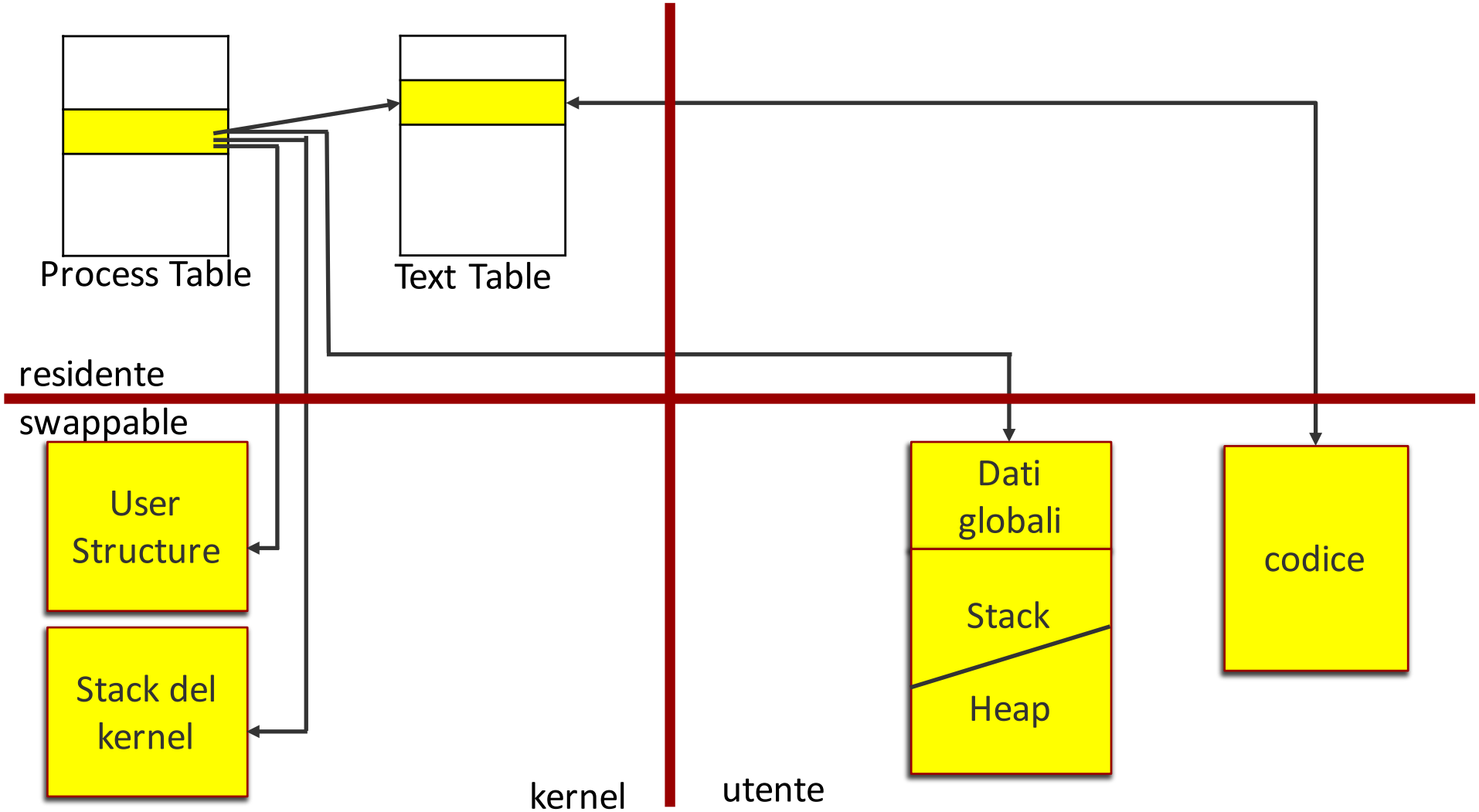
Process table



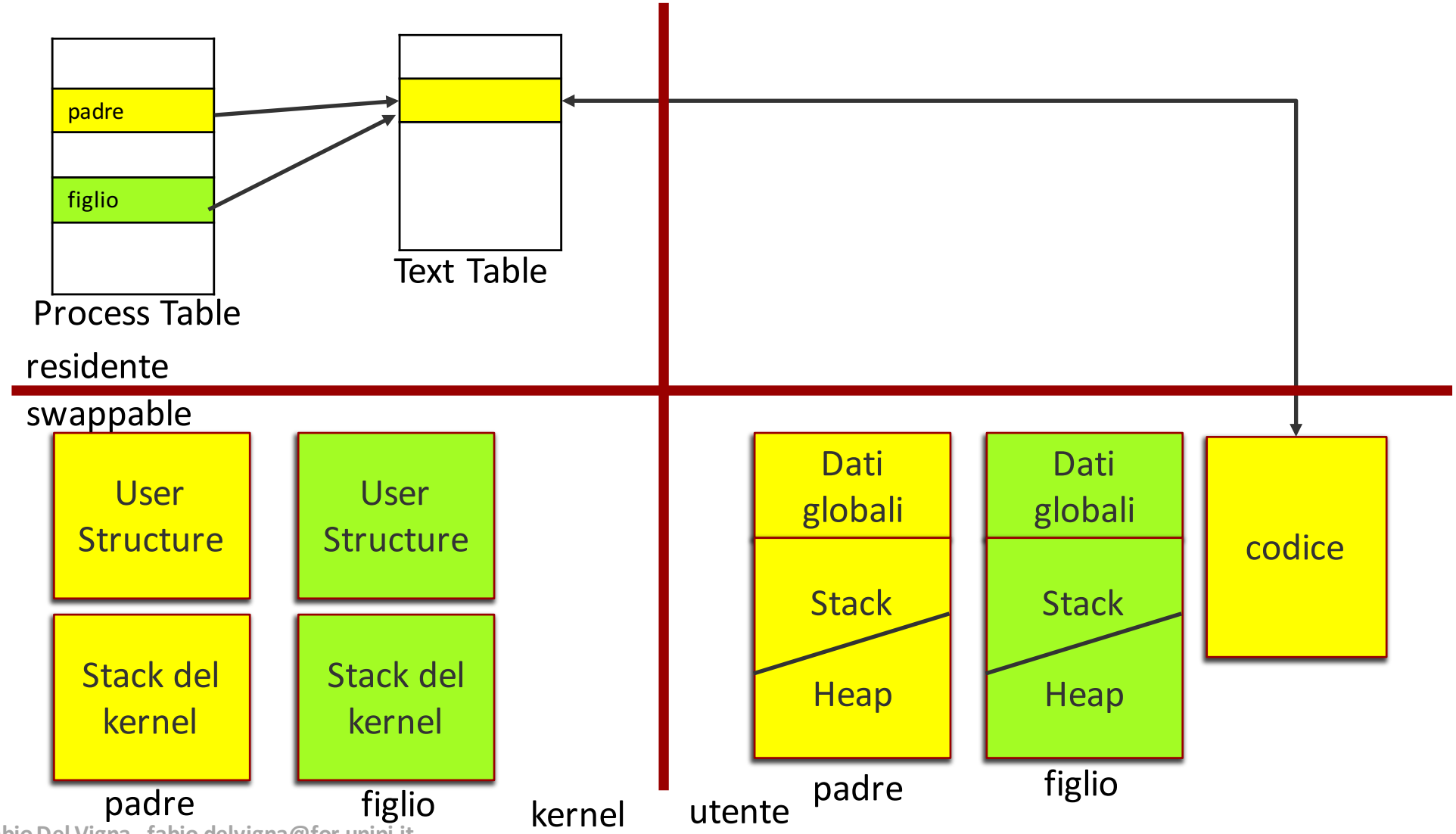
Le process structure sono contenute nella process table, che contiene un elemento per ogni processo.

