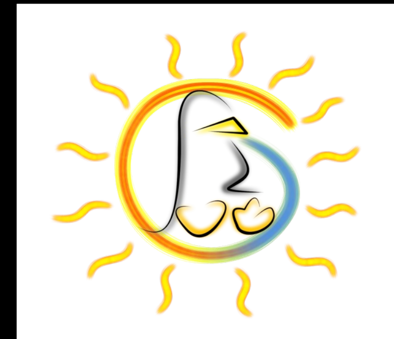
# POSIX Systems Programming

## Syscall e fondamenti della programmazione in ambienti POSIX

By lord_dex
f.apollonio@salug.it

ZEI e Salug! presentano:

geek evening 0x0d

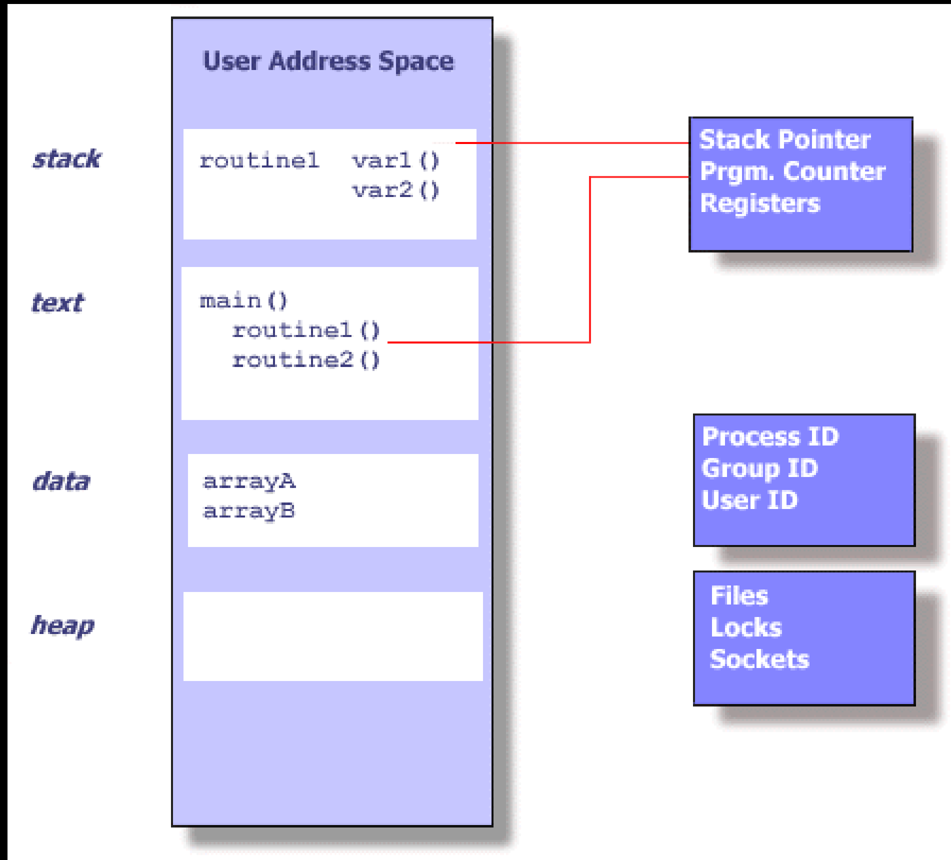


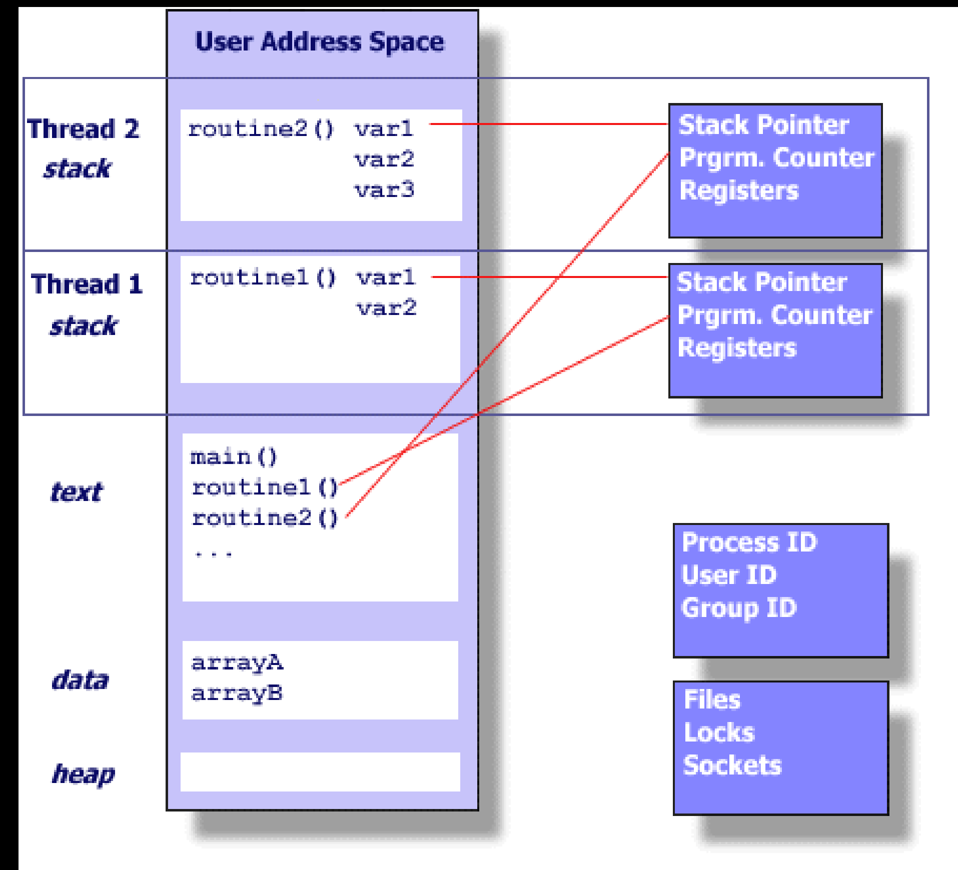
# INDICE


- Processi e Thread
- Creazione ed utilizzo di processi concorrenti
- File System e strutture di sistema
- Primitive di accesso al File System e per la gestione dei File Descriptor
- Sincronizzazione tra processi
- Gestione dei segnali
- Trasferimento dati tra processi concorrenti

# PROCESSI E THREAD

- Il **processo** è l'unità di esecuzione all'interno del SO
- Un SO multiprogrammato permette l'esecuzione concorrente di più processi
- Un processo è un insieme di risorse: PC, registri, stack, dati, file aperti, socket, etc...




- Un **thread** è un'unità di esecuzione che condivide codice e dati con altri thread ad esso associati
- Un thread è un insieme di: PC, registri, stack, etc...
- Il codice, i dati, file aperti e l'heap sono invece associati ad un gruppo di thread chiamato task

## PROCESSI E THREAD: Primitive

In sistemi POSIX i thread sono stati standardizzati solo nel 1995 (IEEE POSIX 1003.1c standard) ma frequentemente vengono utilizzate versioni proprietarie.
Per altre informazioni: https://computing.llnl.gov/tutorials/pthreads/

