SISTEMI OPERATIVI 1: LABORATORIO

Materiale di riferimento

- Slide delle lezioni, codice sorgente usato, materiale addizionale
  - Pubblicato su Dir (Didattica in rete)

    www.dir.uniupo.it

- Libri di riferimento:
  - ***Advanced Programming in the UNIX Environment*** *by W. Richard Stevens*

    Stephen A. Rago Third Edition

    ISBN: 9780321637734
  - ***IL LINGUAGGIO C - PRINCIPI DI PROGRAMMAZIONE E MANUALE DI RIFERIMENTO*** by Brian W. Kernighan, Dennis M. Ritchie

    ISBN: 9788871922003
  - ***Programming with POSIX Threads*** by David R. Butenhof

    published by Addison-Wesley Professional

    ISBN: 9780201633924

Obiettivi del corso

- Familiarizzare con un S.O. (in particolare Unix/Linux) e interagire non solo tramite la GUI
  - "Navigare" nelle directory
  - Usare il sistema di help integrato – man
  - Conoscerne i suoi componenti e dove si trovano, …
- Acquisire la capacità di scrivere programmi che utilizzano le chiamate di sistema, in uno specifico S.O, comprendendo, ad un opportuno livello di astrazione che cosa sta "dietro" tali chiamate

Chiamate di sistema – Introduzione

- Sono il meccanismo con cui i programmi (comprese le interfacce grafiche con cui l'utente accede al sistema di elaborazione) richiedono servizi al sistema operativo. E' opportuno che esistano per gli stessi motivi per cui esiste il sistema operativo:
  - per l'uso delle risorse del sistema, specie i dispositivi (di I/O, memoria): nasconderne i dettagli, spesso complessi, e dipendenti dal singolo tipo di dispositivo. Ad esempio, vogliamo una funzione per "scrivere su un file" che sia indipendente dal fatto che il file sia su hard disk, memoria flash, ...
  - coordinare l'utilizzo delle risorse del sistema di elaborazione da parte di programmi contemporaneamente in corso di esecuzione