Nella user structure invece sono salvate informazioni come la copia dei registri di CPU, le informazioni sulle risorse allocate (file aperti), direttorio ecc. utilizzate quando il processo non è swappato.

Immagine di un processo Unix

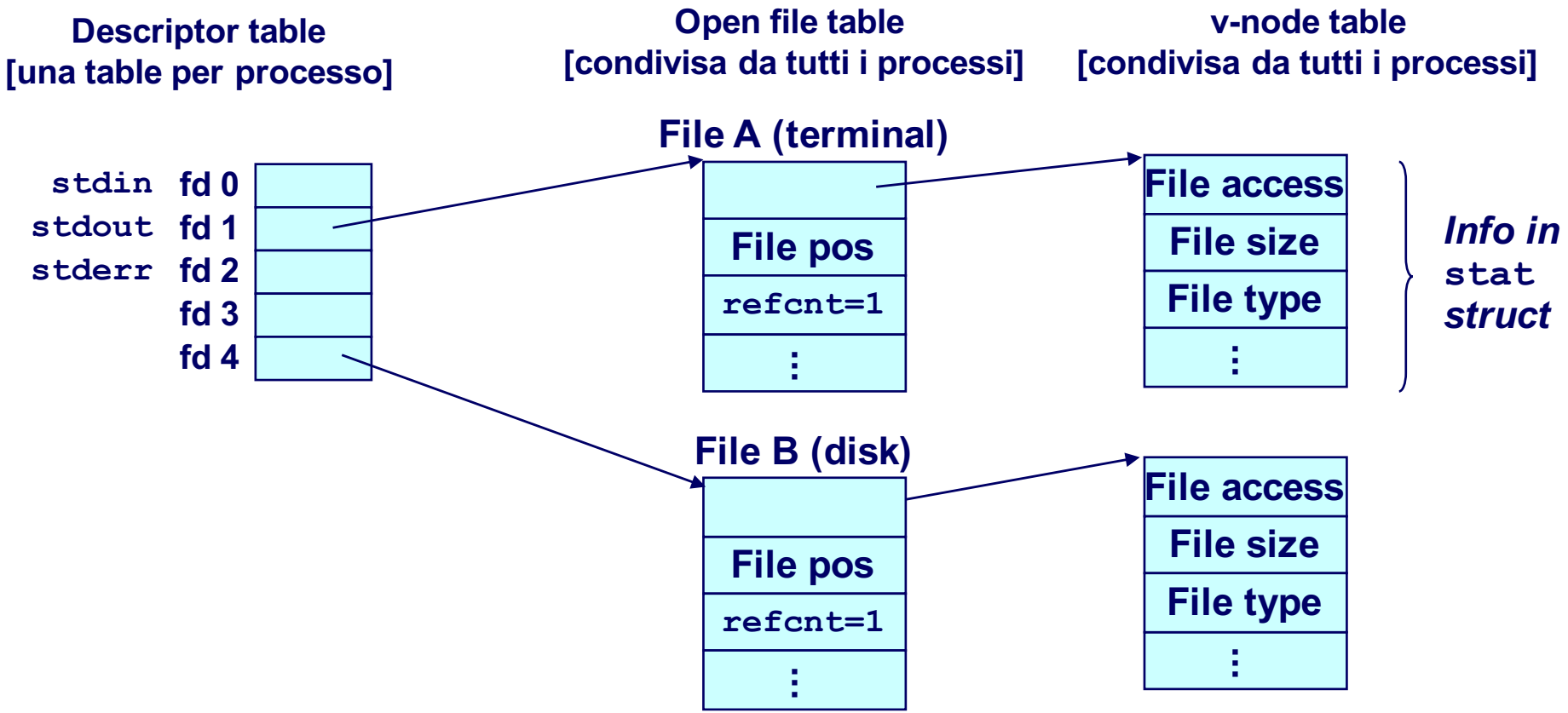


fork()


padre
figlio
Process Table
Text Table
residente
swappable
User Structure
User Structure
Stack del kernel
Stack del kernel
padre
figlio
kernel
utente
Dati globali
Stack
Heap
Dati globali
Stack
Heap
codice
padre
figlio