Molto più utilizzate sono invece le librerie standard per la gestione dei processi "pesanti".

Syscall fondamentali per la gestione dei processi:

- pid_t fork(void);

- int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
- int execv(const char *path, char *const argv[]);
- int execle(const char *path, const char *arg0, ... /*, (char *)0, char *const envp[]*/);
- int execve(const char *path, char *const argv[], char *const envp[]);
- int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
- int execvp(const char *file, char *const argv[]);

- pid_t getpid(void);
- pid_t getppid(void);
- pid_t wait(int *stat_loc);
- void exit(int status);

## PROCESSI: fork()

pid_t fork(void);

- La fork() crea un nuovo processo figlio di quello chiamante, padre e figlio condividono lo stesso codice.
- Il figlio eredita una copia dei dati del padre (file aperti, socket, etc...).

- Il valore di ritorno della funzione è un intero rappresentante il pid del figlio (se lo si utilizza dal padre) oppure 0 (se lo si utilizza dal figlio).

Utilizzando il valore di ritorno si può in questo modo differenziare il codice che deve essere eseguito dal padre e quello del figlio.

```
int pid = fork();
if (pid==0) {
        /* codice eseguito dal figlio */
} else {
        /* codice eseguito dal padre */
}
```

A questo punto si possono avere azioni concorrenti oppure il padre può attendere la terminazione del figlio prima di continuare (wait).

