

UNIVERSITÀ DEL PIEMONTE ORIENTALE

# Sistemi Operativi 1

## Laboratorio 1

**Prof. Luisa Barrera León**

Fonte: Prof. Davide Cerotti

# Obiettivi del Lab

---

- 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.
- Sistema (sono dei leggeri differenze):
  - Distribuzione nativa sulla macchina
  - Macchine Virtuale (è pensante per la CPU)
  - Mac (funzioni leggermente diverse, soprattutto per funzioni di sincronizzazione)
  - Repl.it (si può creare una macchina virtuale in Linux)

# Part I

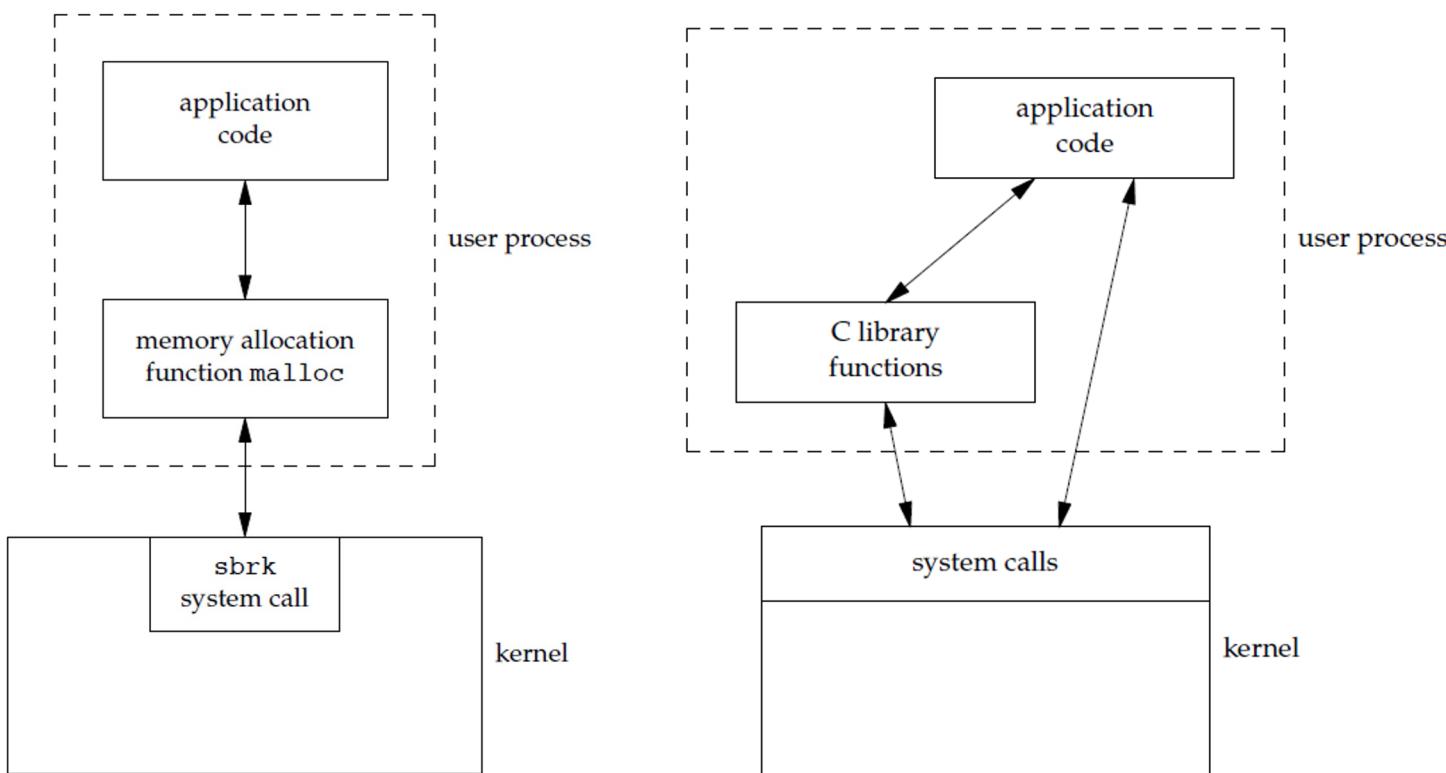
---

# 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 (gestire ed usare le risorse).
- 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.

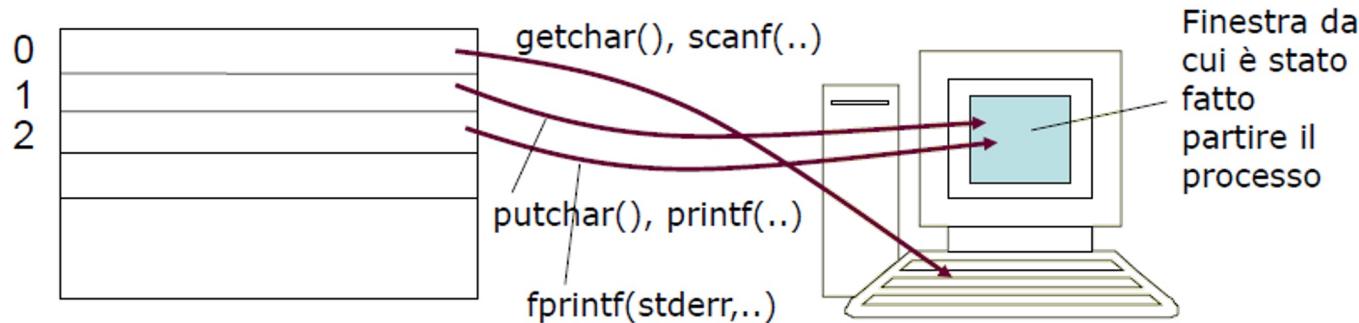
NB: *esistono altre librerie di funzioni (insiemi di funzioni predefinite a disposizione dei programmatore) che non sono chiamate di sistema. Ad es.: librerie matematiche, o la libreria standard del C comprendente le funzioni per l'I/O (printf, getchar, ...).*

# Chiamate di sistema – Differenze



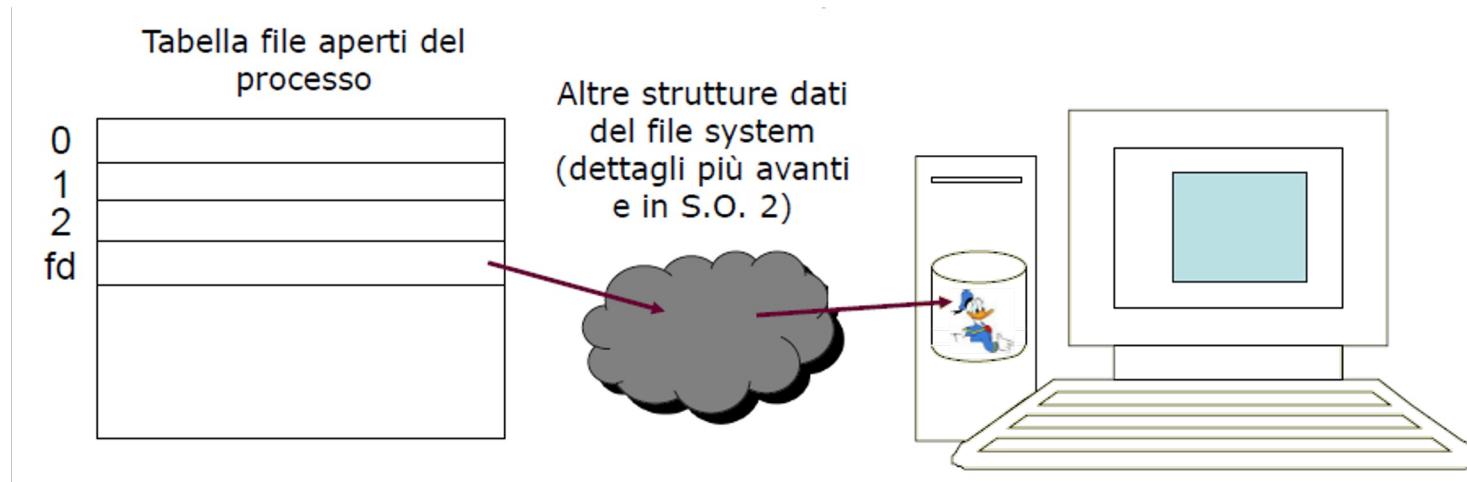
# Chiamate di sistema – files

- Iniziamo con alcune chiamate di sistema relative a files (`open`, `read`, `write`)
- 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", ...)`

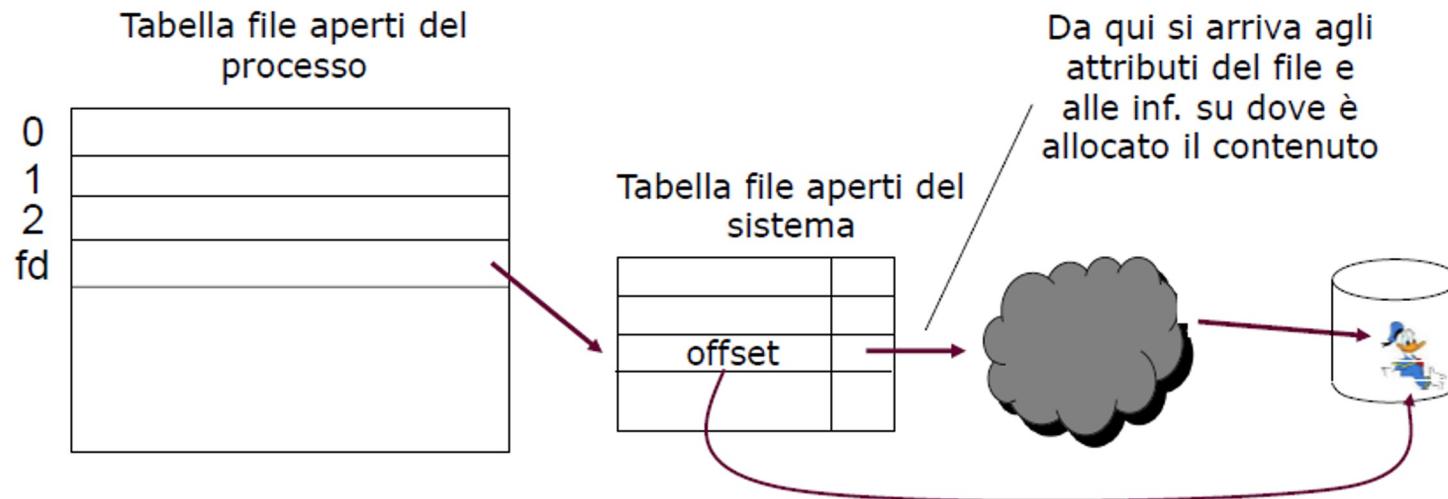


# Chiamate di sistema – files

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

# Chiamate di sistema – files

- 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»:

## Baso Livello

`open`  
`read`  
`write`  
`close`  
`lseek`  
`dup, dup2`

## Analoghi in C (offrono più servizi)

`fopen`  
`getc/getchar, scanf/fscanf`  
`putc/putchar, printf/fprintf`  
`fclose`  
`fseek`  
`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 (liberare la *entri* nella tabella del file aperto del processo)
    - 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()**.

# Al terminale...

- Comandi:
  - **man**: Il comando man in Linux è un comando che consente di visualizzare la pagina di manuale di un comando, di una funzione o di un altro argomento.
  - **man open senza**
  - **man -S2 open** (senza la S funziona anche)

```
XDG-OPEN(1)           xdg-open Manual           XDG-OPEN(1)

NAME
    xdg-open - opens a file or URL in the user's preferred application

SYNOPSIS
    xdg-open {file | URL}

    xdg-open {--help | --manual | --version}

DESCRIPTION
    xdg-open opens a file or URL in the user's preferred application. If a
    URL is provided the URL will be opened in the user's preferred web
    browser. If a file is provided the file will be opened in the
    preferred application for files of that type. xdg-open supports file,
    ftp, http and https URLs.
```

NB. Se non funziona il comando man  
eseguire questi comandi:

1. sudo apt-get update
2. Sudo apt-get install gcc

# 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?

Compilare/Eseguire C file:

```
$ gcc readerr.c -o readerr  
$ ./readerr
```

# 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 simbolo x, ha significato di “permesso di esecuzione” nel caso di file, di “permesso di attraversamento” nel caso di directory.*

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.

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>

# Spiegazione uso maschere di bit

Come sono rappresentati i valori O\_RDWR, O\_CREAT, S\_IRUSR, S\_IWUSR... e perchè si mettono in or?

- Usiamo maschere di bit, i permessi correnti sono rappresentati da una sequenza di bit, ogni opzione è rappresentata dal valore del bit ad una determinata posizione, prima, seconda, ...
- Es `001010000` indica che la terza e quinta opzione sono abilitate. Come faccio ad abilitarne un'altra?
- Uso maschera opportuna e metto in or bit-a-bit (operatore | in c)

	rwxrwxrwx
Permessi attuali	010010000 or
S_IRUSR	100000000
	-----
Permessi nuovi	110010000

	rwxrwxrwx
Permessi attuali	001010000 or
S_IRUSR S_IWUSR	110000000
	-----
Permessi nuovi	111010000

# 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
- È 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.

# Part II

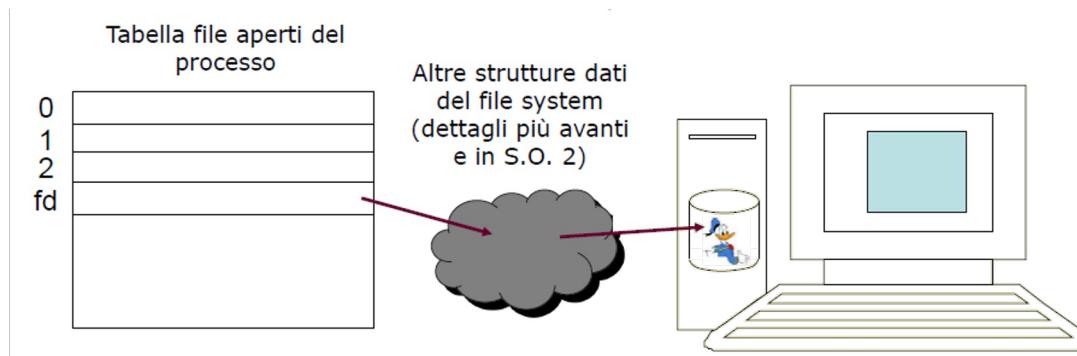
---

# Chiamate di sistema – files

`close(fd)` → Simmetrica di `open` – la riga `fd` della tabella diventa “libera”

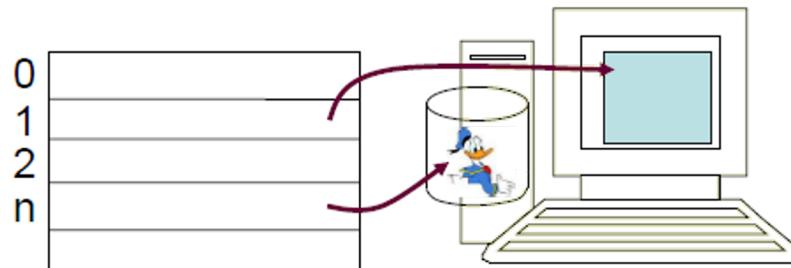
`lseek(fd, ...)` → Sposta l'offset

`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???