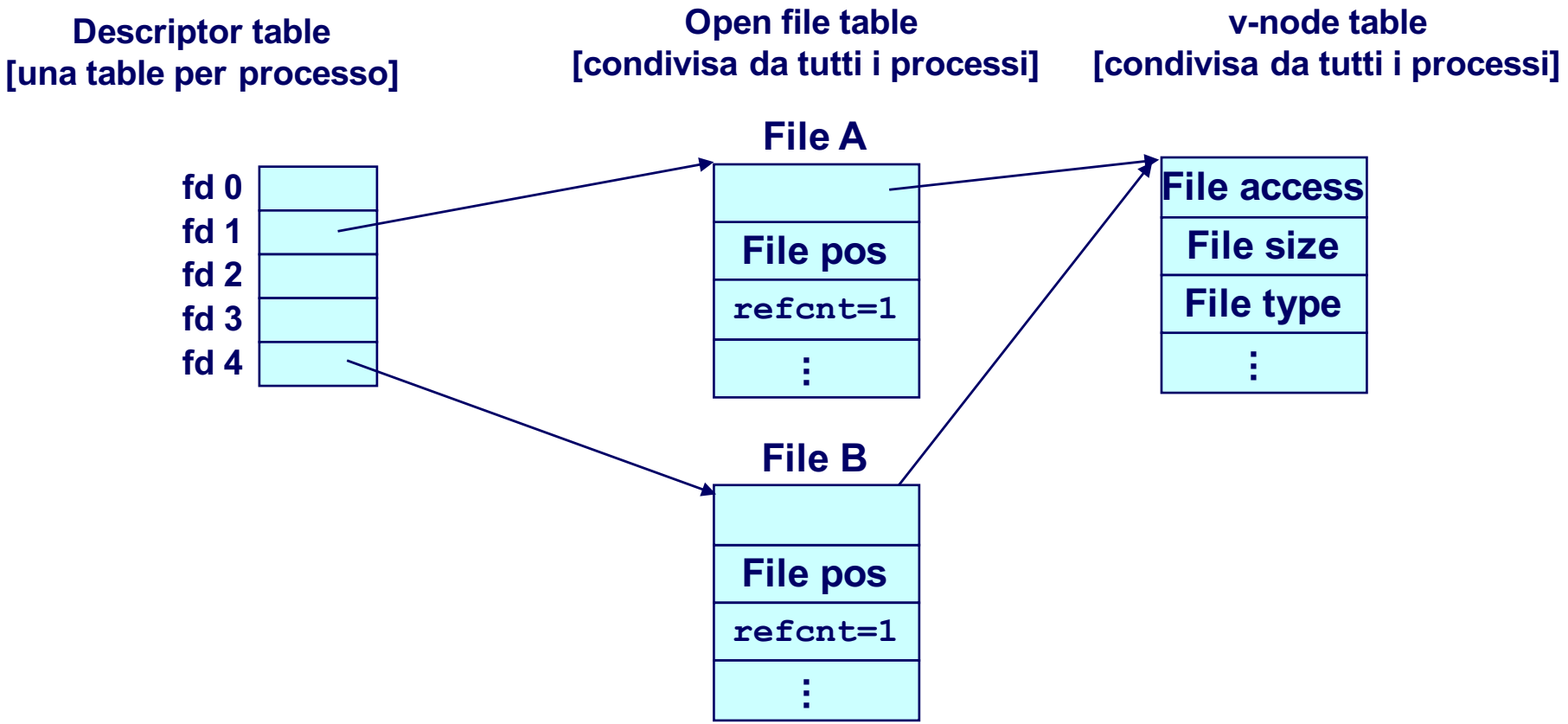
File aperti (kernel view)

Due descrittori che si riferiscono a due file su disco aperti diversi.



File Sharing (kernel view)

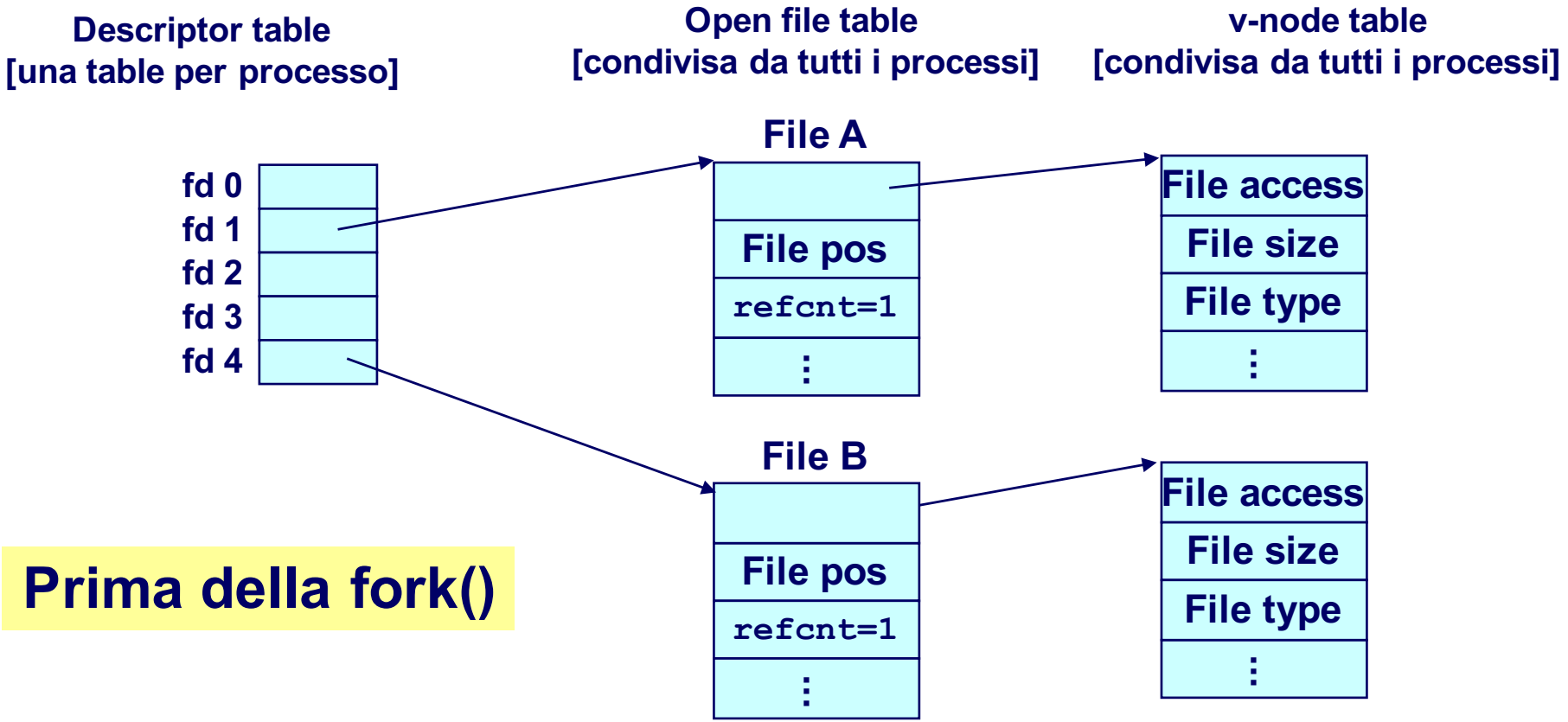
Due descrittori condividono lo stesso file su disco, mediante due file table entries distinte