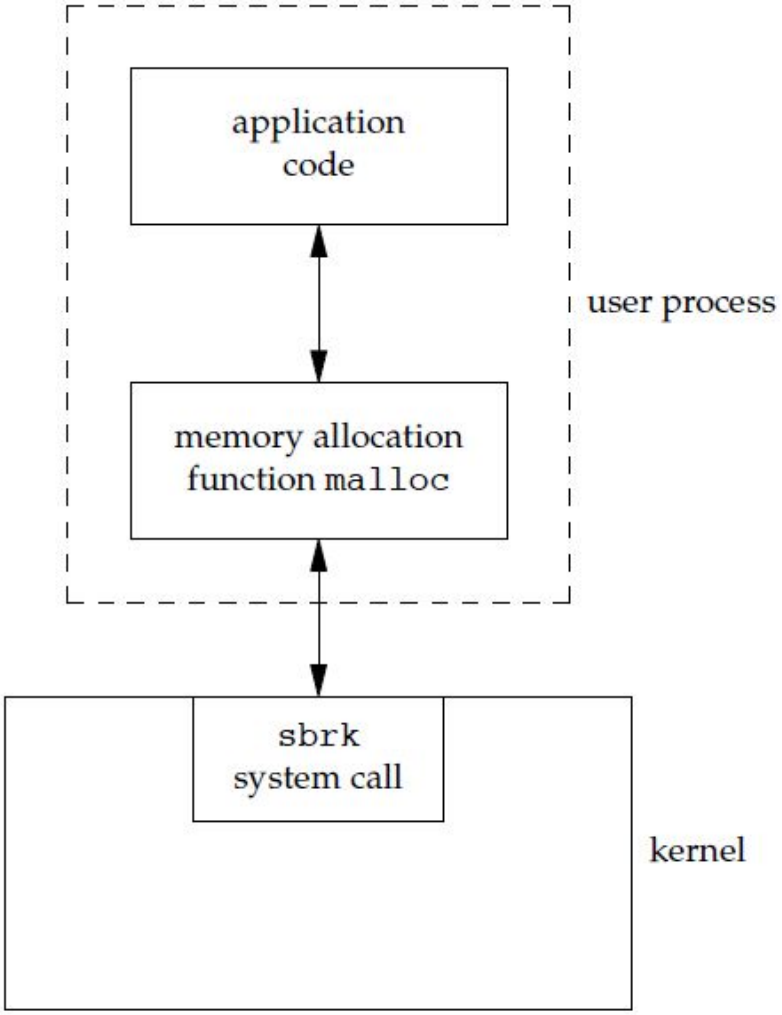
Chiamate di sistema – Introduzione

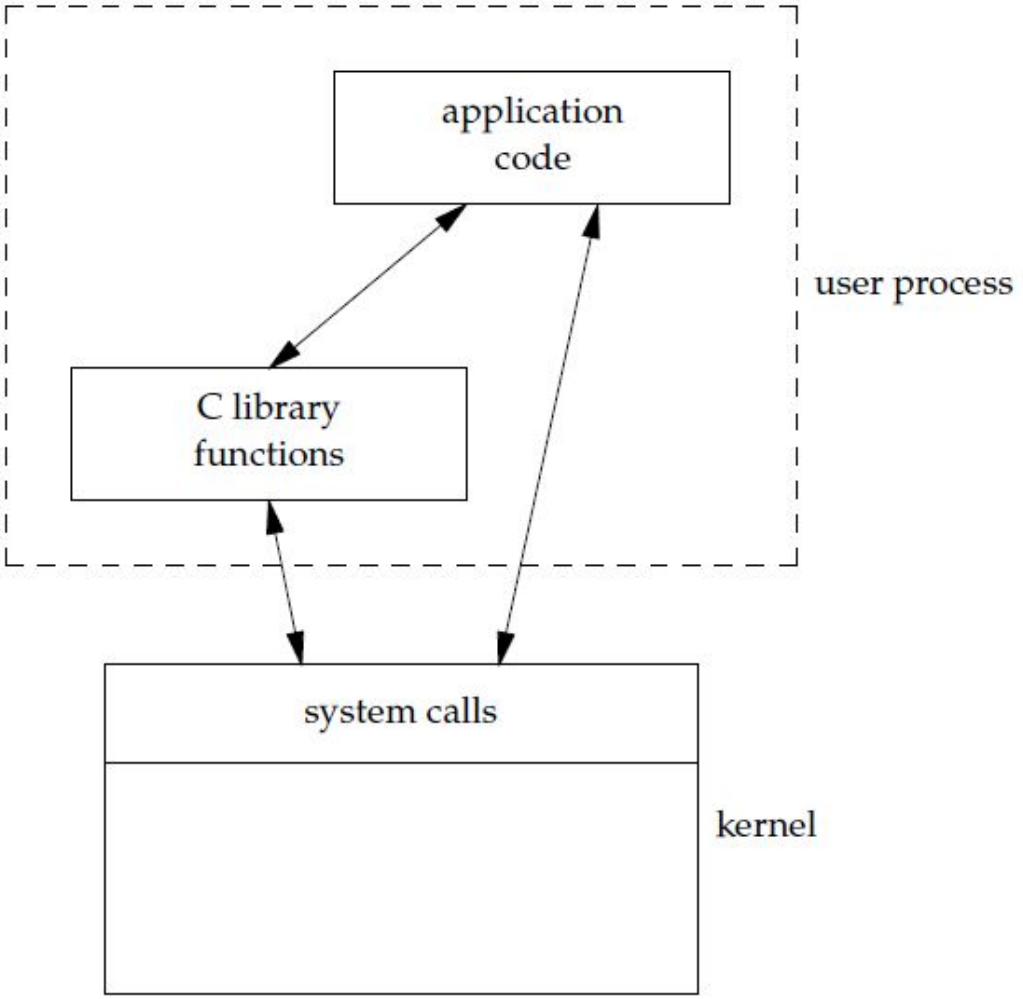
- Dal punto di vista della programmazione, in Unix le chiamate di sistema sono disponibili:
  - (1) Come istruzioni aggiuntive alle istruzioni macchina
  - (2) Come funzioni C, la cui interfaccia è definita nello standard **POSIX** (Portable Operating System - unIX) in modo che i programmi sviluppati su uno Unix girino anche su un altro.
- Il codice delle funzioni è predefinito e comprende la corrispondente chiamata di sistema di tipo (1). In qualche caso a diverse funzioni corrisponde la stessa chiamata di sistema di tipo (1).
- Noi useremo sempre le chiamate di tipo (2), cioè useremo la libreria di funzioni delle chiamate di sistema.

Chiamate di sistema – Differenze

- NB: esistono altre librerie di funzioni (insiemi di funzioni predefinite a disposizione dei programmatori) che non sono chiamate di sistema. Ad es.: librerie matematiche, o la libreria standard del C comprendente le funzioni per l'I/O (printf, getchar...)
- Ma attenzione alle differenze!
  - 1) Le chiamate di un sistema che si conforma allo standard POSIX sono offerte da tutti gli Unix, ma non necessariamente da altri S.O., mentre ad es. la printf è offerta da ogni compilatore C. Inoltre alcune librerie (proprio quelle per l'I/O) sono realizzate usando le chiamate di sistema.
  - 2) Durante l'esecuzione di una c.s. la CPU passa, con un piccolo ma a volte non trascurabile costo, a modalità kernel (nucleo del sistema operativo) nella quale si possono fare cose – tipo accedere ai dati del S.O. - che normalmente i programmi non possono fare. In questo modo a questi dati si accede soltanto usando le funzioni del sistema operativo