# Chiamate di Sistema – files

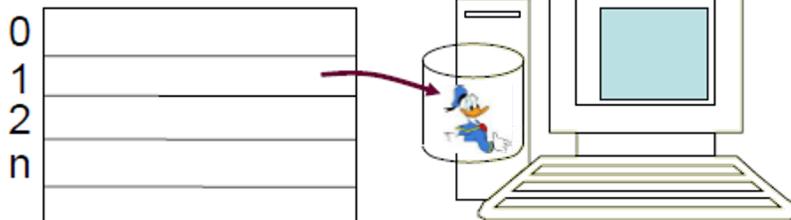
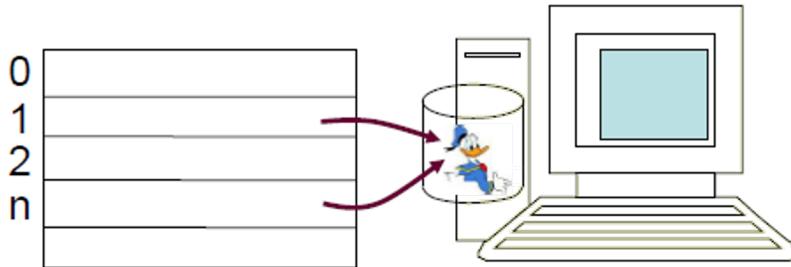
```
n=creat("paperino",...);
```



```
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»:

## Baso Livello

`open`  
`read`  
`write`  
`close`  
`lseek`  
`dup, dup2`

## Analoghi in C (offrono più servizi)

`fopen`  
`getc/getchar, scanf/fscanf`  
`putc/putchar, printf/fprintf`  
`fclose`  
`fseek`  
`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)

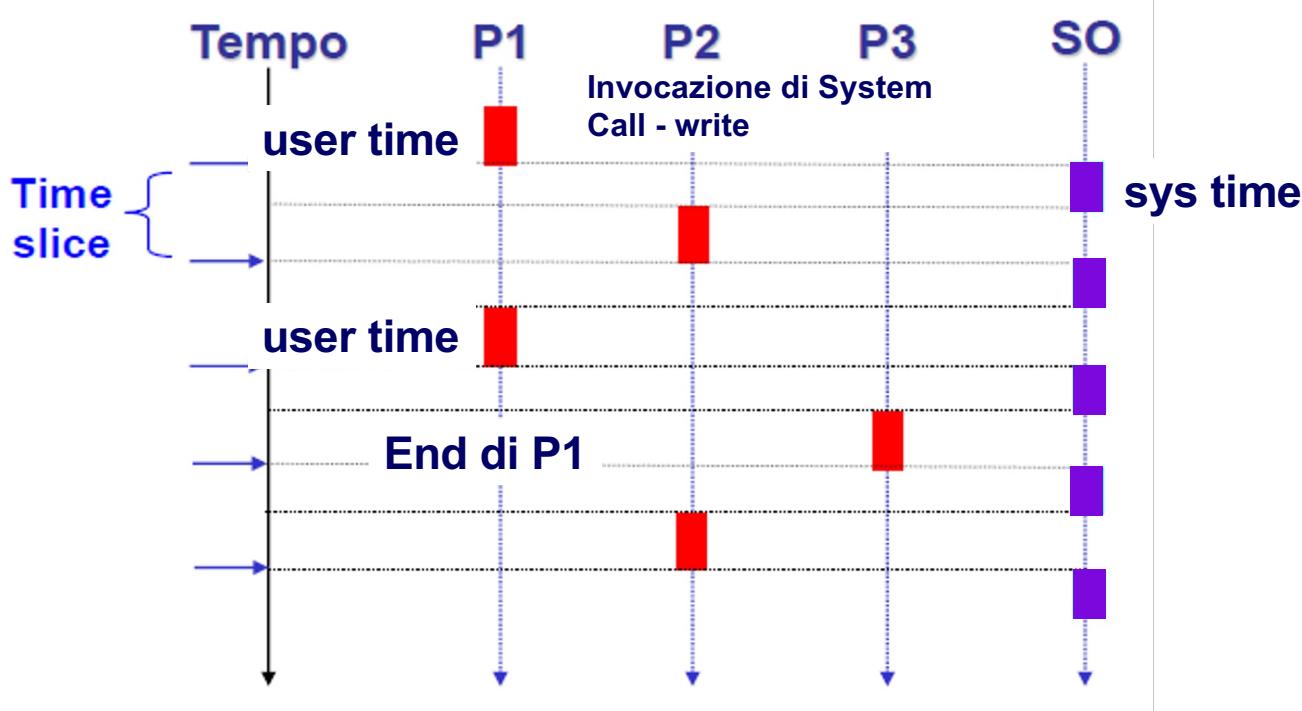
# Esercizio: effetto bufferizzazione

```
:::::::  
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)

```
:::::::::::  
hellof.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.

# 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”.
- È 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

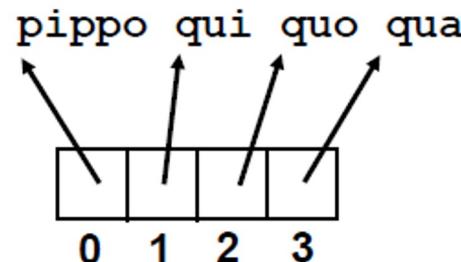
- Il main va scritto così:

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

argc: contatore degli argomenti  
argv: vettore degli argomenti

- 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.



UNIVERSITÀ DEL PIEMONTE ORIENTALE

# Sistemi Operativi 1

## Laboratorio 2

**Prof. Luisa Barrera León**

# Parte I

---

# Puntatori Generici

---

# Puntatori generici

- **Puntatore generico (a void)**: puntatore che punta a variabile di tipo generico, imprecisato. Si dichiara come:  
void \*p;
- Ogni puntatore può essere forzato a puntare a void tramite l'operazione di cast (void \*)
- Prima di **deferenziare** un puntatore generico bisogna fare un cast al tipo originario della variabile a cui punta:

```
int a, b;  
void *p;  
  
...  
p = (void *) &a;           // p punta alla variabile intera a  
b = *((int *) p)+1;       // eseguo cast prima  
                          // di deferenziare  
...
```

# Puntatori a funzione

- Un puntatore ad una variabile è l'indirizzo della prima cella di memoria dove si trova il valore della variabile
- Il tipo della variabile (int, char, char \*, struct...) determina il numero di byte/word necessari per memorizzarne il suo valore

```
int a;  
long b;  
struct {...} c;  
void *p, *q, *r;
```



- fino a quando non deferenzio (valuto valore della variabile), non è necessario conoscerne il tipo (la dimensione)
- A cosa serve questa caratteristica fornita dal linguaggio c?

# Puntatori a funzione

---

- A sviluppare funzioni generiche (indipendenti dai tipi di parametri) e a passare come parametro di una funzione un'altra funzione!
- I puntatori a funzione si dichiarano in questo modo

```
tipo (*nome_funzione)(tipo par1, tipo par2, ...)
```

- la parentesi attorno a `*nome_funzione` è necessaria, senza parentesi indica una funzione che restituisce un valore di tipo `tipo`
- Tale funzione può essere parametro di un'altra, così:

```
funzione2(tipo par1, tipo (*nome_funzione)(tipo par1, tipo par2) )
```

# Esercizio con puntatori a funzione

- Esaminiamo un esempio
- Scaricare pacchetto appunti2

A casa:

- Mantenendo inalterata la funzione `generic_min`, modificate il codice affinchè si possa calcolare il minimo di un array di interi.

# Processi

---

# Processi

---

## Processo:

- attività di elaborazione guidata da un programma
  - Istanza di un programma in esecuzione
- 
- Il processo è identificato da un PID (PID = Process IDentifier)
  - Per ogni processo attivo il SO deve mantenere in memoria una **immagine** che possiamo vedere suddivisa in 3 parti:
    - Codice: in genere condiviso e read-only
    - Dati: globali, stack, heap...
    - Dati processo/SO: Interazione fra processo e s.o., es. tabella file aperti (implementazione: parte in tabella processi, parte in tabella globale)

# Processi: creazione

---

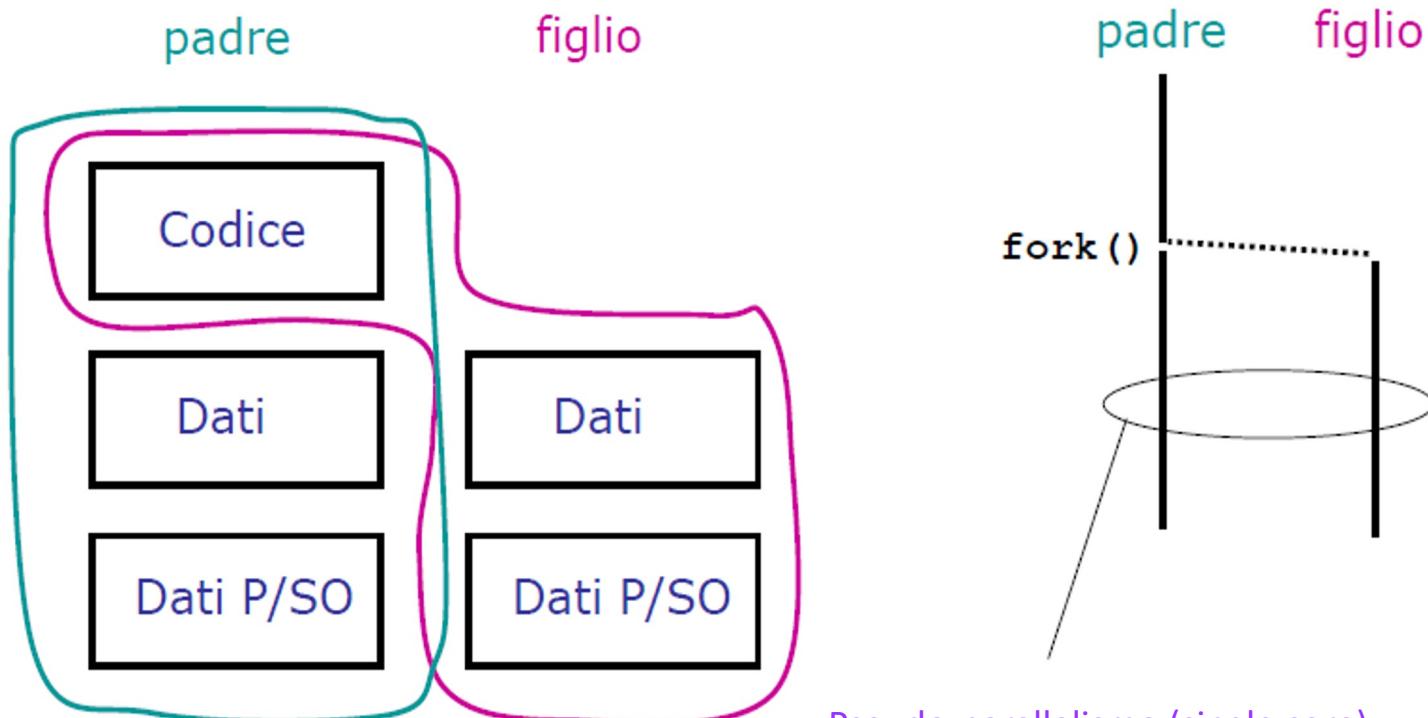
In Unix un processo si crea con una chiamata di sistema

*fork();*

duplica il processo in esecuzione (“**padre**” – parent process), creandone una copia (“**figlio**” – child process) quasi identica:

- ha una copia dell’immagine (il codice può essere condiviso)
- ha un diverso PID
- la funzione `fork` restituisce al processo padre il PID del figlio (un valore sicuramente  $\neq 0$ )
- il nuovo processo “nasce” uscendo anch’esso dalla chiamata di funzione `fork()` ottenendo però come risultato 0

# Processi: effetto della creazione



# Processi: effetto della creazione

## Processo P1 (padre)

Valore dei registri della "CPU virtuale" di P1

$PC = \dots$   
 $SP = \dots$   
...  
...

Altre informazioni su P1 mantenute dal Sistema Op.  
 $pid = 10$

Immagine in memoria di P1

codice  
...  
 $k = fork()$   
 $if (k==0)$   
  {XXXXX}  
else  
  {YYYYY}  
...

dati & stack

$i = 3$   
 $j = 10$   
 $k = 1$

# Processi: effetto della creazione

## Processo P1 (padre)

Valore dei registri della "CPU virtuale" di P1

PC = ...  
SP = ...  
...  
...

Altre informazioni su P1 mantenute dal Sistema Op.  
**pid = 10**

Immagine in memoria di P1

```
codice  
...  
k = fork()  
if (k==0)  
{XXXXX}  
else  
{YYYYY}  
...
```

dati & stack

i = 3  
j = 10  
**k = 15**

## Processo P2 (figlio di P1)

Valore dei registri della "CPU virtuale" di P2 (= a p1)

PC = ...  
SP = ...  
...  
...

Altre informazioni su P2 mantenute dal Sistema Op.  
**pid = 15**

Immagine in memoria di P2 (quasi = a P1)

```
codice  
...  
k = fork()  
if (k==0)  
{XXXXX}  
else  
{YYYYY}  
...
```

dati & stack

i = 3  
j = 10  
**k = 0**

# Processi: effetto della creazione

## Processo P1 (padre)

Valore dei registri della "CPU virtuale" di P1

*PC = ...*  
*SP = ...*  
...  
...

Altre informazioni su P1 mantenute dal Sistema Op.  
*pid = 10*

Immagine in memoria di P1

```
codice  
...  
k = fork()  
if (k==0)  
{XXXXX}  
else  
{YYYYY}  
...
```

dati & stack  
*i = 3*  
*j = 10*  
***k = 15***

## Processo P2 (figlio di P1)

Valore dei registri della "CPU virtuale" di P2

*PC = ...*  
*SP = ...*  
...  
...

Altre informazioni su P2 mantenute dal Sistema Op.  
*pid = 15*

Immagine in memoria di P2

```
codice  
...  
k = fork()  
if (k==0)  
{XXXXX}  
else  
{YYYYY}  
...
```

dati & stack  
*i = 3*  
*j = 10*  
***k = 0***

# Esercizi

---

- Esaminare ed eseguire il programma *clona.c* da appunti2
  - *Come si comporta?*

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

int main()
{
    fork();
    printf("Hello World\n");
}
```