## PROCESSI: wait()

int wait(int *status);

- La wait() permette al padre di attendere la terminazione del figlio
- Ogni processo quando termina l'esecuzione invia un segnale SIGCHLD (vedremo dopo i segnali) al padre ed entra in uno stato detto di Zombie.
- Fino a quando il padre non cattura il SIGCHLD il figlio continuerà ad essere nello stato di zombie.

Quindi è FONDAMENTALE, prima di terminare il padre, catturare tutti i SIGCHLD dei figli.

```
kekko@laptop : ~ $ ./prova
Aspetto la terminazione del figlio 18381!

kekko    18380  0.0  0.0   1560   348 pts/0    S+   14:27   0:00 ./prova
kekko    18381  0.0  0.0   1556   140 pts/0    S+   14:27   0:00 ./prova
```

```
kekko@laptop : ~ $ kill -9 18381 && ps p 18381
  PID TTY      STAT   TIME COMMAND
18381 pts/0    Z+     0:00 [prova] <defunct>
```

La wait() richiede come parametro un intero passato per riferimento dentro al quale andrà a salvare la causa della terminazione del figlio (guardare sys/wait.h per maggiori informazioni).
L'intero ritornato dalla funzione rappresenta il pid del processo del quale viene catturato il SIGCHLD.

## PROCESSI: exec()

La famiglia di primitive exec vengono utilizzate per rimpiazzare codice, stack, heap e dati globali di un processo per differenziarlo in tutto e per tutto da quello del padre.
**NON** vengono generati nuovi processi.

- **L** -> gli argomenti da passare al programma da caricare vengono specificati mediante una LISTA di parametri (terminata da NULL) – execl()

- **P** -> il nome del file eseguibile specificato come argomento della system call viene ricercato nel PATH contenuto nell'ambiente del processo – execlp()

- **V** -> gli argomenti da passare al programma da caricare vengono specificati mediante un VETTORE di parametri – execv()

- **E** -> la system call riceve anche un vettore (envp[]) che rimpiazza l'environment (path, direttorio corrente, ...) del processo chiamante – execle()

```
pid = fork();
if (pid == 0){ /* figlio */
        printf("Figlio: esecuzione di ls\n");
        execl("/bin/ls", "ls", "-l", (char *)0);
        perror("Errore in execl\n");
        exit(1); }
if (pid > 0){ /* padre */
        printf("Padre ....\n");
        exit(0); }
```

# PROCESSI: getpid(), getppid(), exit()

```
pid_t    getpid(void);
pid_t    getppid(void);
void     exit(int status);
```

```
PADRE: getpid() = 5773, pid del figlio 5774!
FIGLIO: getpid() = 5774, getppid() = 5773
```

- int getpid(): ritorna un intero corrispondente al pid del processo invocante
- int getppid(): ritorna il pid del processo padre
- void exit(int errno): termina il processo ritornando come valore di uscita errno

# FILE SYSTEM E STRUTTURE DI SISTEMA

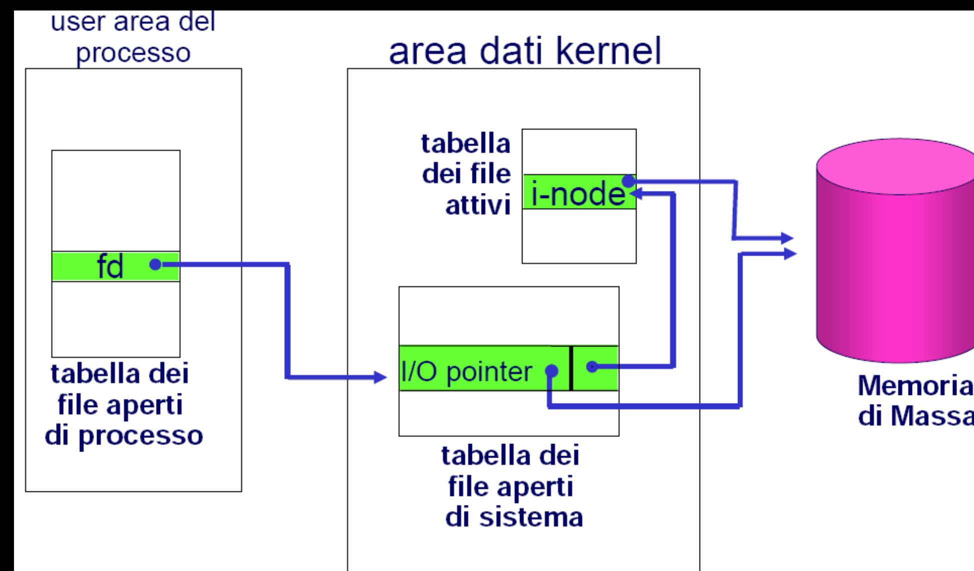
A livello di sistema esistono alcune strutture in cui son contenute le informazioni sui file aperti:

A livello kernel:

• **Tabella dei file attivi**: a livello di SO, contiene l'i-node di ogni file aperto.
• **Tabella file aperti di sistema**: in area SO, contiene un puntatore all'i-node corrispondente nella tabella dei file attivi e la posizione all'interno del file per ogni sessione di accesso a file nel sistema.

A livello di processo:

• **Tabella file aperti di processo**: è localizzata nella user area del processo e contiene un indice (file descriptor) ed un puntatore dalla voce corrispondente nella tabella file aperti di sistema.



## PRIMITIVE FILE SYSTEM

Vi sono decine e decine di syscall per la gestione del filesystem, la libreria principale è fcntl.h

Syscall fondamentali sono:

- int open(const char *path, int oflag, ... );

- ssize_t read(int fildes, void *buf, size_t nbyte);
- ssize_t write(int fildes, const void *buf, size_t nbyte);

- int close(int fildes);
- int dup(int fildes);
- off_t lseek(int fildes, off_t offset, int whence);
- int unlink(const char *path);

## PRIMITIVE FILE SYSTEM: open()

```
int open(const char *path, int oflag [, int mode]);
```

- Path è una stringa rappresentante il path assoluto del file.
- Il secondo parametro è uno (o più) interi rappresentanti il metodo di accesso al file, sono definiti alcuni alias nella libreria fcntl.h:

O_RDONLY, O_WRONLY, O_APPEND, O_CREAT, O_TRUNC

Questi ultimi due devono essere concatenati ad uno di quelli precedenti tramite il carattere | (pipe).

- O_CREAT necessita di un ulteriore parametro intero che corrisponde ai bit di protezione del nuovo file creato.

Il valore di ritorno della funzione è il file descriptor associato al file o -1 in caso di errore.

```
int fdO = open(argv[1], O_APPEND | O_CREAT, 0666);
```

## PRIMITIVE FILE SYSTEM: write() e read()

ssize_t write(int fildes, const void *buf, size_t nbyte);

- filedes è un intero rappresentante il file descriptor associato al file.
- Il secondo parametro è il valore da scrivere sul file passato per riferimento.
- Il terzo parametro è il numero di byte del buffer da scrivere sul file.
- Ritorna il numero di byte effettivamente scritti.

ssize_t read(int fildes, const void *buf, size_t nbyte);

- filedes è un intero rappresentante il file descriptor associato al file.
- Il secondo parametro è il buffer in cui andranno scritti i byte letti dal file
- Il terzo parametro è il numero di byte da leggere dal file.

Queste due primitive (write e read) sono definite **atomiche**.

```
int pid = getpid();
int fd = open(“file.dat”, O_WRONLY);
write(fd, &pid, sizeof(int));
int fd2 = open(“file.dat”, O_RDONLY);
read(fd2, &pid, sizeof(int));
close(fd); close(fd2);
```

## PRIMITIVE FILE SYSTEM: close(), dup(), lseek()

int close(int fildes);

- filedes è un intero rappresentante il file descriptor associato al file.
- La primitiva ritorna un valore uguale a 0 se il file è stato chiuso correttamente.

int dup(int fildes);

- Duplica il file descriptor passato come parametro nella prima posizione libera della tabella dei file aperti di processo.
- filedes è un intero rappresentante il file descriptor da copiare.
- La primitiva ritorna il nuovo file descriptor.

off_t lseek(int fildes, off_t offset, int whence);

- Setta l'offset per il file descriptor passato come primo argomento
- Offset è di quanti byte deve essere incrementata/decrementata la posizione all'interno del file
- Whence è la posizione da cui iniziare sommare/sottrarre i byte specificati con il parametro precedente. Le opzioni possibili sono: SEEK_SET (inizio del file), SEEK_CUR (posizione corrente all'interno del file), SEEK_END (fine del file)

int unlink(const char *path);

- Elimina il link ad un file specificato come path; se il link counter è uguale ad uno elimina fisicamente dal file system il file.
- La primitiva ritorna un valore uguale a 0 se è stato eliminato correttamente il file.