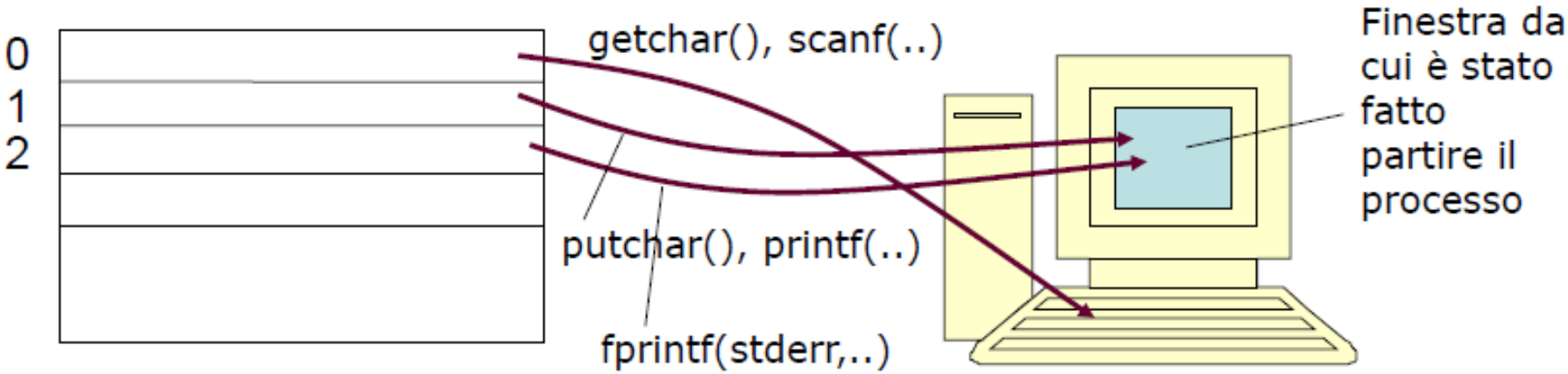
Chiamate di sistema – Differenze





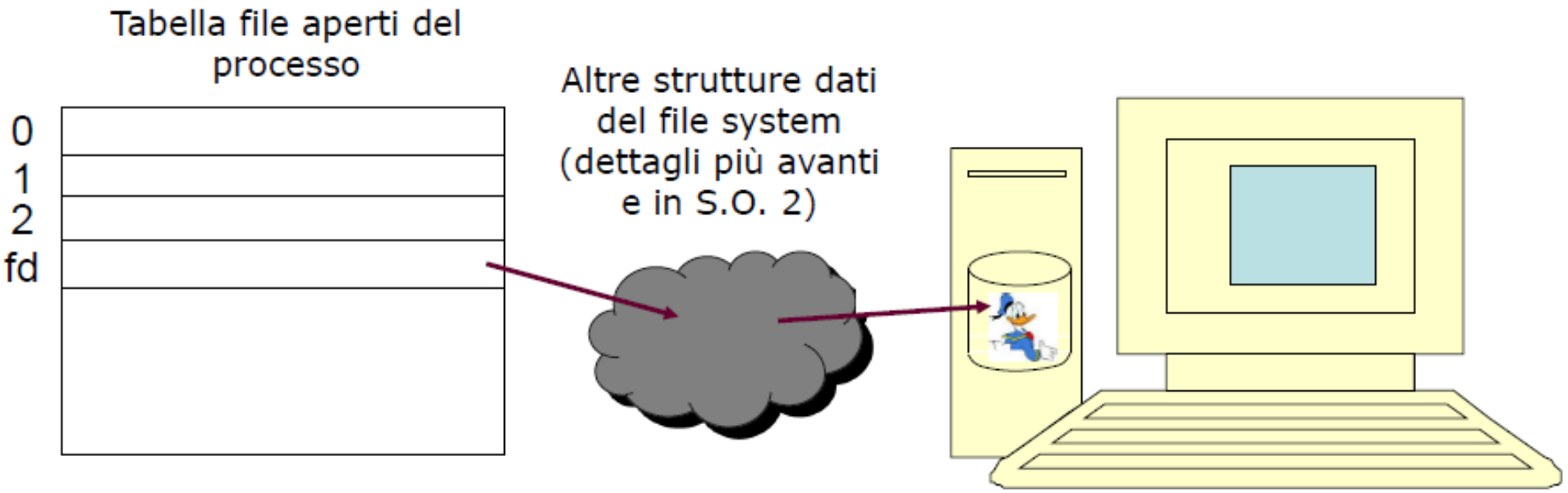
Chiamate di sistema – files

- Iniziamo con alcune chiamate di sistema relative a files
- Il significato delle operazioni richieste è relativamente facile da intuire
  - quando non strettamente necessario è molto più comodo usare la libreria standard del C per l'I/O piuttosto che le chiamate di sistema
- Unix gestisce, per ogni processo (programma in esecuzione), una tabella dei file aperti (più una complessiva per tutto il sistema) Per default, un processo ha aperti 3 file corrispondenti agli elementi 0,1,2 della tabella: standard input, standard output, standard error



Chiamate di sistema – files

- Aprire un file = chiedere al s.o. di utilizzarlo per una serie di operazioni di lettura e/o scrittura
- In C su Unix:
  - `fd = open("paperino", ...)`