# Processi: schema di creazione

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

pid_t pid;

if ((pid = fork()) < (pid_t) 0) {
    /* errore nella fork */
} else if (pid == (pid_t) 0) {
    /* codice eseguito solo dal p. figlio */
} else {
    /* codice eseguito solo dal p. padre */
}
```

# Esercizi

---

- Esaminare ed eseguire *par.c*
  - Modificare il programma affinchè i numeri delle iterazioni di entrambi i cicli vengano passati come argomenti
  - ... poi eseguire il programma ridirigendo l'output su file, es. chiamando "par arg1 arg2 > pippo" se "par" è il nome dell'eseguibile.
  - L'alternanza delle stringhe stampate su file dai due processi è diversa rispetto al caso di output su video? Perché?
  - Modificare il programma affinchè le stampe su file risultino come l'output su video

# Processi: terminazione

---

- La terminazione normale di un processo avviene:
  - Ritornando dal `main()`
  - Chiamando la funzione `exit()`
    - Vengono chiusi tutti gli stream aperti e svuotati i buffer
  - Chiamando la funzione `_exit()` o `_Exit()`
  - Ritornando dall'ultimo thread di un processo
  - Chiamando `pthread_exit()` dall'ultimo thread di un processo
- La terminazione anormale avviene:
  - Chiamando `abort()`
  - Ricevendo un segnale
- Tutte le funzioni exit si aspettano un argomento intero chiamato `exit_status`
- Nel `main return(0)` è equivalente a `exit(0)`

# Processi: attesa e terminazione

---

- Un processo p1 può attendere la terminazione di un processo figlio con:

$$y=wait(&x);$$

- (e una variante *waitpid* più generale). In realtà non sempre c'è una attesa – ad es. se quando viene chiamata c'è già un figlio terminato, ma in entrambi questi casi wait, all'uscita della funzione, informa p1 che un suo processo figlio p2 è terminato.
- il PID del processo terminato (p2) viene restituito dalla funzione
- nell'intero il cui puntatore è stato passato a wait vengono messe informazioni su come p2 è terminato (di sua iniziativa, o è stato interrotto tramite i “segnali” più avanti nel corso)
- Queste informazioni vengono (per default) mantenute nella tabella dei processi dopo che un processo termina, in attesa di essere lette con wait dal processo che lo ha generato; una volta che sono state lette, per default l'elemento della tabella relativo al processo viene etichettato libero, in modo da non occupare un posto

# Processi: attesa e terminazione

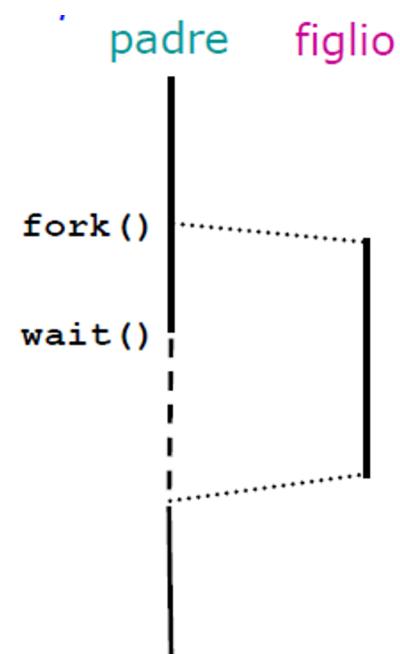
Se p1 ha almeno un processo figlio p2 terminato che non è ancora stato “aspettato” (nessuna wait ha dato a p1 l’informazione che p2 è terminato, o comunque quell’informazione è ancora nella tabella), allora p1 esce subito dalla wait, con l’identificatore di p2

Altrimenti, se p1 ha processi figli non terminati, viene sospeso fino a quando uno di questi termina

Altrimenti? (tutti i p. figli, se ce ne sono stati, sono terminati e sono stati “aspettati”)

R: dalla chiamata si esce subito con errore.

Se p1 venisse sospeso, rimarrebbe sospeso “per sempre”



# Esercizi – generazione più processi

- Nel programma vengono generati 5 processi figli che vanno ad eseguire una stessa funzione con valori diversi del parametro:

```
8 void proc(int i)
9 {
10    int n;
11    printf("Processo %d con pid %d\n",i,getpid());
12    for (n=0;n<500000000;n++);
13 }
14
15 int main(int argc, char *argv[])
16 {
17     int i, status;
18     pid_t pid;
19
20     for(i=0;i<5;i++)
21     {
22         if (fork()==0){
23             proc(i);
24             exit(0); ←
25         };
26         for(i=0;i<5;i++)
27         {
28             pid = wait(&status);
29             if(WIFEXITED(status)){
30                 printf("Processo figlio %d terminato regolarmente, stato d'uscita: %d\n", pid, WEXITSTATUS(status) );
31             }
32         }
33     }
34     return 0; /* ... ma facciamo male a non verificare errori nelle system calls */
35 }
```

Cosa succede rimuovendo la `exit(0)`?

Scrivere un programma analogo usando `waitpid` al posto di `wait` – vedi manuale

# Esercizi

---

- Modificare uno degli esempi precedenti, aggiungendo una variabile che permetta di verificare che processo padre e figlio hanno due copie delle variabili: il figlio eredita i valori precedenti alla fork, ma le modifiche successive (visualizzate con delle printf) nei due processi sono indipendenti.

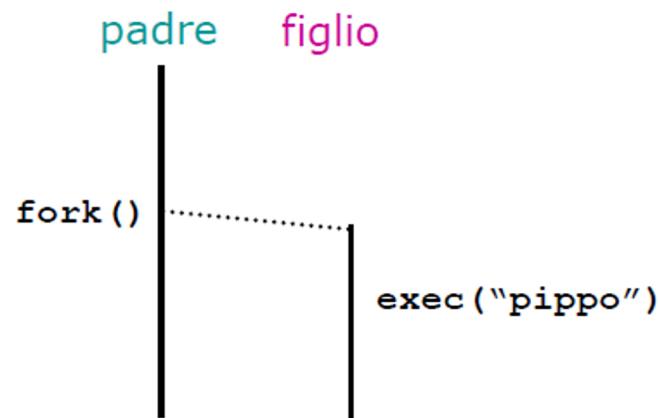
# Parte II

---

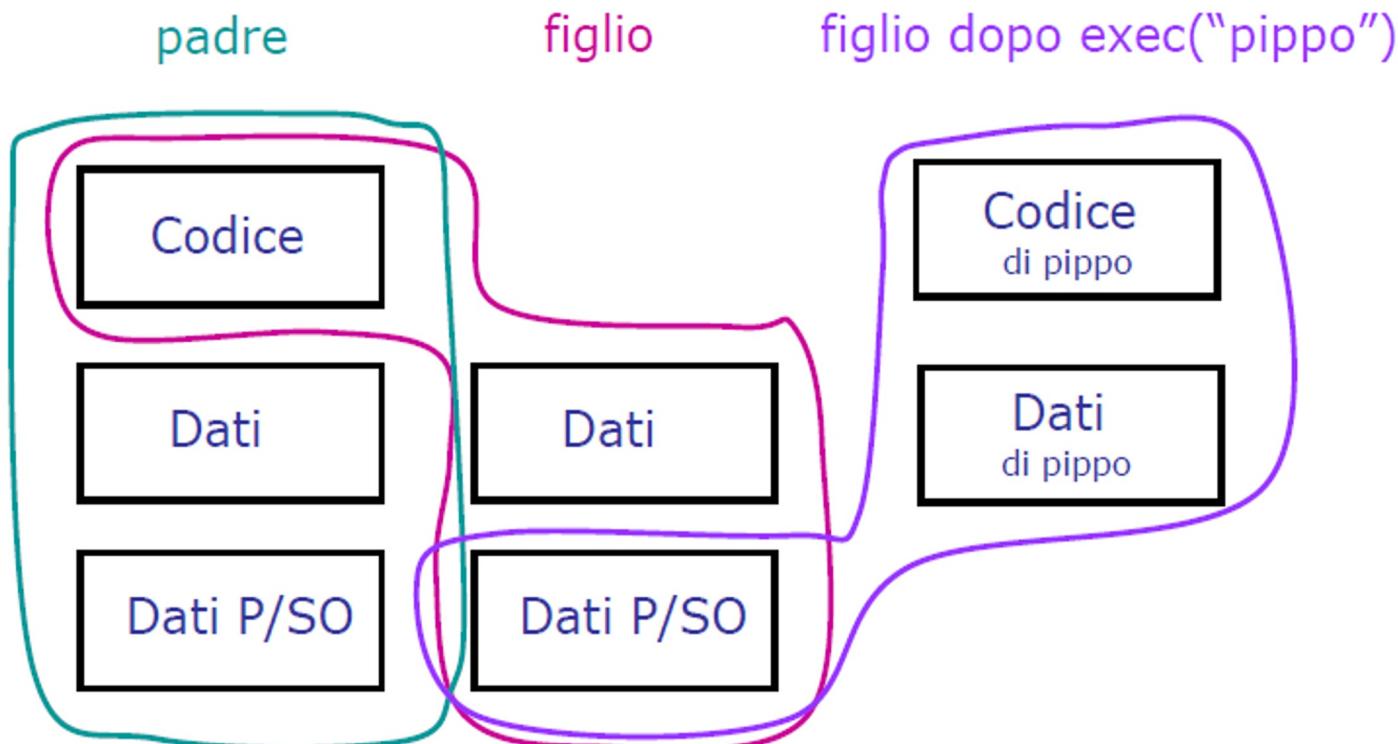
# Processi: esecuzione di un programma

- Clonare un processo identico non pare molto utile (in alcune casi si lo è)
- Con `if (fork() == 0) ...`
  - bisogna scrivere nello stesso programma l'intero codice eseguito dal p. figlio
- In effetti, spesso non si fa così, nel ramo del `p.figlio` si può usare una chiamata di sistema della “famiglia” `exec`

Per la precisione sono diverse funzioni C che permettono di accedere alla stessa chiamata di sistema, qui semplificate in:  
`exec(nome_file_eseguibile)`



# Processi: esecuzione di un programma



NB arrivati a questo punto, fra il p. figlio e il "Codice" iniziale non c'è più alcuna relazione

# Processi: exec in dettaglio

- **int execl(const char \*pathname, const char \*arg0, ... /\* (char \*) NULL \*/ );**

In **execl** si specifica una **lista** di argomenti da passare all'eseguibile terminati da **NULL**. Es:

- `execl("/bin/ls", "ls", "-l", (char *) NULL);`

- **int execv(const char \*pathname, char \*const argv[]);**
  - Sostituiscono il codice (e i dati) attuali del processo chiamante con quelli dell'eseguibile *pathname*, restituisce -1 in caso d'errore.

In **execv** si passa un **vettore di stringhe** contenente **gli** argomenti da passare all'eseguibile terminati da **NULL**:

- `arg={"ls", "-l", NULL};`
- `execv("/bin/ls", arg);`

# Processi: exec in dettaglio

- **`int execlp(const char *filename, const char *arg0, ... /* (char *) NULL */ );`**
- **`int execvp(const char *filename, char *const argv[]);`**
  - Sostituiscono il codice (e i dati) attuali del processo chiamante con quelli dell'eseguibile *filename*, restituisce -1 in caso d'errore
- In **execlp/execvp** si passa il nome dell'eseguibile cercandolo nelle directory specificate in PATH. La variabile d'ambiente viene ereditata dal processo chiamante.

# Esercizi – exec

- Esaminare il programma provaexec.c ed hello.c in appunti2 e provate ad eseguirlo:

```
:::::::  
provaexec.c  
:::::::  
#include <sys/types.h>  
#include <unistd.h>  
#include <stdio.h>  
main()  
{  
    pid_t n;  
    int s;  
    if((n=fork())== (pid_t)-1)  
        {perror("fork fallita");  
         exit(1);  
     }  
    else if (n==(pid_t)0)  
        /* processo figlio */  
        exec("hello","hello",NULL);  
        perror("exec fallita");  
    }  
    else  
        /* processo padre */  
        wait(&s);  
}
```

```
:::::::  
hello.c  
:::::::  
main()  
{  
    printf("Hello \n");  
}  
:::::::
```



# Esercizi – exec

---

- Modificare il precedente esercizio usando la funzione execv al posto di execl.
- Verificare i comportamenti indesiderati che si possono ottenere se si fa una stampa con printf che NON causa lo svuotamento del buffer e poi si chiama:
  - Fork
  - Exec
- per testarlo potete modificare in modo opportuno provaexec.c

# Esercizio

---

- Scrivere un programma che, preso come argomento da linea di comando un percorso ad un file (eventualmente da creare), generi 3 processi nell'ordine P1, P2 e P3. Ciascuno di essi scrive sul file la stringa:
  - Sono il processo P*<i>* con pid: <pid>\n
- I processi devo scrivere sul file in ordine **inverso** alla loro creazione,
- ossia si deve ottenere nel file la sequenza:
  - Sono il processo P3 con pid: <pid3>\n
  - Sono il processo P2 con pid: <pid2>\n
  - Sono il processo P1 con pid: <pid1>\n
- Verificare che l'ordine sia preservato indipendentemente dallo scheduling della CPU, ad esempio ritardando con una sleep la scrittura da parte del processo P3.



UNIVERSITÀ DEL PIEMONTE ORIENTALE

# Sistemi Operativi 1

## Laboratorio 3

**Prof. Luisa Barrera León**

# Parte I

---

# Segnali

**Segnale** in Unix: meccanismo con cui il S.O. avvisa un processo P che si è verificato un evento.

1. **asincrono** cioè scorrelato con ciò che P (in esecuzione o non) sta facendo, ad esempio:
  - a) l'utente ha battuto un carattere di interruzione
  - b) Un altro processo Q ha effettuato una chiamata di sistema per inviare il segnale a P – anche se questo non è il meccanismo migliore per la sincronizzazione fra processi
  - c) è suonata una sveglia “caricata” in precedenza da P (granularità: secondi)
2. **sincrono**: causato da ciò che P, in esecuzione, ha cercato di fare (es. eseguire una istruzione illegale, o accedere ad un indirizzo a lui non permesso)



# Segnali: tipologie

- I diversi tipi di eventi, e quindi i tipi di segnali, in C sono identificati da un valore numerico e una corrispondente costante simbolica, es:
  - 2 SIGINT carattere di interruzione (da tastiera)
  - 4 SIGILL istruzione hw illegale
  - 9 SIGKILL terminazione (comportamento non modificabile)
  - 11 SIGSEGV accesso alla memoria non valido
  - 14 SIGALRM sveglia
  - 15 SIGTERM terminazione
  - SIGUSR1/SIGUSR2 segnali definibili dall'utente
- *L'elenco completo sul man (man -s7 signal)*
- I valori numerici corrispondenti possono variare da versione a versione quindi *meglio usare le opportune costanti*

NB: è possibile per un processo P inviare a un altro processo Q qualunque segnale, anche SIGSEGV, ma solo se P è abilitato a inviare segnali a Q.

# Segnali: ciclo di vita

---

- Il segnale è **generato/spedito** ad un processo quando un particolare evento causa l'occorrenza del segnale:
  - Eccezioni hw, es: divisione per 0
  - Condizioni sw, es: attivazione allarme
- Il S.O. **consegna (delivers)** il segnale al processo. Nel periodo dalla spedizione alla consegna il segnale è detto in **attesa (pendig)**.
- Per ogni tipo di segnale c'è un **comportamento per default** che può essere:
  - ignorare il segnale
  - terminare (variante: terminare salvando un file “core” che può servire per il debugging)
  - sospendersi – poi si può farlo ripartire
- Esempi di comportamento per default:
  - SIGINT terminare
  - SIGSEGV terminare con salvataggio del “core”
  - SIGSTOP sospendersi

# Reazione al segnale

- Un processo P può chiedere al sistema operativo di modificare il proprio comportamento in caso di consegna di un segnale di tipo S rispetto al comportamento di default. In particolare può:
  1. **ignorare** il segnale
  2. eseguire una funzione (“**catturare**” il segnale – signal catching), che può servire ad es. Per:
    - eseguire “ultime volontà” (la funzione svolge qualche operazione, ad es. rimuove dei file temporanei, e poi chiama exit)
    - rileggere file di configurazione
  3. **ripristinare** il comportamento di default
- Il comportamento si può modificare solo se S non è SIGKILL (in quel caso si termina sempre) o SIGSTOP che serve a “sospendere” il processo

ignora(s)



# L'interfaccia affidabile

Come si chiede di **modificare** il comportamento? Esiste un modo “non affidabile” e uno “affidabile” (reliable signals)

L'interfaccia “affidabile” (**reliable signals**) è leggermente meno semplice da programmare:

1. Si deve riempire la seguente struttura:

```
struct sigaction {  
    void (*sa_handler)(int); /* addr of signal handler, */  
                           /* or SIG_IGN to ignore isignal, or SIG_DFL  
                           to reset default behavior*/  
    sigset_t sa_mask;      /* additional signals to block */  
    int sa_flags;          /* signal options */  
    ...  
};
```

dove **\*sa\_handler** è un puntatore alla funzione da eseguire alla ricezione del segnale oppure **SIG\_IGN** (ignoro segnale) o **SIG\_DFL** (ristabilisco comportamento standard)

2. Si passa poi come argomento alla funzione **sigaction** e da quel punto in poi le direttive avranno effetto

# L'interfaccia affidabile

- Lo schema tipo di utilizzo è:

```
struct sigaction act,old;          /* nuova/vecchia azione */  
act.sa_handler=f;                /* oppure SIG_IGN o SIG_DFL */  
sigemptyset(&act.sa_mask);        /* azzera maschera segnali  
act.sa_flags = 0;                /* associa act, ricorda old */  
sigaction(s,&act,&old)
```

- Per default un tipo di gestione del segnale rimane finché non si chiede di cambiarla
- Ogni processo ha una `signal mask`: insieme di segnali la cui consegna è bloccata mentre si sta eseguendo la routine di gestione di un altro segnale.
- `sigemptyset(&act.sa_mask)` assegna a `act.sa_mask` l'insieme vuoto, poi passato a `sigaction` come insieme di segnali da bloccare durante l'esecuzione dell'handler.

# Invio di segnali

---

- Come inviare un segnale:
  - con la chiamata di sistema  
`kill(pid_destinatario, tipo_segnale)`
  - con il comando kill che è realizzato con la chiamata di sistema
- Il nome kill è fuorviante, poichè si può inviare qualunque tipo di segnale, non solo SIGKILL, e (come già visto) non tutti i segnali uccidono

# Chiamate di sistema interrotte

---

- Che succede se arriva un segnale «da catturare» durante una chiamata di sistema «lenta», es.:
  - lettura da tastiera
  - wait, waitpid
- In tal caso la chiamata viene interrotta: si esce con valore -1 ed errno=EINTR
- Con sigaction si può però chiedere che la chiamata di sistema venga invece fatta ripartire, per i dettagli vedere ad es. *Stevens & Rago, Advanced Programming in the Unix Environment*
- In questo corso non vedremo come far ripartire la chiamata di sistema, ma dovrete sapere come interpretare, ed eventualmente risolvere, un errore di EINTR

# Esercizi

---

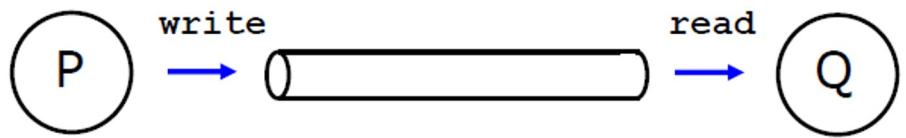
- Scaricare dal sito del corso il codice di prova su segnali e pipe (appunti2b)
- ESERCIZIO 4.1: modificare il programma sig.c affinché stampi il *PID* del processo
- ESERCIZIO 4.2: inviare da terminale segnali con il comando *kill* al processo che esegue "sig.c": inviare il segnale di interruzione (SIGINT), quello di terminazione (SIGTERM) e quello di accesso non valido alla memoria (SIGSEGV).
- ESERCIZIO 4.3: modificare il programma in modo da *ignorare* il segnale di interruzione.
- ESERCIZIO 4.4: verificare, integrando gli esempi precedenti che utilizzano *fork*, *exec* e *sigaction*, se le disposizioni "ignorare il segnale" ed "eseguire una funzione" vengono:
  - "ereditate" da un processo figlio, qualora richieste dal processo padre prima della fork;
  - mantenute da un processo che effettua una system call exec.

# Part II

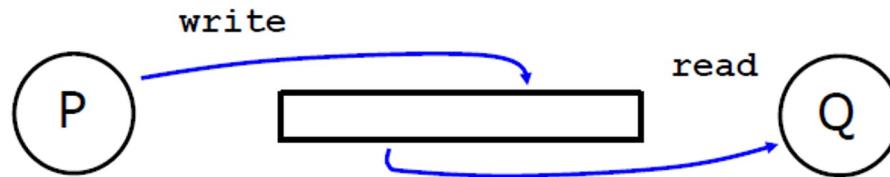
---

# Pipes

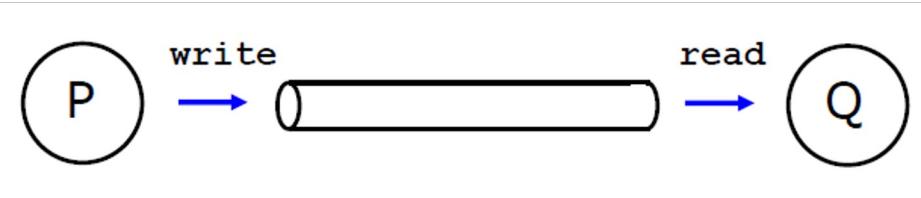
- Le pipes (pipe = tubo, tubazione) sono un meccanismo di comunicazione tra processi che si appoggia alle chiamate di sistema per accedere ai files:



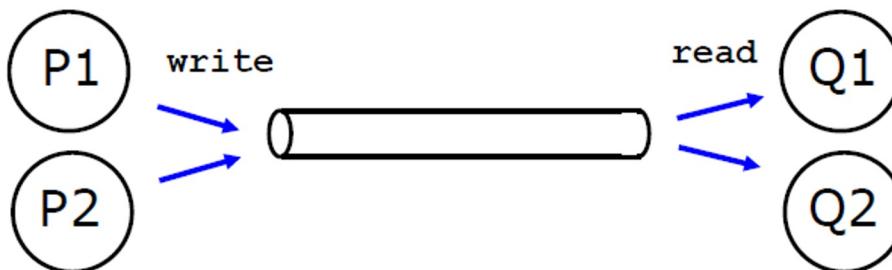
- In realtà i dati non si “muovono” nel “tubo”:



# Pipes

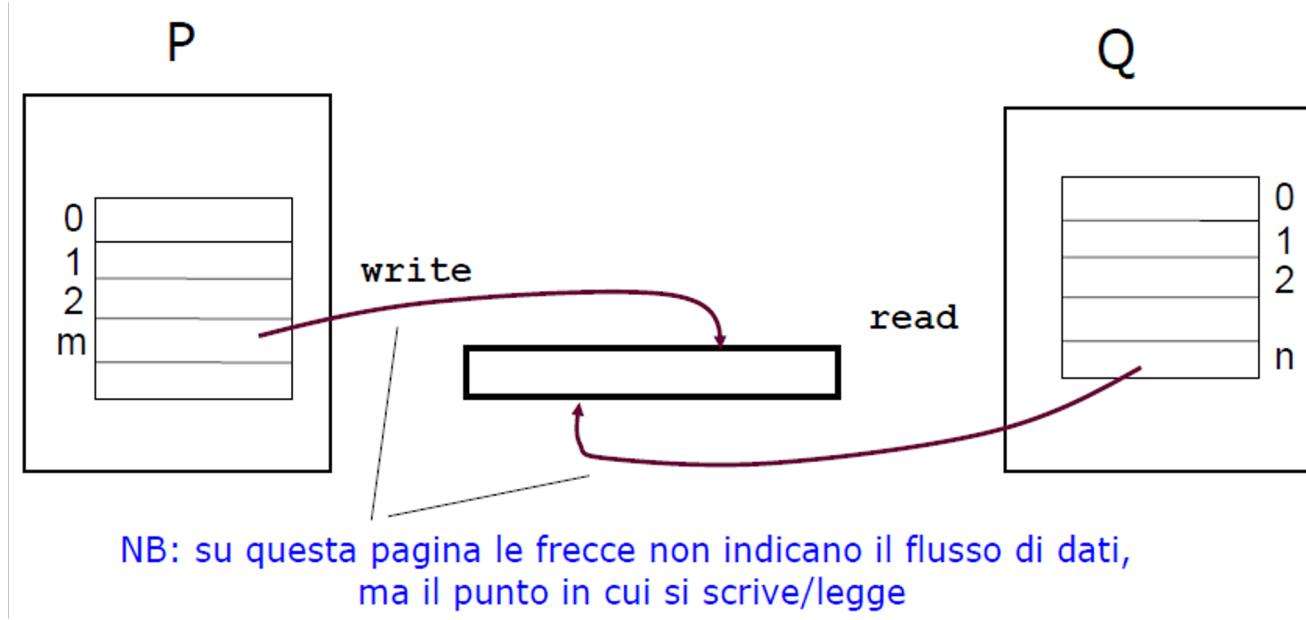


- Una read da una pipe può **sospendere** il processo chiamante (Q), se tutti i dati scritti nella pipe sono già stati letti (pipe “vuota”)
- Una write su una pipe può **sospendere** il processo chiamante (P), se i dati scritti e non ancora letti occupano tutto lo spazio allocato (pipe “piena”)
- Possono esserci più “produttori” e “consumatori”:



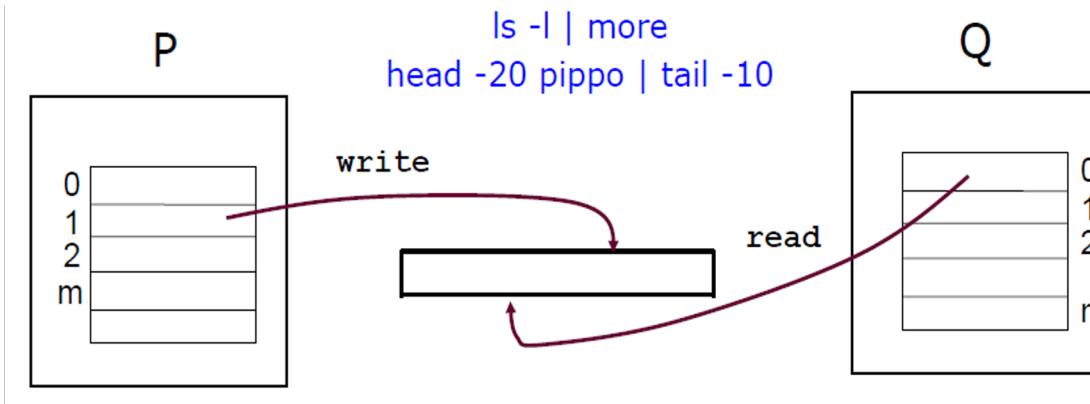
# Pipes

- NB: le strutture a cui ci si appoggia sono le stesse dei files: la tabella dei file aperti
- Proprio per questo si possono usare read e write



# Pipes

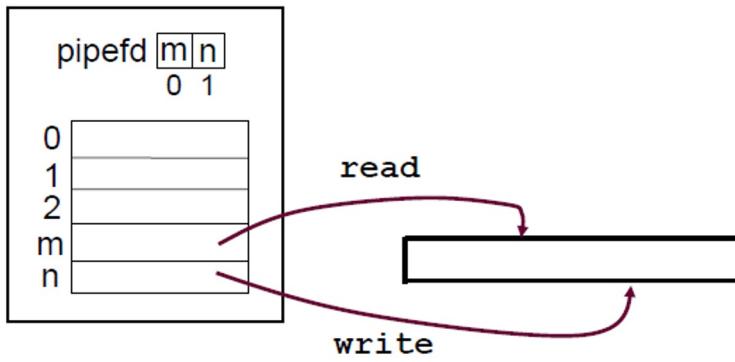
- Ma non solo... ci si può ridirigere lo standard input/output, per avere quello che negli interpreti di comandi si ottiene con:
  - comando1 | comando2
- Lo standard output di comando1 diventa lo standard input di comando2, es.:



- Ma come si implementa ?

# Pipes

P



- Una pipe si apre con la chiamata di sistema pipe:

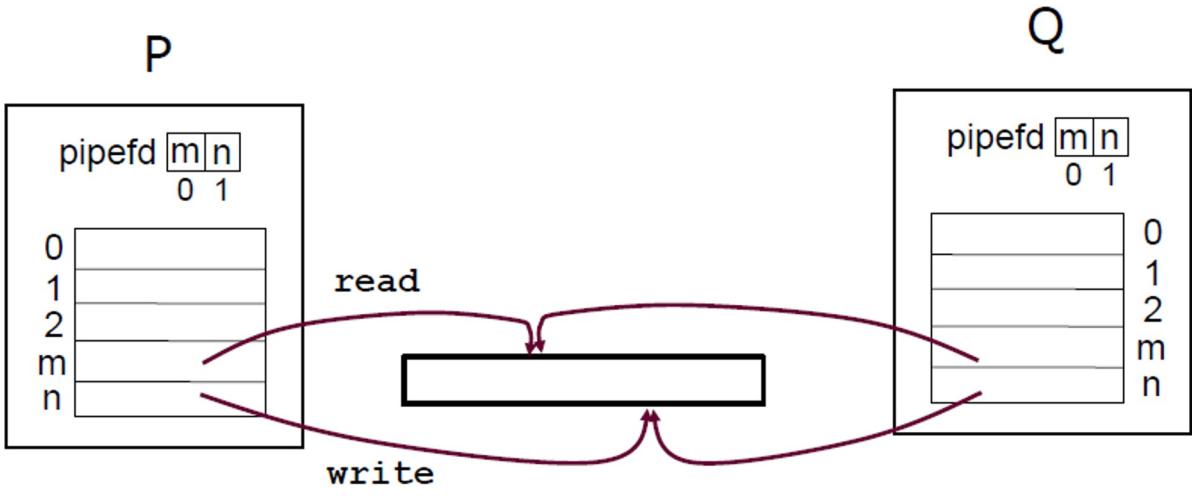
```
int pipefd[2];  
pipe(pipefd);
```

- Che alloca le strutture necessarie per gestire la pipe (es. buffer per parcheggiare i dati) e mette nell'array due “descrittori di file” (indici nella tabella dei file aperti) da usare per leggere/scrivere sulla pipe.

Che se ne fa un processo di tutto questo? Poco, se è uno solo

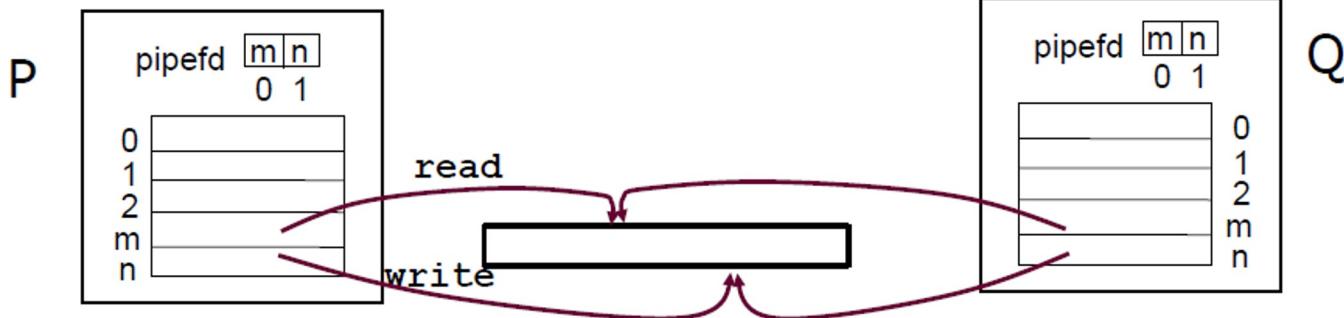
# Pipes

- Però dopo una fork() abbiamo:



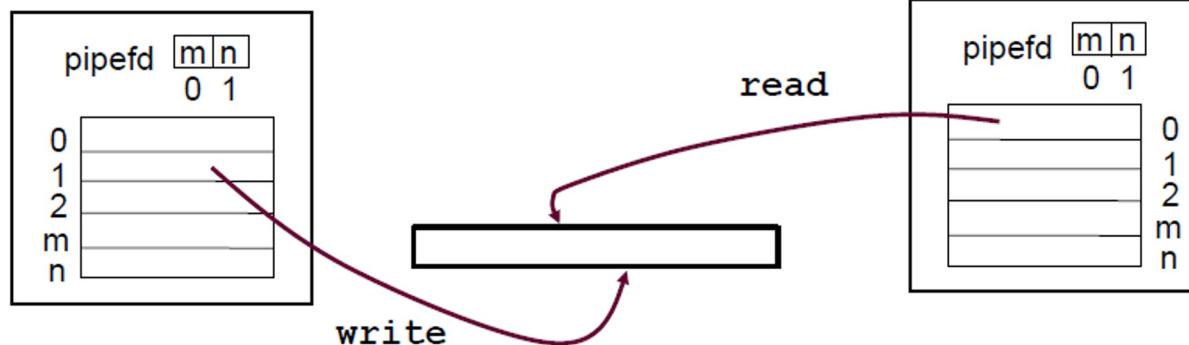
- E con opportune **dup** e **close** si ottiene quanto desiderato

# Pipes



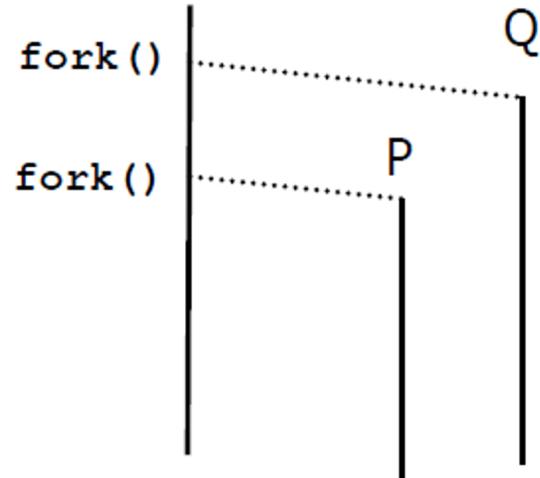
```
dup2(pipefd[1],1); // copia la riga n nella 1  
close(pipefd[0]); // a P non serve leggere  
close(pipefd[1]); // non serve più
```