Condivisione di file 1/2

Un figlio eredita i file aperti dal padre

`Nota: exec()` non cambia la situazione

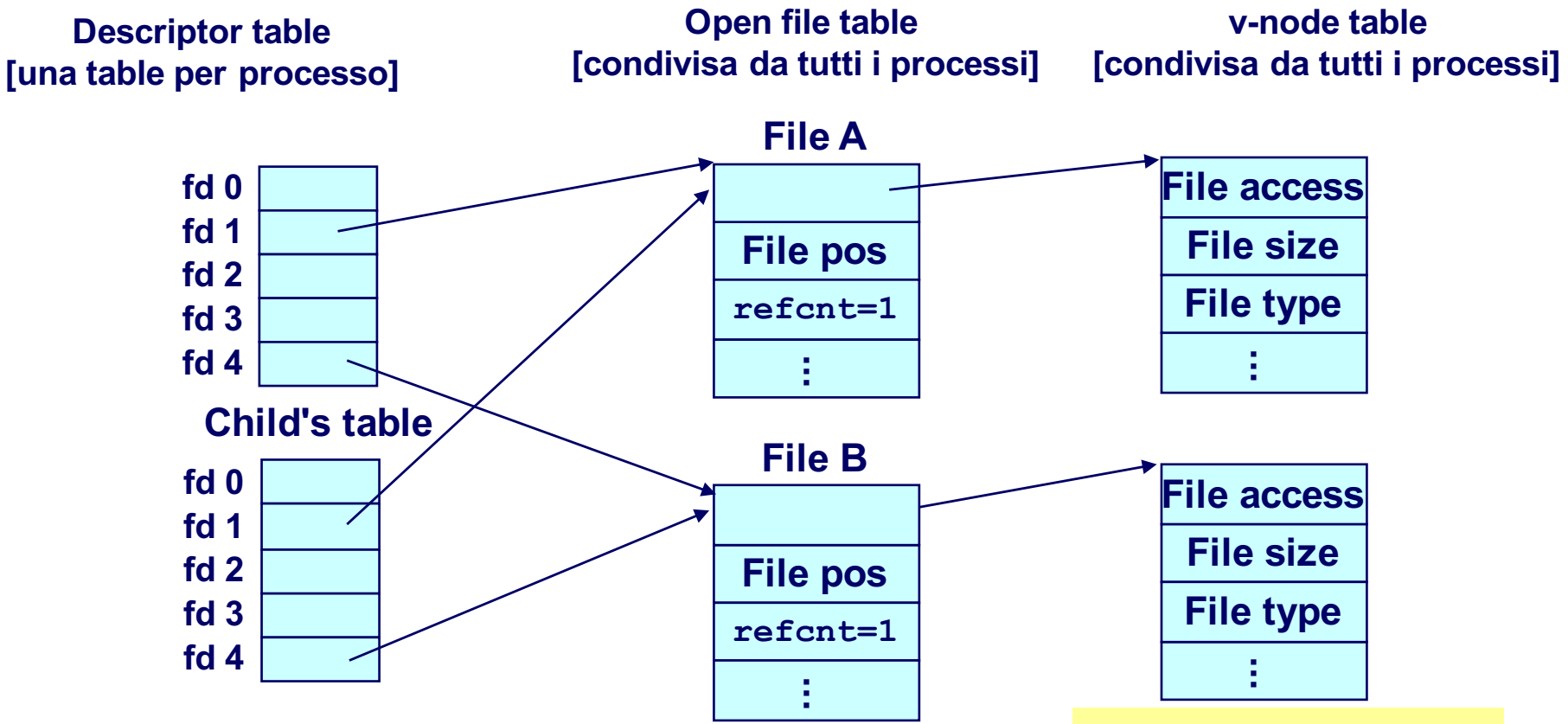


Prima della fork()

Condivisione di file 2/2 (dopo fork)