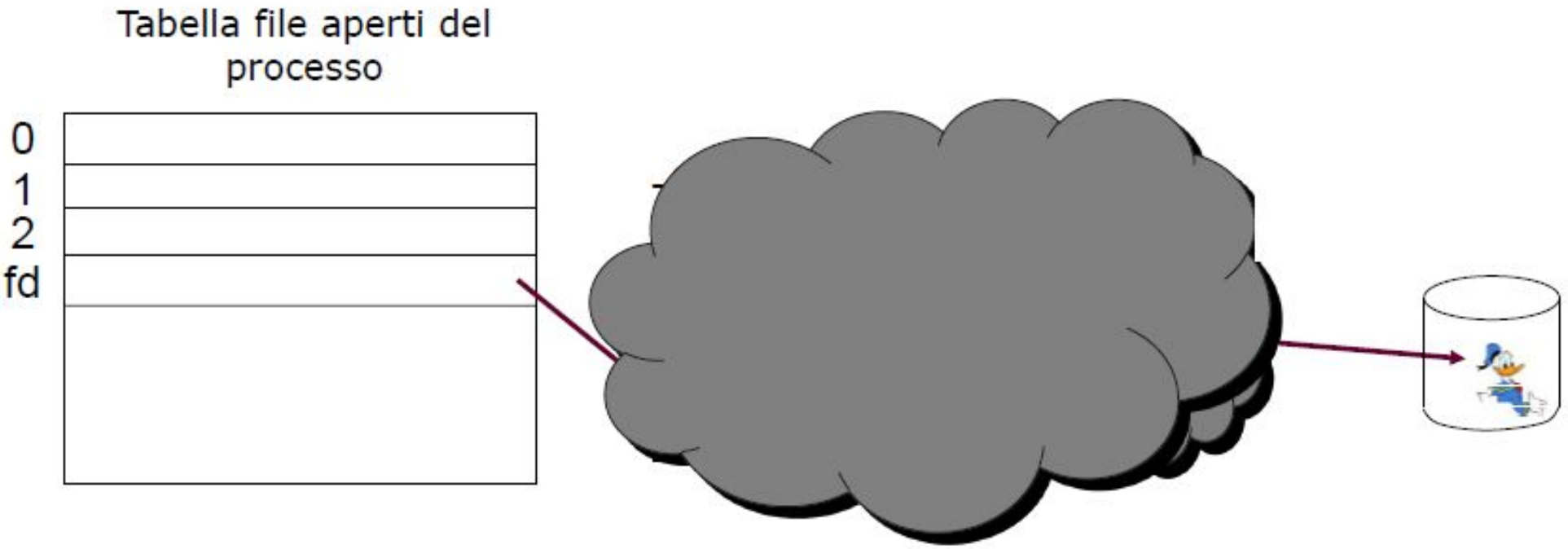


Chiamate di sistema – files

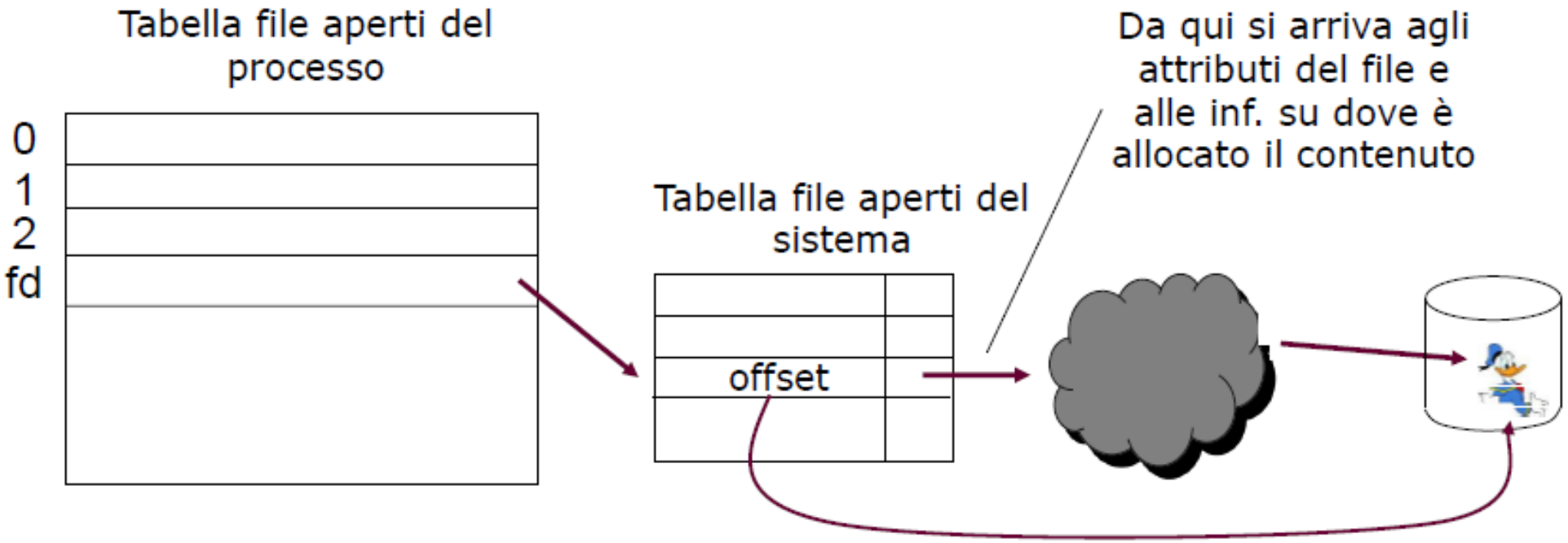
- Altre operazioni:
  - `creat("paperino",...)` crea il file (anche open lo fa con opportune opzioni)
  - `read(fd,puntatore,n)` legge n bytes dal file aperto identificato da fd, scritti in memoria a partire da puntatore
  - `write(fd,puntatore,n)` analoga

- `read` e `write` operano a partire dal punto in cui è arrivata l'ultima operazione – come fanno a sapere qual è?