```
dup2(pipefd[0],0);  
close(pipefd[0]);  
close(pipefd[1]);
```



# Pipes

- In realtà nell'esempio che vediamo in laboratorio, che riproduce:
  - comando1 | comando2
- dopo la chiamata di pipe vengono generati, come succede nella shell, due processi figli che ereditano copia della tabella dei file aperti
- Al processo padre (analogo dell'interprete shell) la pipe non serve, e quindi deve solo chiudere i descrittori dopo le fork



# Pipes

---

- Perché è importante chiudere i descrittori?
- 1. (sempre, quando un file aperto non serve) Per non tenere inutilmente occupate righe della tabella, di dimensione prefissata.
- 2. (per le pipe) Come fa il sistema operativo a dire “end of file” a un processo che legge da una pipe?
  - NB1: avere “end of file” è essenziale, es. per “while not EOF”.
  - NB2: il processo può non sapere che sta leggendo da una pipe, ad es. se legge dal proprio standard input che è stato ridiretto.
- Risposta: se la pipe è “vuota” **e non ci sono processi che hanno la pipe aperta in scrittura**.

# Esercizi

---

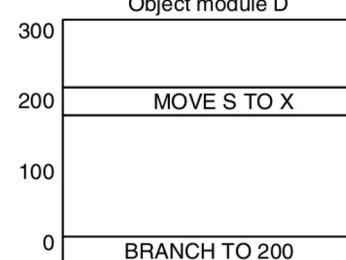
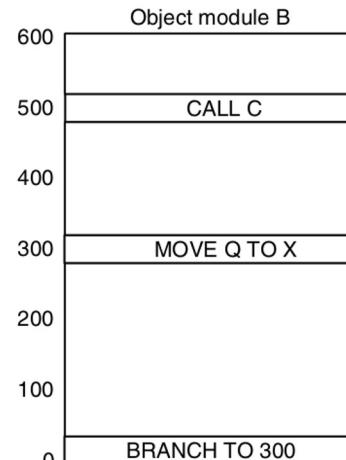
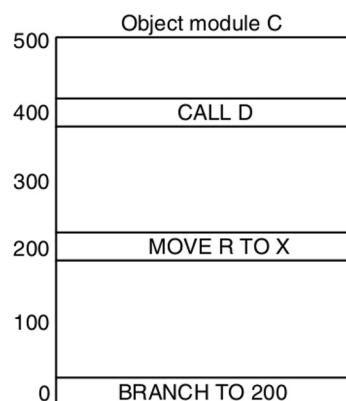
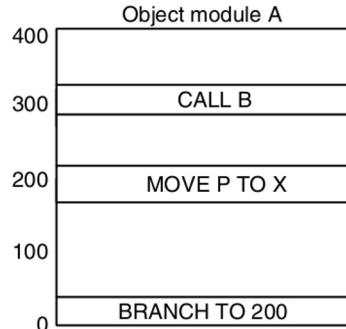
- **ESERCIZIO 4.5:** in pipes.c verificare che cosa accade se nel padre la close(pipefd[1]) viene tolta oppure messa dopo le wait, perchè?
- **ESERCIZIO 4.6:** Realizzare analogo programma che riproduce, invece del comando "ls -l | wc -l", il comando "head -20 pipes.c | tail -10" che produce in output le righe da 11 a 20 del file "pipes.c»
- **ESERCIZIO 4.7:** realizzare una pipe a due produttori e un consumatore. Il programma prende come argomenti le stringhe <file1> e <file2>, poi:
  - Un produttore invoca cat <file1>, un secondo produttore cat <file2> e il consumatore fa il more
  - Poichè non usiamo funzioni di sincronizzazione, l'output finale risulterà un mix confuso di <file1> e <file2>

# Parte III

---

# Collegamento (linking)

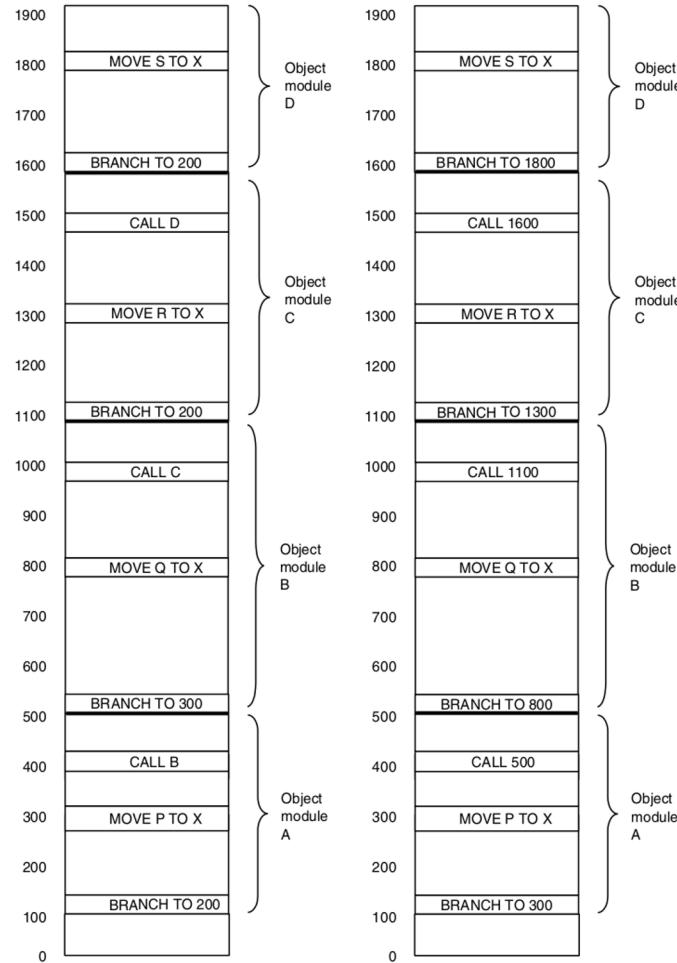
- Da A.S. Tanenbaum, Structured Computer Organization



- Diversi moduli oggetto, ciascuno con spazio indirizzi che inizia da 0
- Per il collegamento i moduli hanno una tabella dei simboli esterni.  
Es per il mod A:  
“B al byte 300”

# Collegamento (linking statico)

Moduli giustapposti,  
ma non rilocati e  
collegati



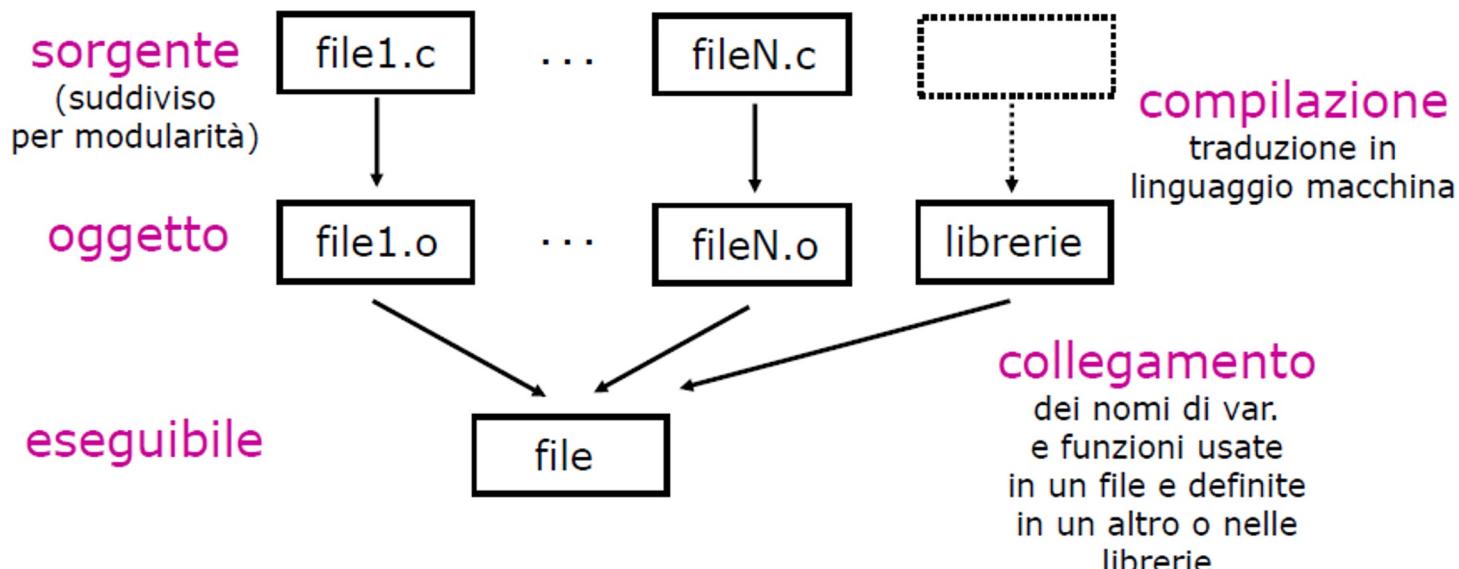
Dopo rilocazione  
e collegamenti

# Collegamento (linking)

- Perché serve il collegamento:
  - Per scrivere programmi modulari: isolare un “modulo” che contiene un insieme di funzioni (es. per inserire nodo in una lista, estrarre un nodo, trovare un nodo etc); le funzioni del modulo possono essere chiamate senza conoscerne l’implementazione, che può cambiare.
  - Per risparmiare spazio nei file sorgenti (rispetto a copiare e incollare il testo in C).
  - Per risparmiare tempo: evitare di ricompilare le librerie quando compiliamo un programma che la usa (rispetto a #includere il file in C contenente il codice delle funzioni).
  - Risparmiare spazio nei file eseguibili e poter modificare le librerie senza ricompilare, se si usa il collegamento **dinamico**: il codice non fa parte dell’eseguibile, il collegamento avviene, con meccanismi dipendenti dal sistema, al momento del caricamento o della prima chiamata.

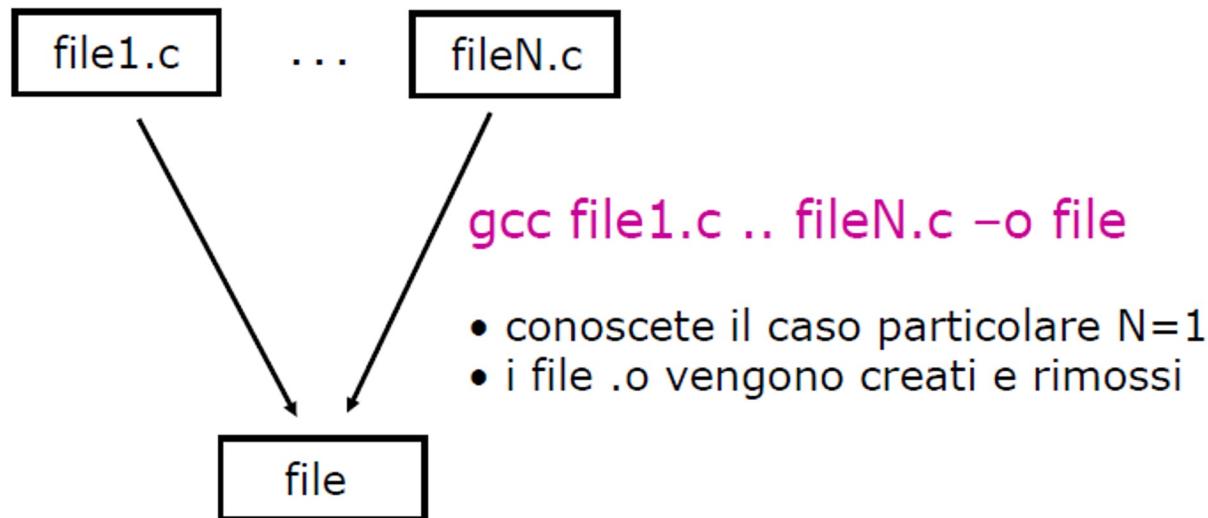
# Compilazione separata

- Richiamiamo alcuni concetti visti ad Architettura degli Elaboratori e aspetti pratici sulla compilazione/collegamento del C su Unix
- Per produrre un file eseguibile dal C in generale si ha:



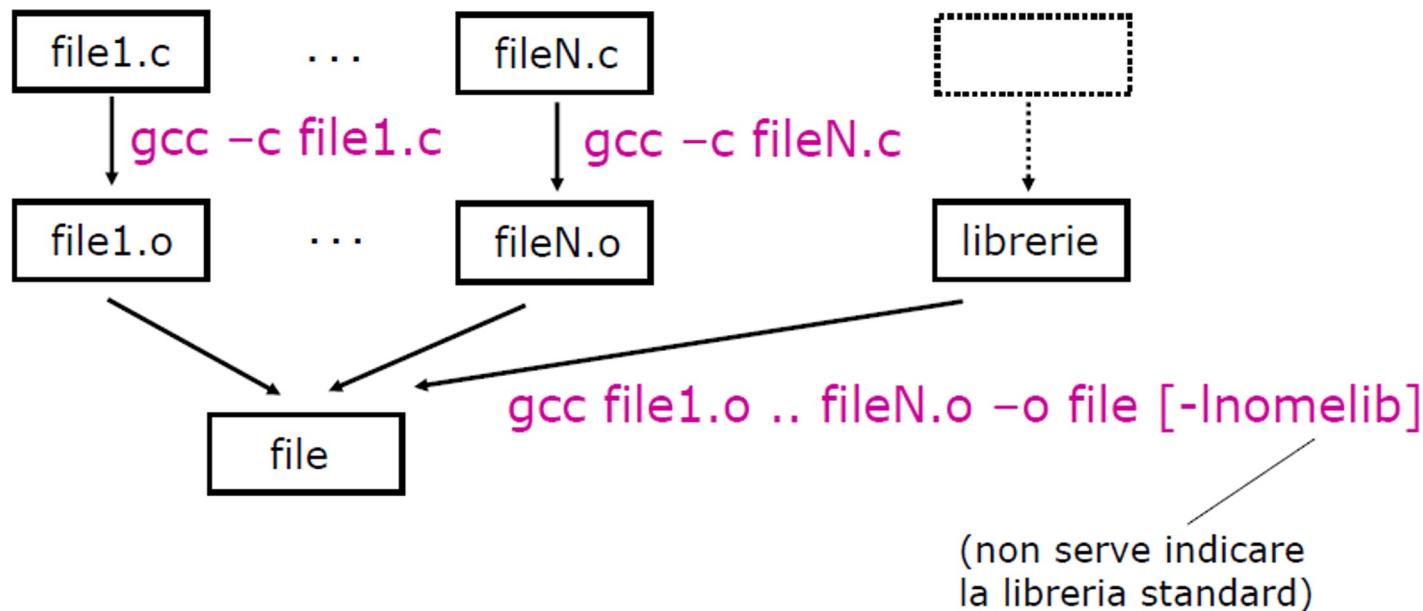
# Compilazione separata

- I compilatori ci permettono di fare tutto in un colpo:



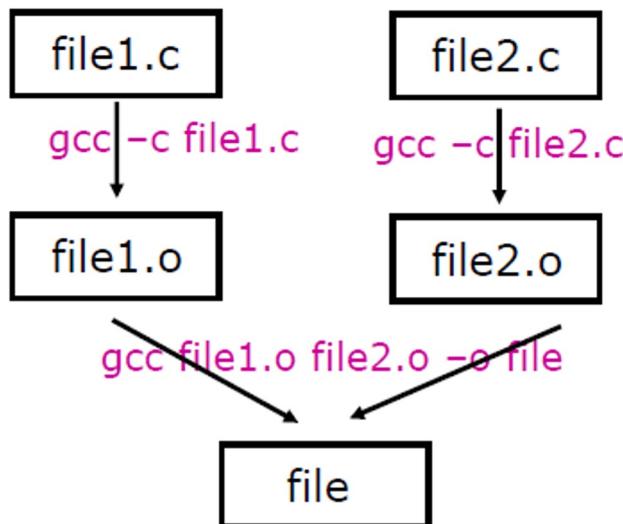
# Compilazione separata

- ... o in singoli passi:



# Il comando make

- Il comando make e i “makefile” da esso usati ci aiutano:
  - ad effettuare (solo) gli aggiornamenti necessari chiamando solo “make” quando uno dei file viene modificato – il che era più importante con le CPU lente di qualche anno fa
  - a tenere traccia delle dipendenze – es. chi fornisce il sorgente di un pacchetto fornisce anche le dipendenze, chi lo riceve chiama solo “make”



Contenuto del “makefile”:

```
file: file1.o file2.o
<TAB> gcc -o file file1.o file2.o
file1.o: file1.c
<TAB> gcc -c file1.c
file2.o: file2.c
<TAB> gcc -c file2.c
```

e si può fare molto altro...

# Il comando make

---

`make -f nomefile` // prende i comandi da nomefile

`make -n` // elenca i comandi che dovrebbe eseguire ma non li esegue

`make TARGET` // esegue le azioni per creare TARGET (provare `make clean` nell'esempio fornito)

Se in un *makefile* inseriamo nella prima riga un target «fittizio» e lo dichiariamo dipendente da ogni altro eseguibile che vogliamo creare otteniamo un *makefile* che compila tutti gli eseguibili elencati.

NOTA:

non deve esistere nella cartella un file con lo stesso nome del target



UNIVERSITÀ DEL PIEMONTE ORIENTALE

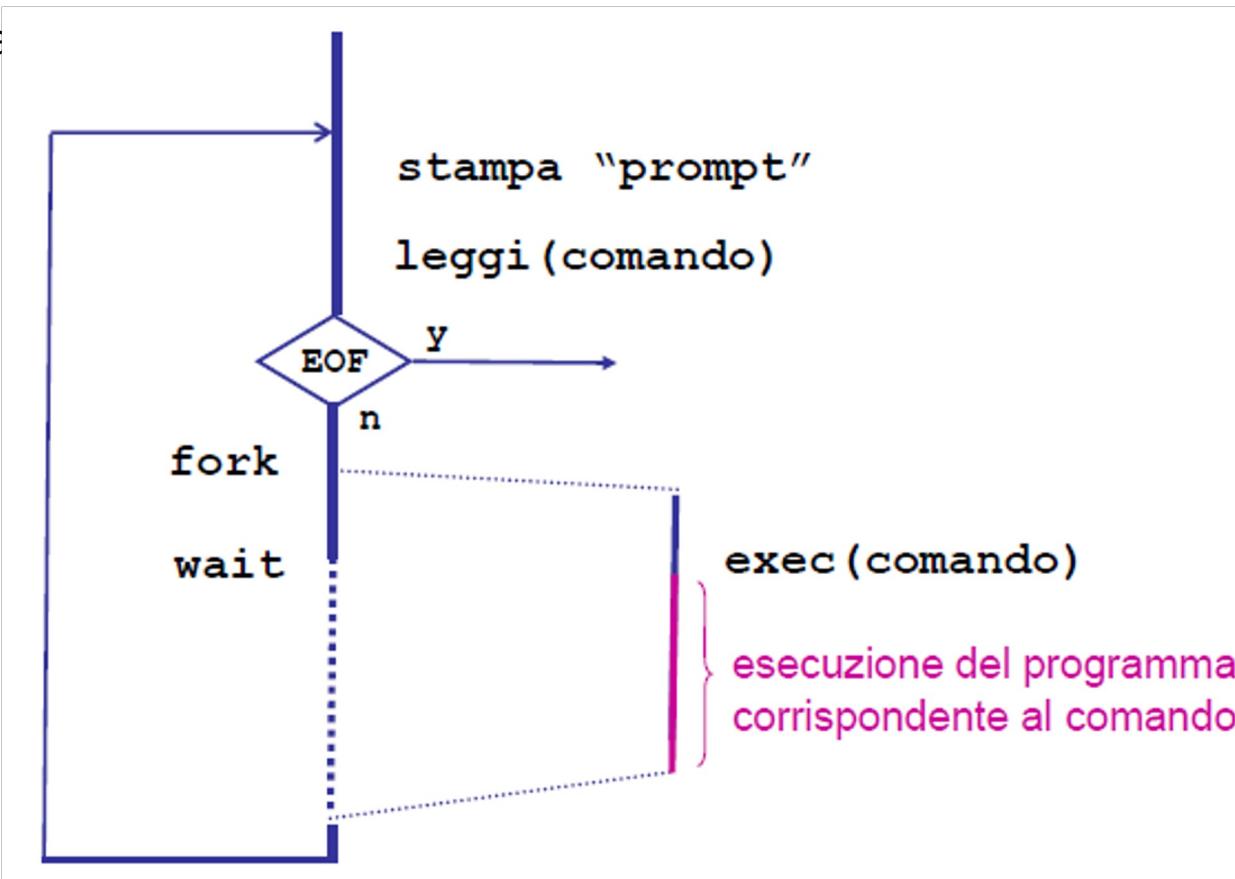
# Sistemi Operativi 1

## Laboratorio 4

**Prof. Luisa Barrera León**

# La shell - l'interprete dei comandi

- Funzionamento



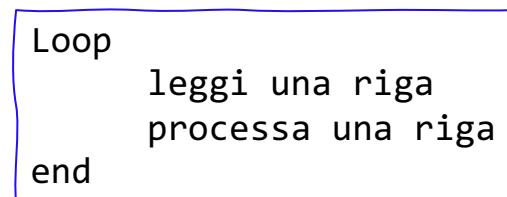
# La shell - l'interprete dei comandi

Perché creare un nuovo processo? Al solito, varie risposte:

1. La più importante: il programma eseguito dall'interprete e quello del comando devono avere ben poco contesto in comune, ad es.:
  - codice e dati sono ben diversi
  - il comando non deve poter accedere a tutti i dati dell'interprete
2. Per come è l'`exec`, se la facesse direttamente il processo che legge il comando, una volta finito il programma passato a exec, il processo terminerebbe senza leggere i comandi successivi.
3. L'interprete per default attende che il processo termini, ma non necessariamente; volendo (scrivendo “**comando &**”), il comando gira “in background” (nello sfondo) e nel frattempo si può usare l'interprete per lanciare altri comandi e controllare l'esecuzione del comando in corso.

# La shell - l'interprete dei comandi

- Il codice di un semplice interprete, la “small shell” (da Haviland, Gray & Salama, “Unix System Programming”, 2nd ed., Addison-Wesley 1998) realizza lo schema precedente
- Il sorgente è suddiviso(come es. di compilazione separata) in 2 file:
  - smallsh.c, che contiene le funzioni di maggior interesse per questo corso (effettuano le chiamate di sistema)
  - input.c, che contiene le funzioni per la lettura dell'input
- da compilare digitando make da linea di comando nella directory dell'interprete
- Il main è del tipo



# La shell - l'interprete dei comandi

---

- Una riga contiene una sequenza di “simboli”:
  - simboli speciali di un solo carattere, nella shell base: fine riga, “;”, “&”
  - sequenza di caratteri non speciali (e non spazi): nomi di comandi o argomenti

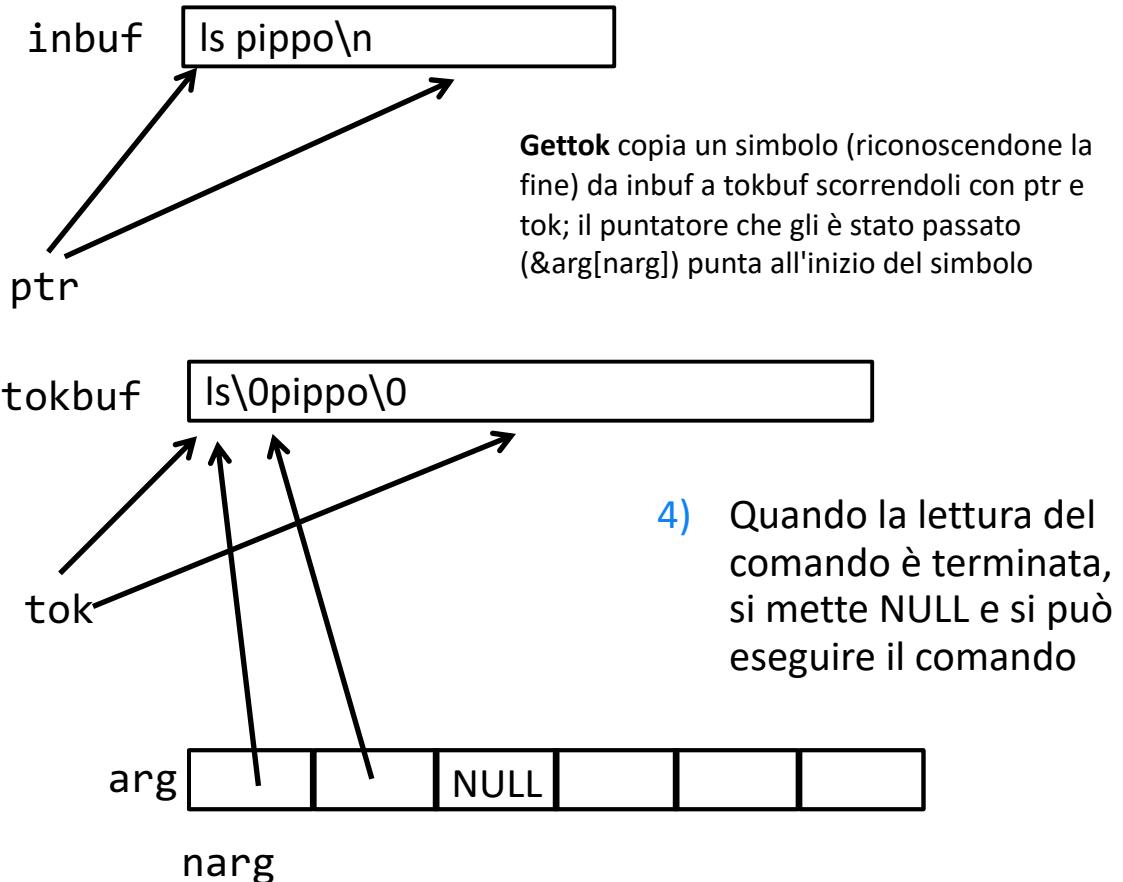
NB: gli spazi servono a separare i simboli

- Ad esempio, la riga:
  - ls qui quo qua <return>*  
contiene 5 simboli e un comando
  - ls qui ; ls quo <return>*  
contiene 6 simboli e 2 comandi

# La parsificazione

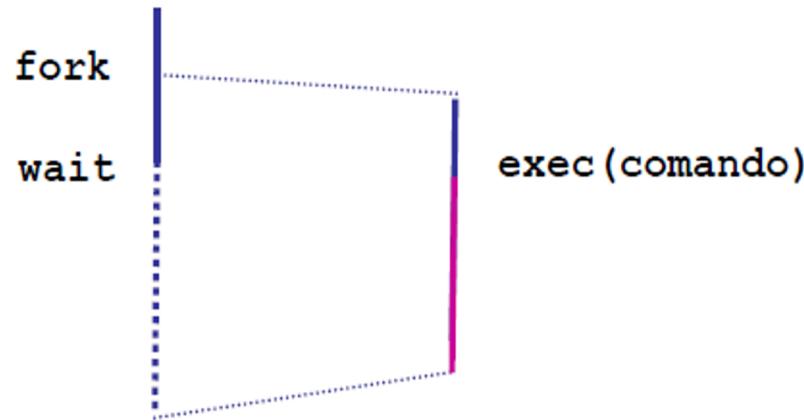
- 1) La riga in input viene parcheggiata in un array di caratteri.
- 2) Ogni volta che serve conoscere il prossimo simbolo, si scorre l'array copiando il simbolo in un altro, terminando con "\0" (= fine stringa in C)...  
3) Riempiendo anche un elemento di un array di puntatori da passare a execvp.

Userin riempie inbuf con la riga in input (usando l'indice count per scorrerlo)



# La shell - Schema Complessivo

- L'esecuzione del comando è la combinazione fork/exec/wait già vista:

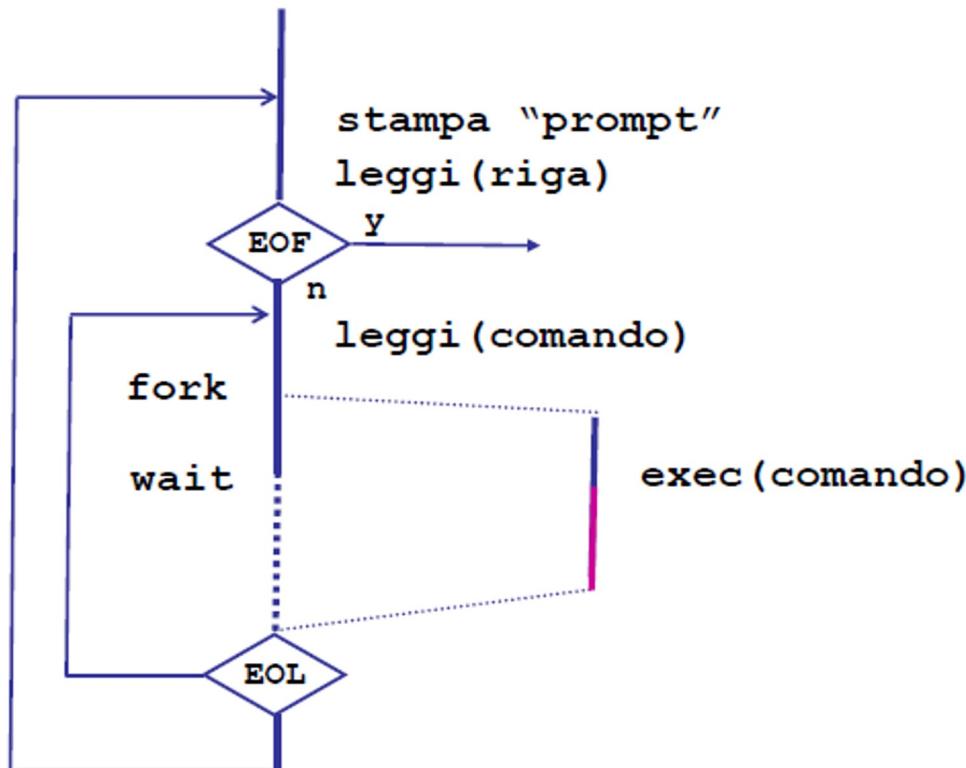


# La shell - Lo Schema Complessivo

- Ma la struttura complessiva è leggermente più complicata di quella vista inizialmente:

Comando1; Comando2

ls pippo; ls pluto



# La shell - Funzionalità Aggiuntive

La bash (Bourne Again Shell) e altre shell esistenti, fanno molto di più, ad esempio:

- Comando «in background», cioè non si aspetta la terminazione del processo che lo esegue (ma poi bisogna dare notizia della sua terminazione):