Un figlio eredita I file aperti dal padre

`Nota: exec()` non cambia la situazione



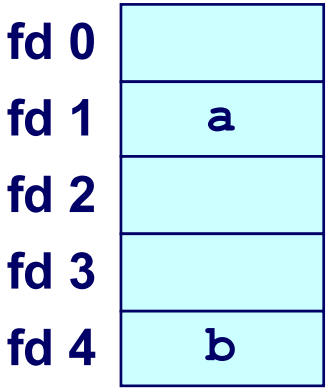
Redirezione dell'I/O

```
unix> ls > foo.txt
```

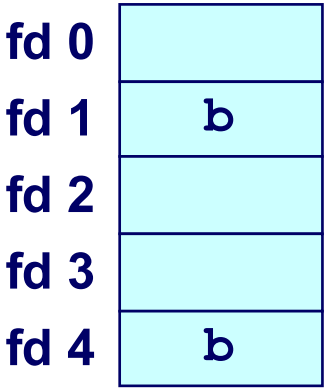
Usando `dup2(oldfd, newfd)`

Copia la entry `oldfd` alla entry `newfd`

Descriptor table before `dup2(4,1)`

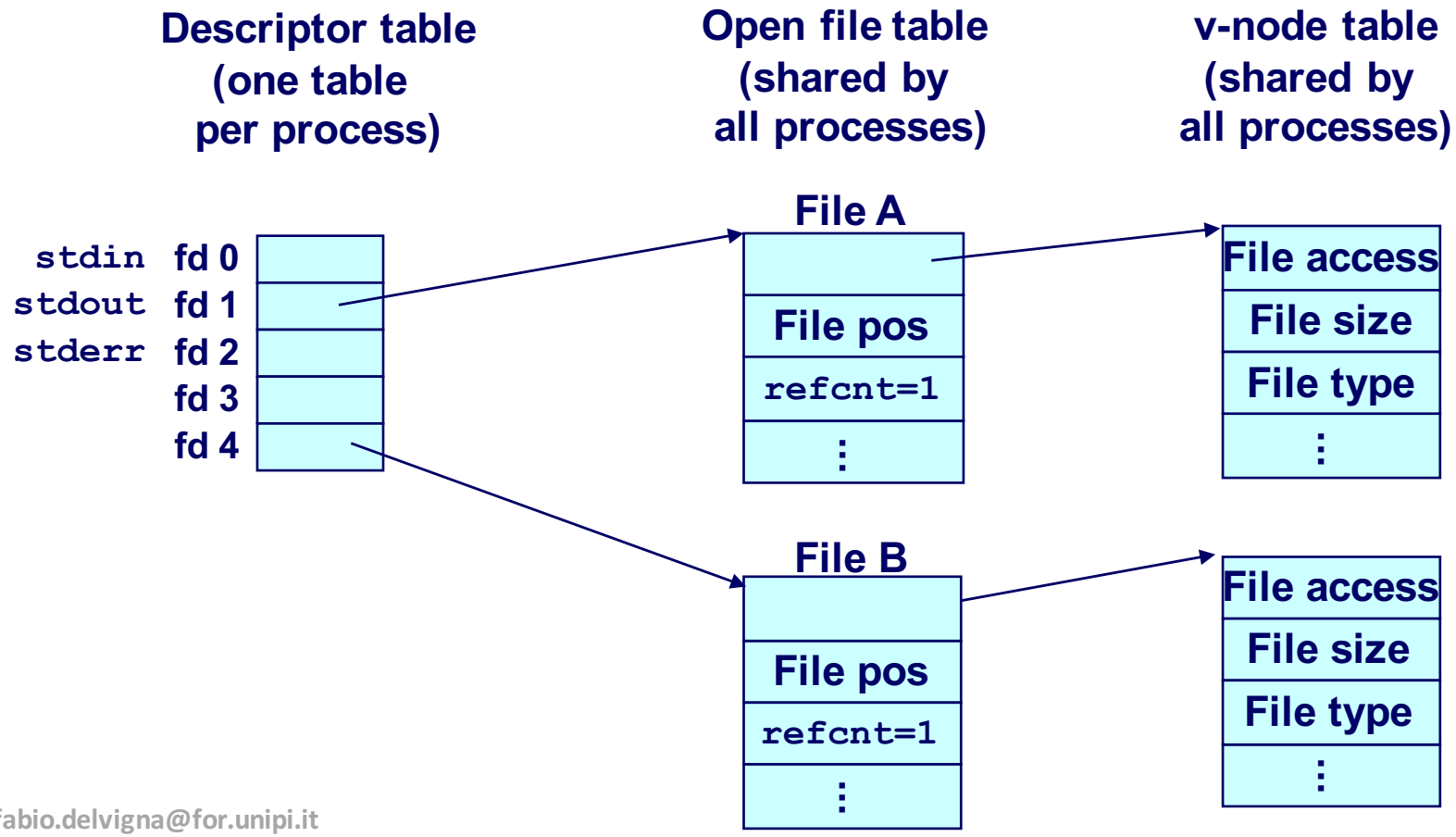


Descriptor table after `dup2(4,1)`



Redirezione dell'I/O - esempio

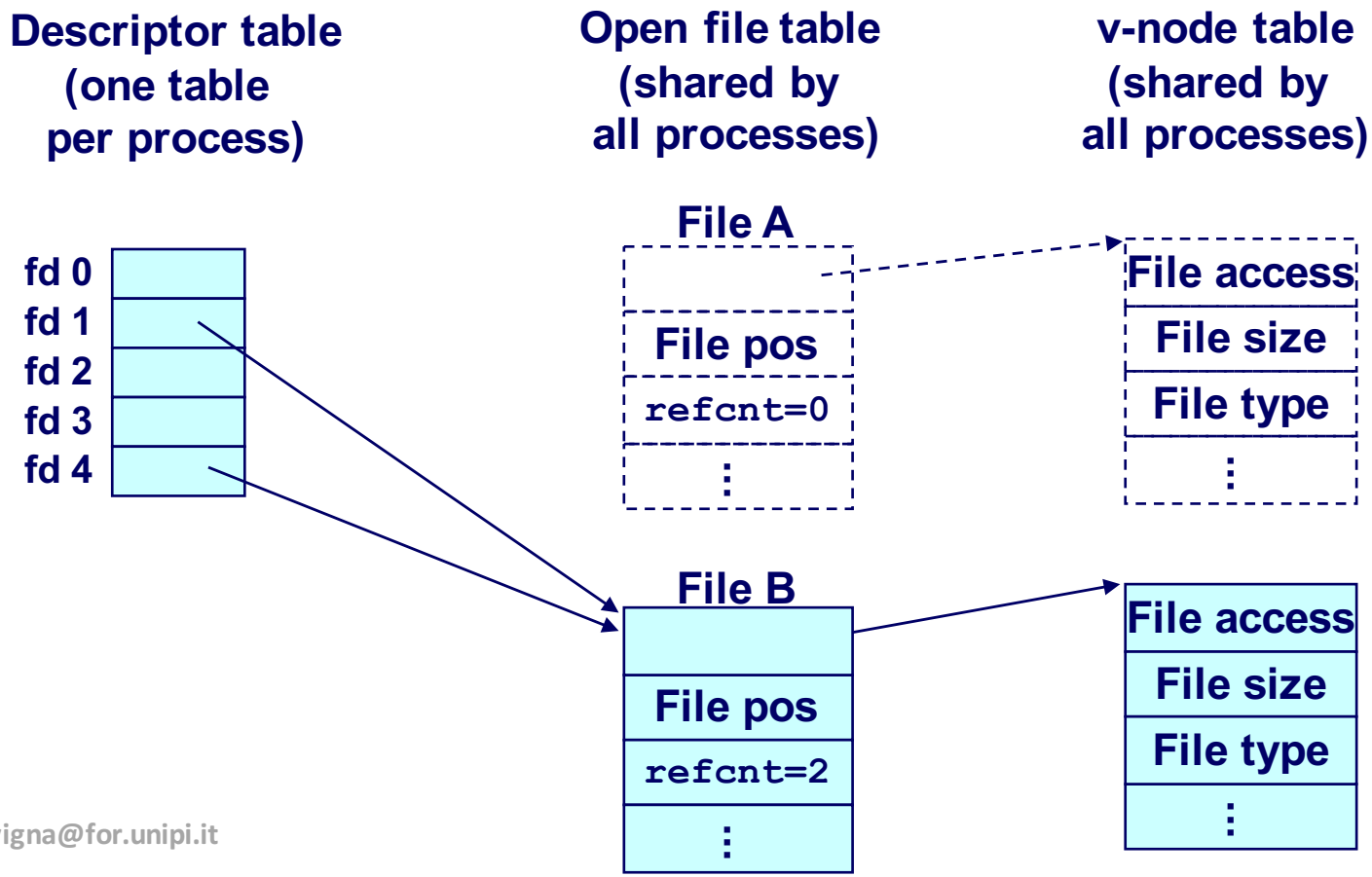
Step #1: aprire il file verso cui redirigere lo stdout



Redirezione dell'I/O – esempio