- Viene mantenuto un “offset” (indirizzo relativo) che indica il numero del byte a partire dal quale avverrà la prossima operazione



- Viene mantenuto un “offset” (indirizzo relativo) che indica il numero del byte a partire dal quale avverrà la prossima operazione



Il fatto che non sia nella tabella dei file aperti del singolo processo ha la conseguenza (utile in qualche caso) di poter essere condiviso tra processi diversi

Chiamate di sistema - files

- Per le chiamate di sistema per files esistono nella libreria standard per l'I/O del C delle funzioni «corrispondenti»:

open	**fopen**
read	**getc/getchar, scanf/fscanf**
write	**putc/putchar, printf/fprintf**
close	**fclose**
lseek	**fseek**
dup, dup2	**freopen**

Unbuffered I/O: apertura file - dettagli

- *int open(const char *path, int oflag, ... /* mode_t mode */)*
  - Apre il file specificato nella stringa **path* (la specifica *const* indica che il contenuto della stringa non sarà modificato dalla funzione), restituendo in caso di successo, un intero (il file descriptor del file aperto)
  - Il file viene aperto secondo la modalità specificata tramite *oflag:*
    - *O_RDONLY* - file aperto in sola lettura
    - *O_WRONLY* – file aperto in sola scrittura
    - *O_RDWR* – file aperto in lettura e scrittura

  Uno solo dei precedenti valori devono essere passati alla open, inoltre possono essere specificate ulteriori opzioni (es. O_CREAT, O_APPEND, vedere il man)
- Qualora il file specificato da *path* non esista e si è passato O_CREAT, il file viene creato con i permessi specificati da *mode*
- La dicitura */* mode_t mode */* indica che mode è opzionale, quindi:
  - *int open(const char *path, int oflag)* – se non creo il file
  - *int open(const char *path, int oflag, mode_t mode)* – se devo crearlo

Unbuffered I/O: lettura/ scrittura su file

- *ssize_t read(int fd, void *buf, size_t nbytes)*
  - Leggo dal file con descrittore *fd* al più *nbytes* e li copio in *buf*
    - Buf deve puntare ad una zona di memoria allocata!
  - Restituisce numero di byte letti, 0 se file terminato, -1 in caso di errore
- *ssize_t write(int fd, const void *buf, size_t nbytes)*
  - Scrivo sul file con descrittore *fd* gli *nbytes* contenuti in *buf*
  - Restituisce numero di byte scritti, -1 in caso di errore
- In entrambi i casi le operazioni iniziano a partire dall'offset del file, l'offset verrà poi incrementato del numero di byte letti/scritti
- *int close(fd)*
  - Chiudo il file
    - NB: quando un processo termina, tutti i suoi file aperti sono chiusi automaticamente dal kernel
  - Restituisce 0 in caso di successo, -1 altrimenti

Chiamate di sistema – errori

- Tutte le chiamate di sistema possono dare errore per tanti motivi:
  - la richiesta non ha senso (es. le risorse su cui si chiede di operare non esistono e non è stato chiesto di crearle)
  - mancano risorse per soddisfare la richiesta, o sono stati raggiunti i limiti fissati per l'utente o per un singolo processo (programma in esecuzione)
  - In caso di errore la chiamata tipicamente (se è previsto che restituisca un intero) restituisce **il valore -1**, inoltre viene valorizzata una variabile dal nome prefissato (**`errno`** = numero di errore) che individua quale errore si è verificato tra quelli elencati nel "man" - vedremo esempi in lab
  - Cosa fare se una chiamata dà errore? Dipende dai casi, ma come minimo stampare un messaggio di errore significativo, per questo si usa la funzione *perror()*

Esercizio

- Scaricare pacchetto appunti1
- Esaminiamo insieme readerr.c
- Cosa accade se “fileprova” non è presente nella directory?
- Cercare sul man la funzione perror, cosa fa?

Argomenti sulla linea di comando

- Le funzioni si definiscono con argomenti per fare eseguire lo stesso codice su valori diversi
- Si può fare lo stesso con i programmi, es. “ls –l paperino” scrive informazioni (in forma “lunga”) sul file “paperino”, o sul contenuto della directory “paperino”
- NB “`ls`” è anch’esso un programma: qualcuno ne ha scritto il codice, e da qualche parte (in questo caso in /bin) c’è un file eseguibile con il nome “ls”
- É così per quasi tutti i comandi di Unix
- E se volessimo fare lo stesso per il programma “pippo” che scriviamo e compiliamo noi, come si fa ad accedere alle stringhe che “passiamo”, cioè scriviamo dopo il nome del comando?

pippo qui quo qua

Argomenti sulla linea di comando

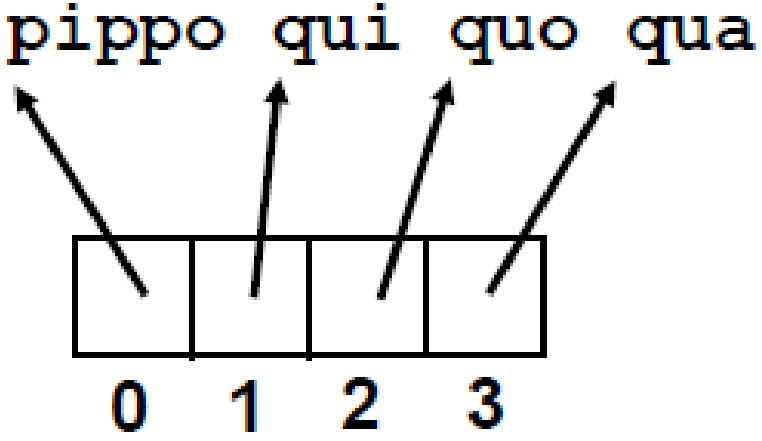
- Il main va scritto così:

argc: contatore degli argomenti
argv: vettore degli argomenti

```
main(int argc, char *argv[])
{
   /* qui argv[1], ... , argv[argc-1] sono
      le stringhe passate come argomento */
}
```