```
nomecomando arg1 ... argN &
```

- Ridirezione standard I/O con la notazione :

```
nomecomando arg1 ... argN < filein
```

```
nomecomando arg1 ... argN > fileout
```

```
nomecomando arg1 ... argN 2> errors
```

- Pipeline:

```
comando1 | ... | comandoN
```

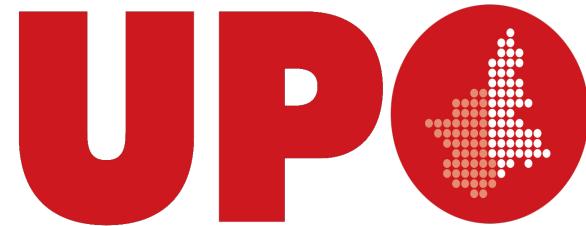
- ...

# Shell: esercizi

---

- Modificate il codice shell affinché lanci comandi «in background» ciò è non aspetti la terminazione del processo che lo esegue:
  - Inizialmente fate in modo che tutti i comandi siano lanciati in background
  - In seguito fate in modo che solo in caso di presenza del simbolo &, venga lanciato il comando in background
    - Fate attenzione che venga trattata correttamente l'esecuzione di un comando in background seguito da uno in foreground.

**Da consegnare per ammissione al esame!!!**



UNIVERSITÀ DEL PIEMONTE ORIENTALE

# Sistemi Operativi 1

## Laboratorio 5

**Prof. Luisa Barrera León**

# Parte I - Threads

---

# Posix Thread: funzioni

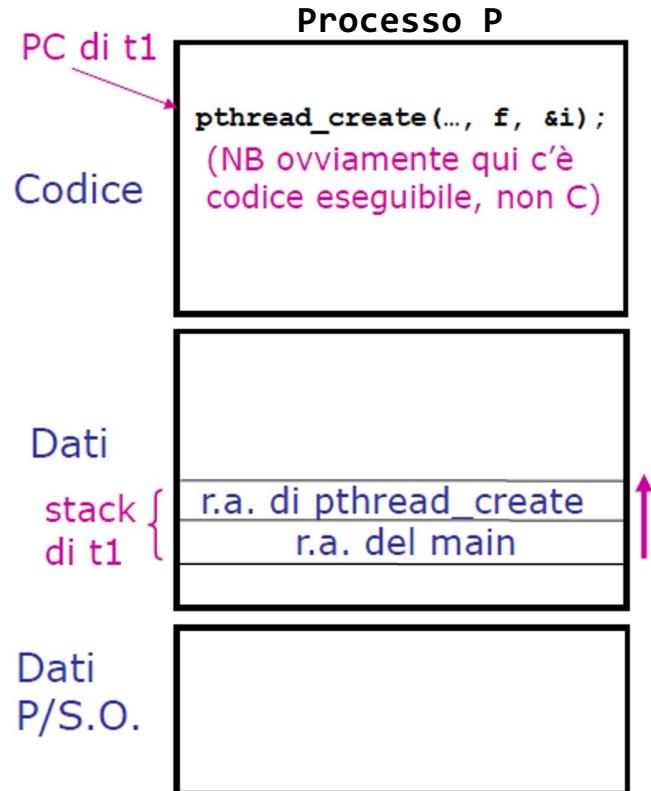
- Su molte versioni di Unix è disponibile la libreria dei **POSIX threads** (Pthreads in breve), cioè una libreria per la programmazione mediante threads che fa parte dello standard di Unix
- La libreria prevede:
  - una funzione *pthread\_create* che permette di creare, all'interno del processo corrente, un nuovo thread che esegue la funzione il cui puntatore viene passato come argomento alla create;
  - una funzione *pthread\_exit* che permette ad un thread di terminare (ma il thread termina anche quando esce dalla funzione passata alla create che lo ha generato);
  - una funzione *pthread\_join* che permette ad un thread di attendere la terminazione di un altro thread.
- **Vedere nel man l'opzione per il linking della libreria pthread!**
- Considerato il flusso di controllo, queste funzioni sono l'analogo di ciò che in programmi con processi multipli è rappresentato da:
  - la combinazione *fork-exec*;
  - *exit* (o l'uscita dal main);
  - *wait*

# Thread: creazione

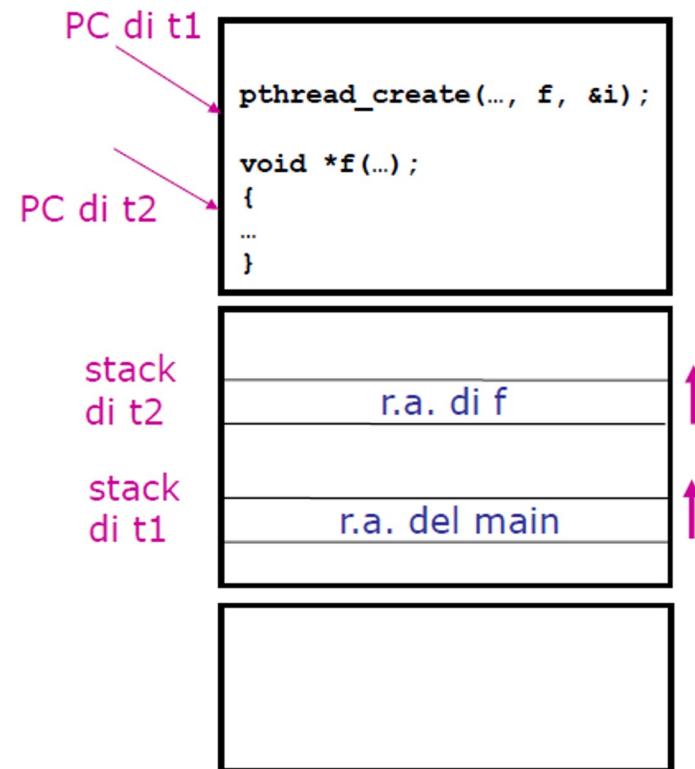
- Consideriamo un processo P (con codice sorgente in C) che inizia l'esecuzione. All'inizio P comprende un unico thread, chiamiamolo t1
- Se la funzione main (eseguita da t1) chiama:
  - `pthread_create(..., f, &i);`
- lo stack di t1 comprende il “record di attivazione” (che contiene una copia di parametri, variabili locali e altro) della funzione `main` e quello della funzione `pthread_create`.
- L'effetto è la creazione di un **nuovo thread t2**, allocando spazio per **nuovo stack**, sulla quale avviene la chiamata di `f(&i)`.
- La memoria accessibile al programma è automaticamente condivisa fra i vari threads. Esiste – ma non la trattiamo - la possibilità di allocare dati a cui un solo thread può accedere.
- Ogni thread ha degli attributi (es. dimensione della stack), con un valore di default, la `pthread_create` può specificare valori diversi (non si vede nel corso).

# Thread: effetto della creazione

Alla chiamata di *pthread\_create()*



All'uscita di *pthread\_create()*



# **pthread\_create()**

- Prototipo della funzione

```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

- Esempio di invocazione

```
#include <pthread.h>

void *tbody(void *arg)
{...}

pthread_t t;
. . .
pthread_create(&t, NULL, tbody, NULL);
```

- Compilazione e linking con libreria -pthread.

# Scheduling delle thread

- Lo scheduling dei threads nella libreria **può dipendere dalle scelte implementative**, inoltre gli attributi possono definire proprietà relative allo scheduling (ma le implementazioni non necessariamente supportano tutti i tipi di scheduling).
- In generale lo scheduling deve garantire che se un processo P ha almeno un thread T pronto per l'esecuzione, T deve essere preso in considerazione dallo scheduling.
  - Ad es. se un thread T1 di un processo P compie una operazione sospensiva, e P ha altri thread pronti, questi devono poter andare avanti.
- Questo deve avvenire se l'operazione sospensiva è una lettura da file e ancora più ovviamente se si tratta di una operazione di sincronizzazione con altri threads (T1 attende una segnalazione da altri thread, che devono poter girare per effettuarla).