Step #2: chiamare `dup2(4,1)`

- fd=1 (stdout) si riferisce al file puntato da fd=4



Standard I/O

Standard I/O - Funzioni

La libreria C standard (`libc.a`) contiene un'insieme di funzioni standard I/O di alto livello

- Aprire / chiudere file (`fopen` and `fclose`)
- Leggere / scrivere byte (`fread` and `fwrite`)
- Leggere / scrivere righe (`fgets` and `fputs`)
- Letture / scritture *formattate* (`fscanf` and `fprintf`)

Standard I/O - Stream

Lo Standard I/O tratta i file come *streams*

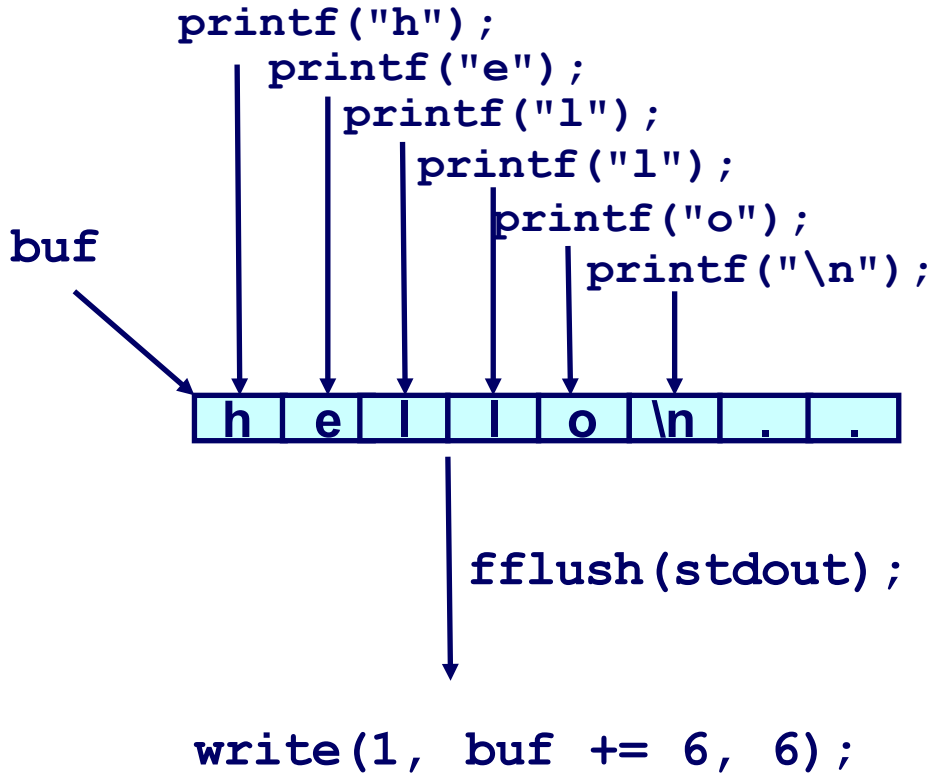
Astrazione per un descrittore di file e un buffer in memoria