- Generando l'eseguibile con nome "pippo", e scrivendo:

- Si ha argc=4 e argv è:



Esercizio

- Modificare il codice di readerr.c affinchè il nome del file da aprire venga passato come argomento da linea di comando

Protezione dei File 1/2

- Ad ogni file e directory, il sistema associa una serie di informazioni tra cui:
  - Nome del proprietario
  - Nome del gruppo di appartenenza
  - Tipo di file
  - Dimensione
  - Permessi di accesso al file
  - Un indirizzo ai blocchi del disco su cui è memorizzato il contenuto del file
- I permessi di accesso al file sono descritti da tre triple di attributi di protezione. Ogni tripla consiste di tre flag
  - Accesso in lettura, flag “r”
  - Accesso in scrittura, flag “w”
  - Accesso in esecuzione, flag “x”

Protezione dei File 2/2

Mediante il comando “ls –l” si possono ottenere diverse informazioni sui file, tra cui le informazioni sul tipo di file e le protezioni in corrispondenza della prima colonna.

-rw-r--r--

drwxrwx--x

Il primo elemento indica il tipo di file:

- -: file normale
- d: directory
- c: file speciale a caratteri
- b: file speciale a blocchi
- ...

I rimanenti si riferiscono come dicevamo ai permessi associati al proprietario, al gruppo associato al proprietario e a tutti gli altri.

Il simbolo x, ha significato di “permesso di esecuzione” nel caso di file, di “permesso di attraversamento” nel caso di directory. Se non si ha il permesso di attraversamento, non si può in alcun modo usare un file o una sottodirectory al di sotto di essa.

Vedere su man comando **chmod** per cambiare i permessi ai file

Unbuffered I/O: apertura file - permessi

- Come si specificano i permessi? Passando a *mode* i valori della tabella sottostante
- Es: fd=*open("pippo.txt",O_RDWR | O_CREAT, S_IRUSR)*
  - *Apre e, se non esiste, crea il file pippo.txt con permesso di lettura al proprietario*
- *Come si passano più permessi? Tramite l'or bit a bit (operatore |)*
- Es: fd=*int open("pippo.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR)* - permesso di scrittura e lettura al proprietario del file

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

The nine file access permission bits, from <sys/stat.h>

Esercizio append (1)

- Nella modalità di apertura *append*, ogni volta che il file viene aperto, l'offset è spostato al termine del file
- Le successive scritture vengono quindi accodate al file
- Es da bash:
  - *echo "prova " > pippo.txt* – scrive la stringa *"prova"* nel file *pippo.txt*
  - Cosa capita la file *pippo.txt* invocando più volte lo stesso comando?
  - Provare invece:
    - *echo "prova " >> pippo.txt* – scrive la stringa *"prova"* nel file *pippo.txt* in append
  - Cosa capita invocando più volte lo stesso comando?

Esercizio append (2)

- Scrivere un programma che usando le funzioni di I/O unbuffered:
  - prende in input il nome di un file passato come argomento
  - apre o crea il file, in caso di creazione deve dare permessi di lettura e scrittura al proprietario e al gruppo
  - Il file deve essere aperto in modalità append
  - Scrive una stringa a piacere nel file
  - Chiude il file
- Verificare che il programma apra effettivamente il file in modalità append: esecuzioni successive del programma sullo stesso file devono scrivere la stringa più volte.

Chiamate di sistema – files

`close(fd)`

simmetrica di `open` – la riga `fd` della tabella diventa “libera”

`lseek(fd, …)`

sposta l’offset

`dup(fd)` e `dup2(fd, nfd)`

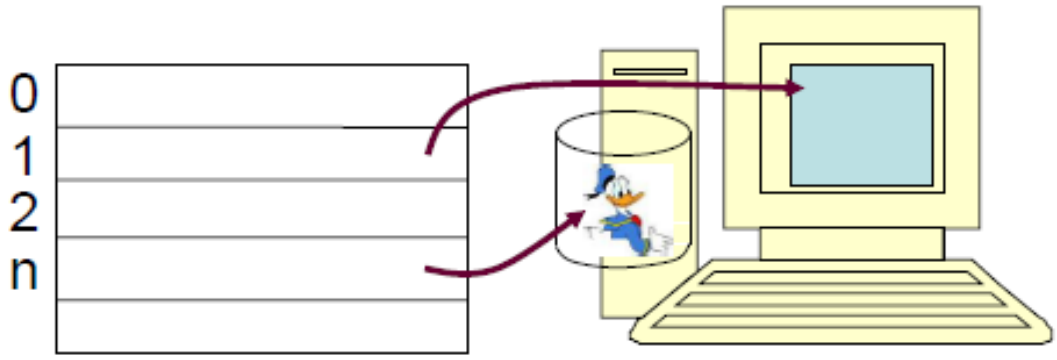
duplicano la riga `fd` della tabella dei file aperti, nella prima libera o in quella `nfd` - Ma a che serve???