## SINCRONIZZAZIONE TRA PROCESSI

- Processi concorrenti possono richiedere determinati livelli di sincronizzazione per lavorare correttamente (es. lettura/scrittura)
- In sistemi POSIX si utilizzano delle interrupt software: i **segnali**.
- Alla ricezione di un segnale il processo può gestirlo con un **handler** definito dall'utente, ignorare il segnale o eseguire l'azione di default associata dal sistema operativo.

| | | | |
|---|---|---|---|
| SIGHUP | Hangup | POSIX | T |
| SIGINT | Interrupt | ANSI | T |
| SIGQUIT | Quit | POSIX | A |
| SIGILL | Illegal instruction | ANSI | A |
| SIGABRT | Abort | ANSI | A |
| SIGFPE | Floating-point exception | ANSI | A |
| SIGKILL | Kill, unblock-able | POSIX | T |
| SIGSEGV | Segmentation violation | ANSI | A |
| SIGPIPE | Broken pipe | POSIX | T |
| SIGALRM | Alarm clock | POSIX | T |
| SIGTERM | Termination | ANSI | T |
| SIGUSR1 | User-defined signal 1 | POSIX | T |
| SIGUSR2 | User-defined signal 2 | POSIX | T |
| SIGCHLD | Child status has changed | POSIX | I |
| SIGTSTP | Terminal stop signal. | POSIX | S |
| SIGCONT | Continue if stopped. | POSIX | C |

T=TERMINATION; A=TERMINATION; I=IGNORE; S=STOP; C=CONTINUE;
**N.B.** L'azione del segnale SIGKILL (9) e SIGSTOP (19) non è modificabile.

## GESTIONE DEI SEGNALI: PRIMITIVE

Le primitive sono molte, è importante includere la libreria signal.h dove si trovano (tra le altre cose) gli alias dei segnali.

Syscall fondamentali sono:

- handler_t signal(int sig, handler_t handler);
- int kill(int pid, int sig);
- unsigned int sleep(unsigned int N);
- int pause(void);
- pid_t wait(int *stat_loc);

# GESTIONE DEI SEGNALI: signal()

handler_t signal(int sig, handler_t handler);

- La funzione associa un gestore definito dall'utente al segnale specificato.
- Sig è il segnale da gestire
- Handler è la funzione da associare al segnale sig per la sua gestione oppure SIG_IGN per ignorare il segnale o SIG_DFL per impostare l'azione di default.
- Ritorna un puntatore alla funzione precedentemente associata come handler.

```
kekko@laptop : ~/geekevening $ ./signal &
[1] 5801
kekko@laptop : ~/geekevening $ kill -10 5801
Catturato Segnale SIGUSR1 10!
[1]+  Done                    ./signal
kekko@laptop : ~/geekevening $ ps p 5801
  PID TTY      STAT   TIME COMMAND
```

## GESTIONE DEI SEGNALI: kill(), sleep(), pause()

```
int kill(int pid, int sig);
```

• La funzione invia il segnale specificato dal parametro sig.
• Se il pid = 0 manda il segnale a tutti i processi con lo stesso gid del processo invocante, se il pid=-1 invia il segnale a tutti i processi, se il pid<-1 invia il segnale a tutti i processi con gid uguale al valore assoluto del pid passato come argomento.
• Ritorna 0 se l'invio è riuscito -1 altrimenti.

```
unsigned int sleep(unsigned int N);
```

• La funzione manda in sleep il processo per N secondi o fino alla ricezione di un qualunque segnale.
• Se la funzione viene stoppata prima della terminazione del timer viene ritornato il valore N – tempo effettivo di sleep.