I programmi C iniziano l'esecuzione con tre stream aperti (definiti in `stdio.h`)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Buffering nello Standard I/O

Standard I/O usa l'I/O buffered



Il buffer è “flushato” verso l'output fd con **“\n”** oppure **fflush()**

Buffering in azione

Si può vedere il buffering usando **strace**

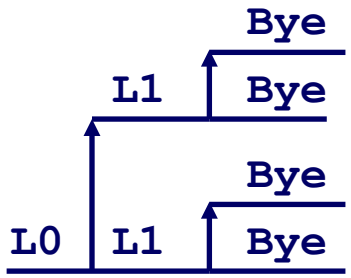
```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...)                = 6
...
_exit(0)                                 = ?
```

Fork Example 1/2

Notare che sia il padre che il figlio possono invocare fork!

```
void fork2a()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

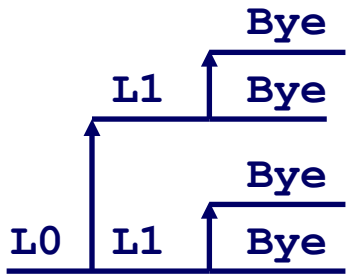


Fork Example 2/2

Rimosso “\n” dalla prima printf

- “L0” viene stampato due volte

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Altro…

Accedere alle directory

L'unica operazione raccomandabile sulle directory è la lettura del contenuto

- La struttura *dirent* contiene informazioni sulla entry della directory
- La struttura DIR contiene informazioni sulla directory, si cicla sulle sue entry

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

{
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(dir_name)))
      error("Failed to open directory");
  ...
  while (0 != (de = readdir(directory))) {
      printf("Found file: %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```

Metadata dei file

Metadata sono dati che descrivono dati

I metadati di ogni file sono mantenuti nel kernel

- l'utente vi accede tramite le funzioni `stat` e `fstat`

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

Riferimenti

The Unix bible:

- Stevens Rago, Advanced Programming in the Unix Environment 2nd edition, Addison Wesley, 2005.

Stevens è probabilmente il miglior autore tecnico di sempre.

- Ha scritto lavori di:
  - Unix programming
  - TCP/IP
  - Unix network programming
  - Unix IPC programming.

Stevens è morto il Sept 1, 1999.

Riferimenti

Parte (quasi tutto) del materiale di queste slide (la parte sull'I/O) è adattamento e traduzione del materiale di http://csapp.cs.cmu.edu/

Pipe

Pipe 1/2

- Le pipe sono degli strumenti di comunicazione tra processi, di tipo monodirezionale
  - Sono spesso utilizzate per “connettere” lo std output di un processo allo std input di un altro
  - Nella Shell sono individuate dall'operatore “|”

Pipe 2/2

- Per il linguaggio C esiste la primitiva

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

- crea una pipe e inserisce l'indice dei suoi descrittori nella struttura fd[]
- Sulla pipe si può operare con i normali operatori per file `write` e `read`.
- Ricordarsi (sempre) di chiudere il “lato” della pipe che non ci serve
  - `close(fd[0])` - input
  - `close(fd[1])` - output

Esempio: pipe e fork (1 di 2)

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1) {
                perror("fork");
                exit(1);
        }
```

Esempio: pipe e fork (2 di 2)

```
        if(childpid == 0) {
                /* Child process closes up input side of pipe */
                close(fd[0]);
                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }
        return(0);
}
```

dup() 1/2

- Primitiva

```
int dup( int oldfd );
```

- Duplica il descrittore oldfd
- Solitamente si chiude prima uno dei descrittori standard (infatti dup assegna il primo descrittore libero)

dup() 2/2

```
...

childpid = fork();


if(childpid == 0) {
     /* Close up standard input of the child */
     close(0);


     /* Duplicate the input side of pipe to stdin */

     dup(fd[0]);

     execlp("sort", "sort", NULL);

}
```

dup2() 1/2

- Primitiva

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

- Duplica il descrittore oldfd
- dup2() chiude il vecchio descrittore

dup2() 2/2

```
childpid = fork();


if(childpid == 0) {
   /* Close stdin, duplicate the
   input side of pipe to stdin */
   dup2(0, fd[0]);
   execlp("sort", "sort", NULL);
}
```

Altre funzioni sulle pipe

```
FILE *popen( char *command, char *type)

int pclose( FILE *stream )
```

file: **popen1.c**

```
popen("ls ~scottb", "r");

popen("sort > /tmp/foo", "w");
```

file: **popen2.c**

file: **popen3.c**

```
popen3 sort popen3.c

popen3 cat popen3.c

popen3 cat popen3.c | grep main
```

Fifo

Fabio Del Vigna - fabio.delvigna@for.unipi.it

Fifo

- Sono simili alle pipe (named pipe).
- Non sono gestite nel kernel, ma nel filesystem
- Hanno associato un nome (a differenza delle pipe)

- Creare una FIFO

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

```
int mknod( chat *pathname, mode_t mode, dev_t
  dev);
```