serve per realizzare la “ridirezione” degli “stream” (letteralmente, “correnti”) di I/O standard (input-output-error), quello che avviene per comandi tipo:

ls > paperino

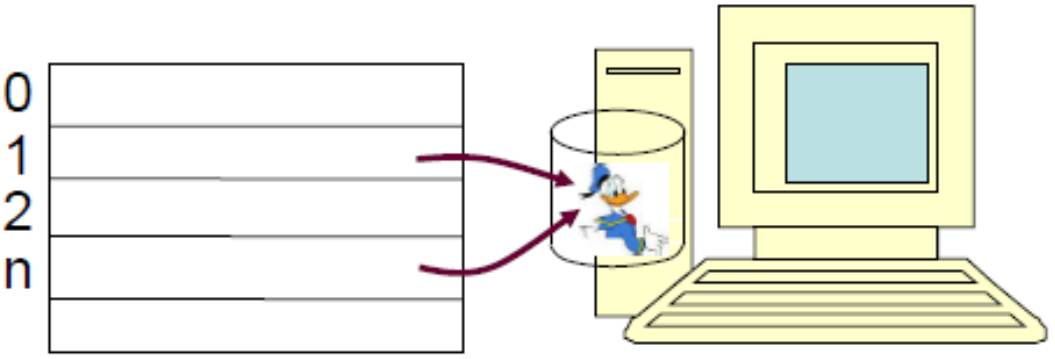
Chiamate di sistema – files

```
n=creat("paperino",..);
```

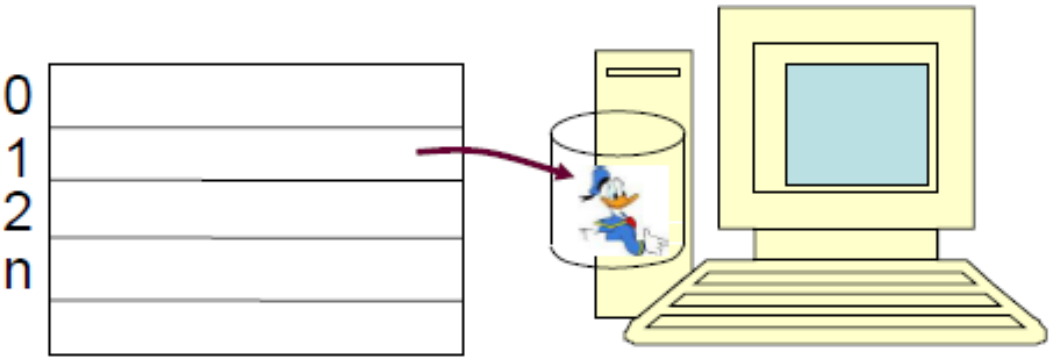



```
dup2(n,1);

/* ora le write(1,...)
e quindi le printf, putchar
vanno su "paperino" */
```




```
close(n);
```



Esercizio-ridirezione

- Scrivere un programma che:
  - ricevuto da linea di comando un nome di file
  - Apra o crei il file
  - ridirezioni lo standard output sul file aperto
  - scriva una stringa a piacere sullo standard output
  - chiuda il file
- verificare che la stringa venga scritta sul file fornito in input
- Il file redir.c in appunti1 è molto simile, ma suggerisco di esaminarlo solo DOPO aver provato a svolgere questo esercizio

Esercizio-ridirezione

- Scrivere un programma che esegua la ridirezione dell'input
  - riceve da linea di comando un nome di file opzionale
  - se il *nomefile* è presente
    - Ridireziona il suo standard input usando *nomefile* passato da linea di comando
  - in ogni caso prosegue richiedendo una stringa di input tramite **scanf** e la stampa a video terminando il programma.
  - In questo modo:
    - senza argomenti, la stringa verrà richiesta all'utente da tastiera
    - passando *nomefile*, la stringa verrà automaticamente letta dal file
  - Gestite opportunamente le possibili condizioni di errore

Chiamate di sistema - files

- Per le chiamate di sistema per files esistono nella libreria standard per l'I/O del C delle funzioni «corrispondenti»:

open	**fopen**
read	**getc/getchar, scanf/fscanf**
write	**putc/putchar, printf/fprintf**
close	**fclose**
lseek	**fseek**
dup, dup2	**freopen**

Funzioni di I/O bufferizzato

... la lettura/scrittura può essere «bufferizzata» parcheggiando dati in un array:

- le funzioni per la lettura (**`getchar,scanf..`**) chiamano una volta **`read`** mettendo nel buffer più dati di quelli che servono subito, e alle chiamate successive prendono i dati dal buffer
- quelle per la scrittura (**`putchar, printf..`**) mettono i dati nel buffer, chiamano **`write`** solo in alcuni casi:
  - il buffer è pieno
  - c'è un «\n» e la bufferizzazione è «a righe», es. si sta scrivendo su una finestra «terminale»)
  - il file viene chiuso, o il processo termina
  - viene chiamata una apposita funzione per svuotare il buffer:

    `int fflush(FILE *stream)`

Vantaggio: si fanno meno chiamate di sistema (che comportano il passaggio a modo kernel – sarebbe inefficiente farlo per ogni lettura/scrittura di 1 o pochi bytes)