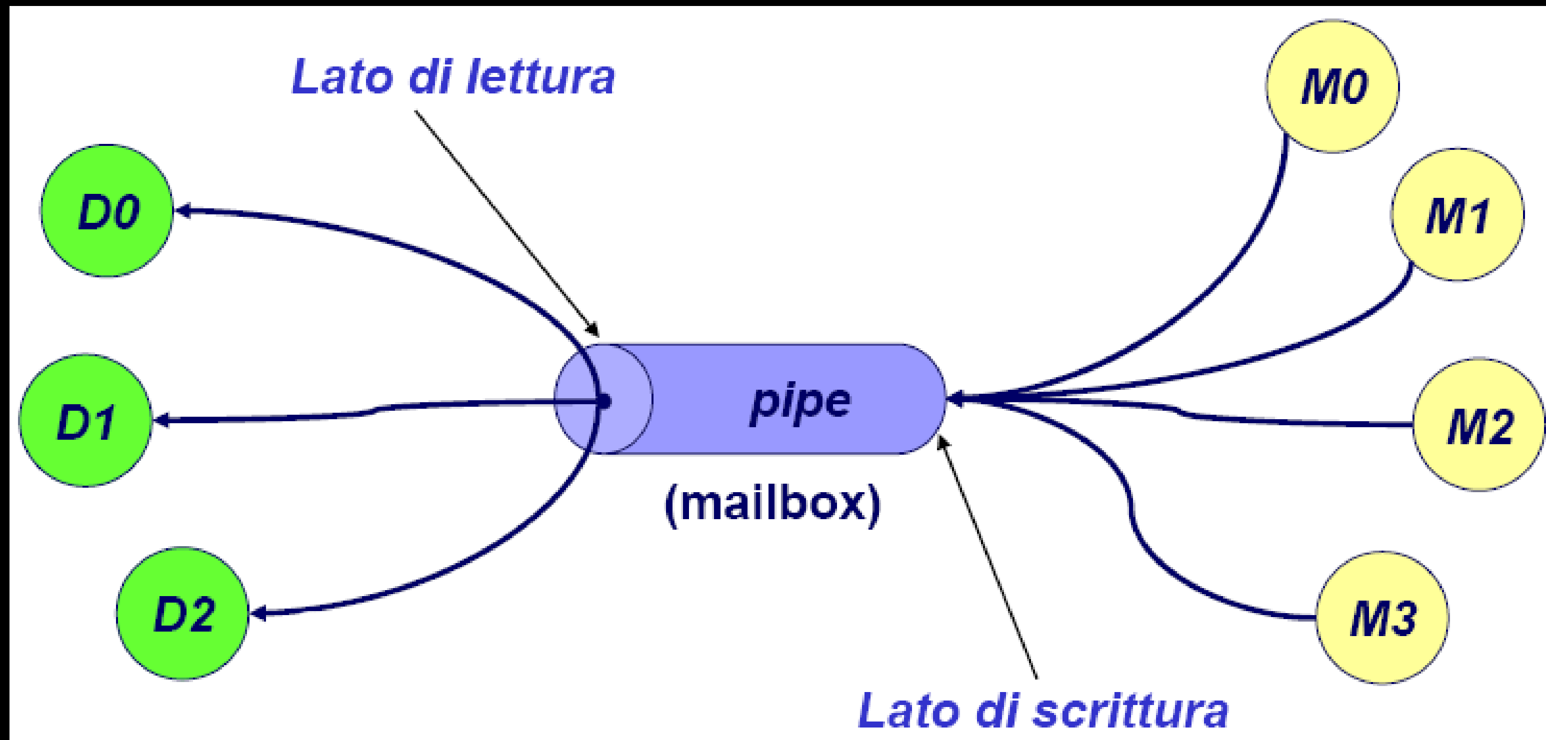
```
int pause(void);
```

• La funzione manda in pausa il processo fino alla ricezione di un qualsiasi segnale.
• Ritorna -1 se c'è qualche errore.

## TRASFERIMENTO DATI TRA PROCESSI CONCORRENTI

Come abbiamo detto i processi non hanno aree di memoria condivisa.
È però spesso necessario condividere tra diversi processi concorrenti alcune informazioni (valore di una variabile, pid, etc...).
In ambienti POSIX si utilizzano le pipe, aree di memoria a livello kernel che i processi possono utilizzare per la comunicazione.

- I processi comunicanti tramite pipe devono avere un antenato in comune!
- Si comportano esattamente come le socket.



# TRASFERIMENTO DATI TRA PROCESSI CONCORRENTI: pipe(), mkfifo()

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

- Richiede un array di due interi nel quale verranno salvati il nodo di lettura (0) e quello di scrittura(1)
- Ritorna un valore negativo in caso di errore

```
int mkfifo(char* pathname, int mode);
```

- Crea una coda FIFO nel path specificato con i permessi specificati tramite il paramentro mode.
- Al contrario delle pipe tutti i processi possono leggere/scrivere da questa coda.
- La si utilizza come un normalissimo file, open(), close(), unlink()

## BIBLIOGRAFIA:


- Manpages POSIX
- http://lia.deis.unibo.it/Courses/sola0607-info/
- http://en.wikipedia.org