# Scheduling delle thread

---

- A differenza del caso dei processi in sistemi interattivi, non è sempre indispensabile che avvenga il *timesharing* fra threads di uno stesso processo in quanto:
  - Diversi processi si trovano di solito (tranne nel caso di una applicazione costituita da processi multipli che cooperano) per caso a condividere le risorse del sistema e bisogna che non ne soffrano troppo: bisogna evitare che uno monopolizzi la CPU a danno degli altri.
  - Una applicazione costituita da diversi thread è probabilmente progettata come un tutto unico, e in generale basta che almeno un thread vada avanti; se si deve sincronizzare con altri (deve attendere qualcosa dagli altri), chiamerà una primitiva sospensiva, e potranno andare avanti solo gli altri.
  - I threads esistono per avere a costo basso (rispetto ai processi) la commutazione di contesto, quindi non costa moltissimo fare il *timesharing*

# Esercizi in appunti parte 3

---

- Compilare il programma `t1.c`
  - Esercizio 3.1: verificare che i thread condividono le variabili globali. Inserendo una variabile a cui si assegna un valore prima della `pthread_create`, e che viene modificata dai due thread quando girano in pseudoparallelo
- Esaminare i programmi `t2.c` e `t2a.c`
  - Qual è l'ordine di schedulazione delle thread?
- Eseguire il programma `race.c`
  - Incrementando considerevolmente il valore passato da linea di comando, cosa capita? Perchè?
- **(a casa)** Implementare il programma `alarm_fork.c` (basato su processi) usando le thread
  - Dovete passare alle thread due argomenti:
    - il numero di secondi da attendere
    - Il messaggio

# Parte II - Semafori

---

# Semafori Posix

- Lo standard POSIX definisce i semafori *named* e *unnamed*: vediamo solo quelli unnamed.
- Un semaforo *unnamed* si dichiara così:

```
sem_t s; // tipo sem_t definito in semaphore.h  
si deve includere l'header e compilare con -pthread
```

- Si inizializza tramite la funzione `sem_init`:

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

- dove:
  - `sem` è il puntatore alla variabile dichiarata
  - `pshared` indica se il semaforo è condiviso tra thread di un processo o tra processi diversi
  - `value` è il valore a cui viene inizializzato il semaforo

# Schemi d'uso

- Perciò per inizializzare un semaforo **s** condiviso tra thread:

```
sem_init(&s, 0, val);
```

- dove **val** è il valore a cui viene inizializzato
- il secondo parametro **0** serve ad indicare che il semaforo può essere usato solo dai thread del processo che l'ha creato

- Mentre per inizializzare un semaforo **s** condiviso tra processi:

```
sem_init(&s, 1, val);
```

- dove **s** *deve essere una variabile allocata in un segmento di memoria condivisa*

# Semafori Posix: Up e Down

---

- L'equivalente della funzione "up" è
  - `sem_post(&s);`
- e l'equivalente della "down" è
  - `sem_wait(&s);`

# Mutex

Esistono inoltre i mutex: una versione semplificata di semafori binari (hanno solo due stati) su cui sono ammesse solo le operazioni **lock** e **unlock**

- Dichiarazione e inizializzazione di un mutex:
  - `pthread_mutex_t m; // tipo sem_t definito in semaphore.h`
- Inizializzazione del mutex nella condizione unlocked:
  - `pthread_mutex_init(&m,NULL); // default attributes`
- Lock del mutex (down):
  - `pthread_mutex_lock(&m);`
- Unlock del mutex (up):
  - `pthread_mutex_unlock(&m);`

# Esercizi

---

- Considerare il programma `race.c` con thread (in Lab5) e risolvere il problema di corsa critica tramite semafori Posix
- Considerare il programma `pc_sem_thr.c` (in Lab5/P-C) che risolve il problema di N produttori e un consumatore
  - Aggiungete due variabili che contino il numero di item inseriti e consumati e verificate se tutti gli item prodotti sono consumati.



UNIVERSITÀ DEL PIEMONTE ORIENTALE

# Sistemi Operativi 1

## Laboratorio 5

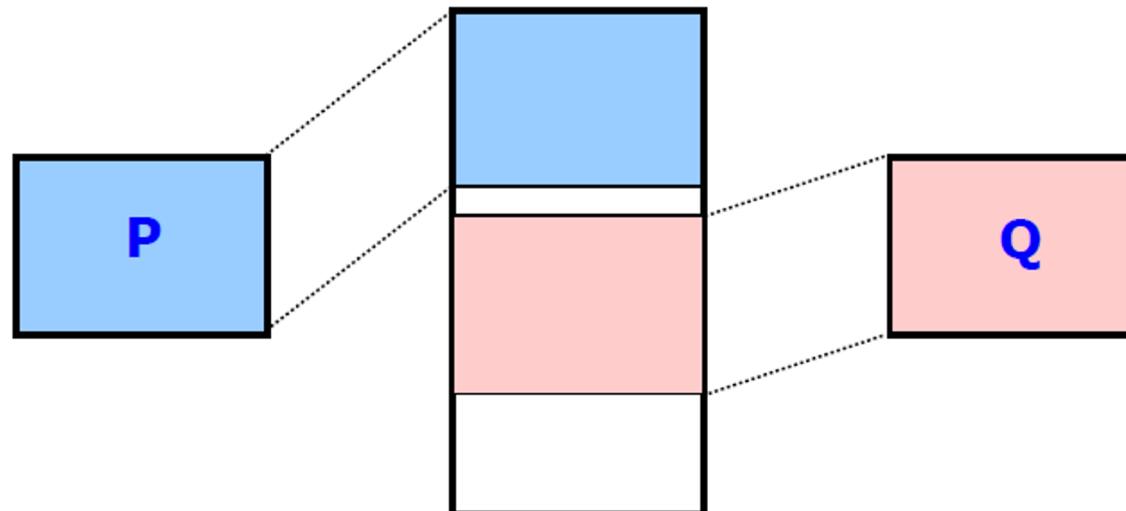
**Prof. Luisa Barrera León**

# Parte I

---

# Memoria Condivisa

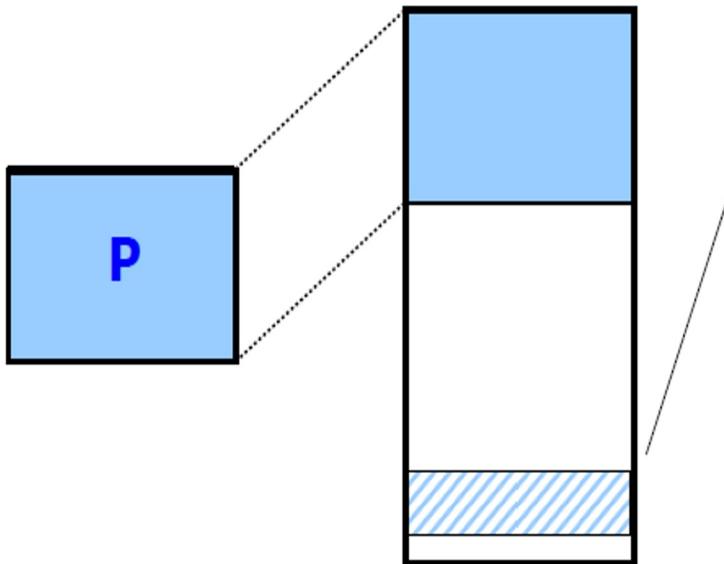
- La idea è che tra la memoria RAM una certa porzione venga usata per essere condivisa tra più processi.
- Qui assumiamo, come meccanismo di base, uno semplice in cui l'immagine di ogni processo è allocata interamente in memoria, ed è allocata in modo contiguo, cioè in un intervallo di indirizzi della RAM.



# Memoria Condivisa

Nello standard POSIX si può creare un'area di memoria condivisibile fra processi con:

- `shm_fd = shm_open(name, O_CREAT | O_RDWR, flags);`
- `ftruncate(shm_fd, size);`



Si ha uno «shared memory object»:

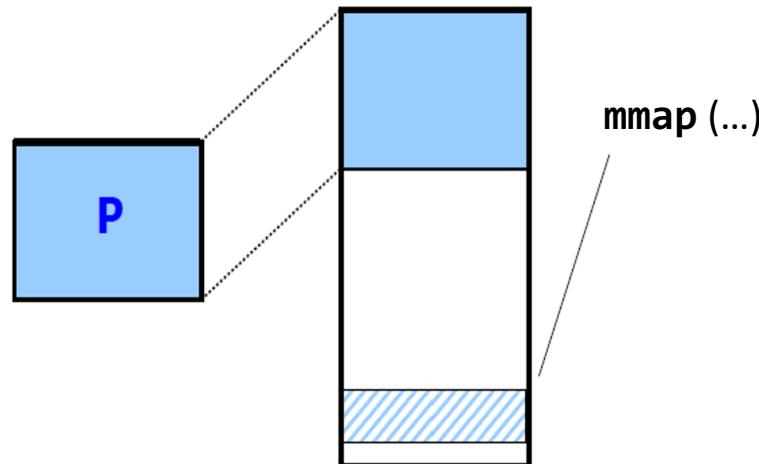
- Identificato nel sistema da **name**, stringa che inizia con uno “/” e non ne contiene altri nel processo
- Identificato da **shm\_fd** che è un file descriptor (!)
- Usando **O\_CREAT** viene creato se non esiste, se no solo «aperto».
- con diritti di accesso dati da **flags**
- di dimensione **size**

# Memoria Condivisa

- il segmento può poi essere “collegato” allo spazio di indirizzamento di P con:

```
p = mmap(..., size, ..., MAP_SHARED, shm_fd, ...)
```

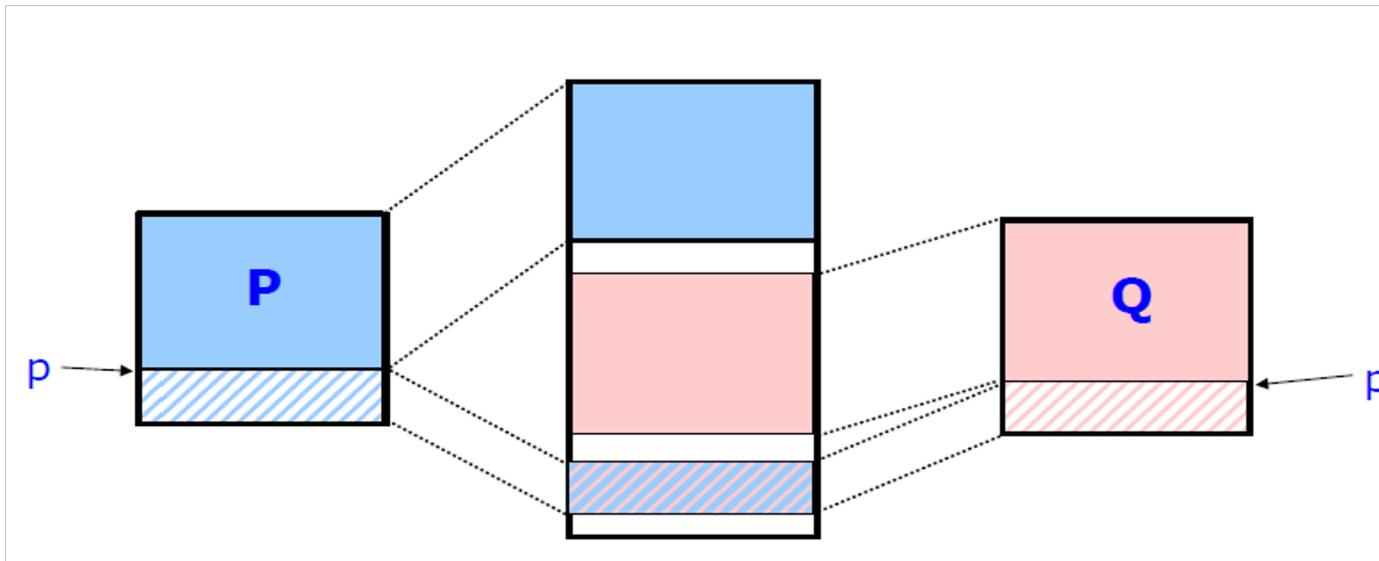
- mmap** restituisce un puntatore, come malloc



# Memoria Condivisa – Tra Processi

Come si fa? (dipende)

1. Creando nuovi processi con **fork()** dopo **mmap** abbiamo effettivamente memoria condivisa tra il processo e i suoi discendenti.
2. Processi non «parenti» in questo senso possono invece ripetere le operazioni usando lo stesso **name**(il primo che arriva «crea» la memoria condivisa, gli altri vi si collegano solo).



# Esempio di Utilizzo

```
int shm_fd1;  
  
shm_fd1 = shm_open("/myshm1", O_CREAT|O_RDWR, 0600);  
  
ftruncate(shm_fd1, 100*sizeof(int));  
  
a1 = mmap(0 ,100*sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, shm_fd1,0);
```

Creazione, se non esiste

Permessi di lettura e scrittura al proprietario

Specifica dimensione in byte del segmento

L'indirizzo a cui attaccare il segmento viene scelto dal sistema.  
Scelta con maggiore portabilità (perché si può anche inviare il puntatore)

Specifica operazioni permesse

Specifica condivisione segmento con più processi

# Parte II

---

# Variabili d'ambiente

- Le variabili d'ambiente permettono di memorizzare informazioni utili per la configurazione dei programmi
  - Sono variabili specifiche per ogni processo
  - Sono accessibili tramite API fornite dal SO
  - Un processo *figlio* può ereditarne una copia dal processo *padre* che lo ha generato
- Nella shell possono essere
  - Elencate tramite il comando env
  - Definite con
    - <NOME\_VAR>=<VALORE> - in tal caso sono definite solo per il processo corrente (la shell)
  - Esportate ai processi figli con
    - export <NOME\_VAR>
  - Lette con
    - \$<NOME\_VAR>

# Variabili d'ambiente

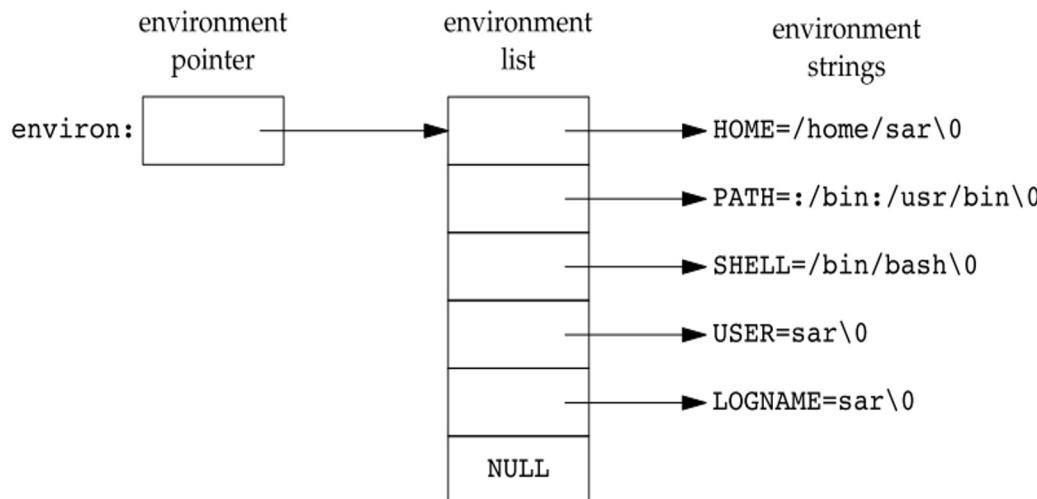
---

- Alcune variabili d'ambiente di sistema:

- **HOSTNAME** il nome del computer
- **HOME** la home directory dell'utente corrente
- **PATH** il percorso di ricerca per i comandi
  - lista di directory separate da ; in cui la shell ricerca i comandi
- **PWD** la directory attuale

# Variabili d'ambiente

- Per operare sulle variabili d'ambiente in un programma C è sufficiente dichiarare:
  - `extern char **environ;`
  - La specifica `extern` indica al compilatore che la variabile è definita in altro file
  - Come per il passaggio di argomenti è il SO che si occupa di settare la variabile `environ` all'inizio del programma



# Variabili d'ambiente: lettura e setting

- Per leggere le variabili d'ambiente:
  - `char *getenv(const char *name);`
  - Restituisce il puntatore al valore della variabile associata a *name*, *NULL* se non esiste;
- Per settare una variabile d'ambiente:
  - `int putenv(char *str);`
  - Prende una stringa di tipo *nome=valore* e l'aggiunge alla lista delle variabili d'ambiente