Esercizio: effetto bufferizzazione

```
::::::::::::::
loopsc.c
::::::::::::::
#include <stdio.h>

main(int argc, char *argv[])
{
    int n=1000,i;

    if (argc==2) n=atoi(argv[1]);

    for(i=0;i<n;i++)
        write(1,"Q",1);
}
```

```
::::::::::::::
looplf.c
::::::::::::::
#include <stdio.h>

main(int argc, char *argv[])
{
    int n=1000,i;

    if (argc==2) n=atoi(argv[1]);

    for(i=0;i<n;i++)
        putchar('q');
}
```

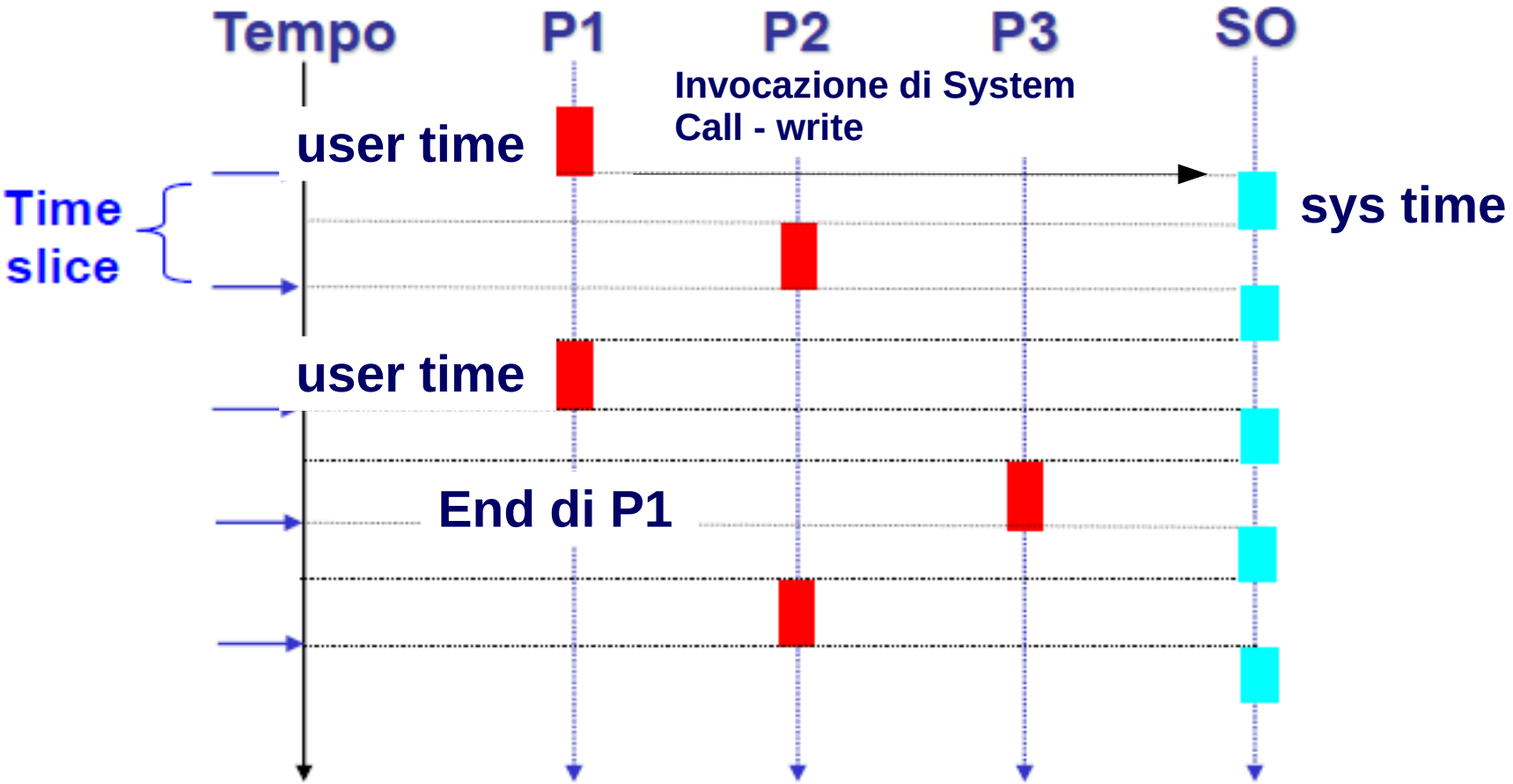
- Quali differenze si notano nei due programmi ?
- Dopo aver generato gli eseguibili `loopsc` e `looplf`, chiamare ad esempio:

```
time ./looplf 1000 > /dev/null

time ./loopsc 1000 > /dev/null
```

- proseguendo con valori più grandi dell'argomento (10000, 100000, 1000000,...) si noterà una differenza sempre più significativa tra i tempi di esecuzione delle due versioni.

Esercizio: time



Esercizio: effetto bufferizzazione(2)

```
::::::::::::::
hellolf.c
::::::::::::::

#include <stdio.h>


int main(int argc, char *argv[])
{
   printf("Hello \n");
   for (;;);
}
```

```
::::::::::::::
hellosc.c
::::::::::::::

#include <unistd.h>


int main(int argc, char *argv[])
{
   write(1,"Hello \n",7);
   for (;;);
}
```

- Essi sembrano equivalenti: entrambi scrivono e poi devono essere interrotti.
- Se però si ridirige l'output dei due programmi su un file, oppure non si stampa il \n, sono ancora equivalenti?
- ESERCIZIO: usare fflush per rendere il loro comportamento analogo anche in questi casi.