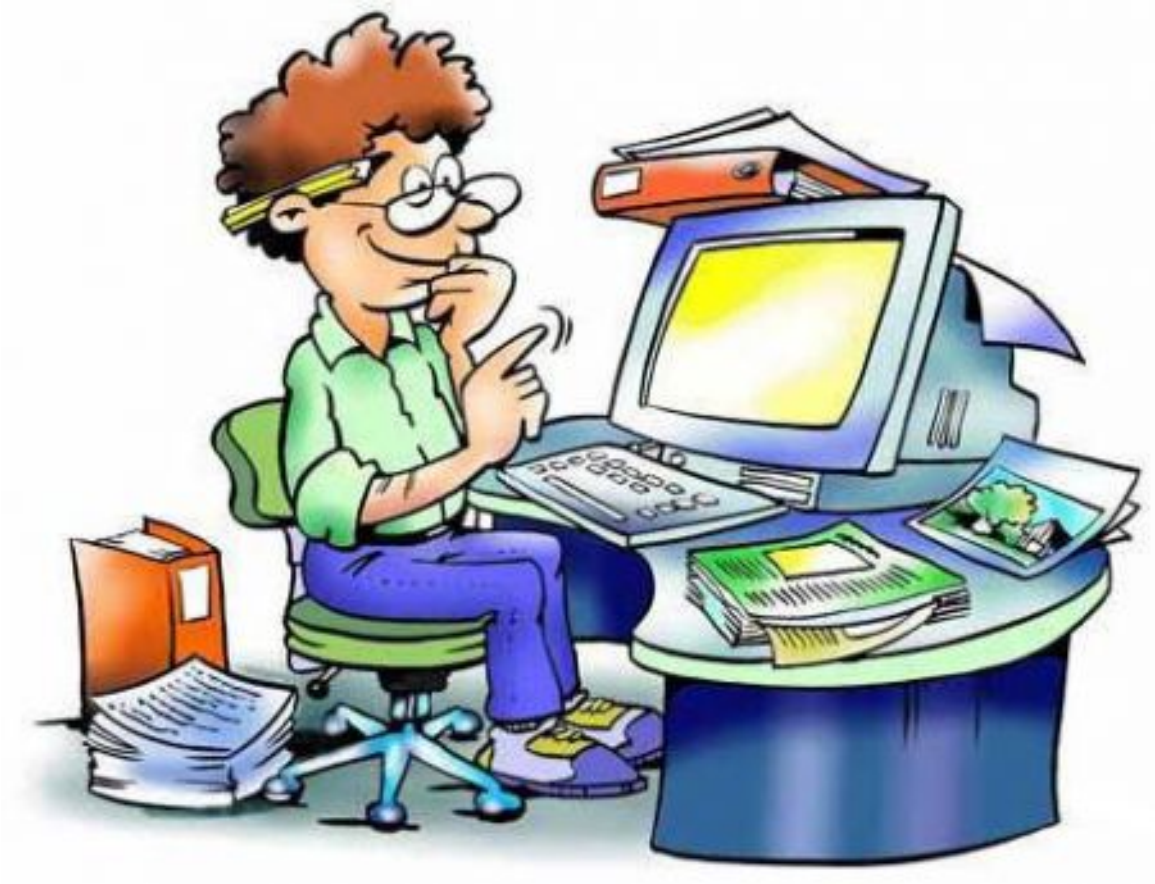
Lavorare sulle FIFO

- Una volta creata la FIFO possiamo lavorarci come su un file
- Hanno come le pipe un “senso” di utilizzo
- Vanno “aperte” con una open e chiuse con una close
- All'atto dell'apertura specificare se “r” o “w”

- `file: fifoserver.c fifoclient.c`

```
./fifoserver&
./fifoclient
```

Esercizi



Esercizi (1 / 2)

1. Eseguire il login come utente **root.**
2. Creare un utente **pippo** utilizzando il comando `adduser -s` (con home la cartella `/tmp/pippo`).
3. Creare un nuovo gruppo **usbkeyusr** a cui deve appartenere l'utente **pippo.**
4. Creare una directory `/tmp/usbkey` e montare la penna usb al suo interno. (bisogna prima sapere il nome che il sistema ha assegnato al dispositivo usb ...)
5. Provare con l'utente **pippo** a modificare il contenuto della penna usb.
6. Impostare **usbkeyusr** come group owner della directory `/tmp/usbkey` e assegnare alla cartella i seguenti diritti: accesso illimitato per i membri del gruppo, nessun tipo di accesso per gli altri.

Esercizi (2 / 2)

7. Verificare che l'utente **pippo** riesca a creare, visualizzare, cancellare file e cartelle nella penna usb.
8. Smontare la penna usb.
9. Aggiungere una nuova riga al file `/etc/fstab` per la penna usb: fare in modo che il disco sia montato in lettura/scrittura, in modo sincrono e che **non** venga montato automaticamente al **boot.**
10. Provare a montare nuovamente la penna, tramite la sintassi permessa per i mount situati in `/etc/fstab`.
11. Verificare che **pippo** riesca a fare quello che vuole sui file.
12. Eliminare l'utente **pippo.**

Esercizio 2

1. **Eseguire il login come utente root.**
2. **Giocate con** `ln, mtools, mdconfig, dd, ... altri comandi utili per lavorare sui filesystem ...`
3. **Cosa avete scoperto?**
4. **Che dubbi avete?**

Soluzione (1 / 2)

Login root

```
adduser -s [...]
```

`vi /etc/group` : **aggiungere la riga (es.):**

```
    usbkeyusr:*:1003:pippo
```

Attenzione: scegliere come ID di gruppo un valore non in uso

```
mkdir /tmp/usbkey
    mount -t msdos /dev/DEVNAME /tmp/usbkey
```

Non è permesso.

```
umount /tmp/usbkey
```

(altrimenti i comandi seguenti non hanno effetto)

```
    chown :usbkeyusr /tmp/usbkey
```

Soluzione (2 / 2)

```
    chmod g+rwx,o-rwx /tmp/usbkey
    mount -t msdos /dev/DEVNAME /tmp/usbkey
…
umount /tmp/usbkey
emacs /etc/fstab : aggiungere la riga:
    /dev/DEVNAME /tmp/usbkey msdos
    rw,noauto,sync 0 0
umount /tmp/usbkey (se necessario)
 mount /tmp/floppy
rmuser pippo
```

Soluzione esercizio 2

Non c'è soluzione ...

..., giocate, sperimentate, imparate