INCLUDE UTILI

#include <arpa/inet.h>

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <signal.h>
#include <string.h>
#include <sys/stat.h>
#include <errno.h>
#include <time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <malloc.h>
#include <fcntl.h>
```

FILE

Apre un file in diverse modalità

Chiude File directory

int **open** (const char *pathname, int flags, mode_t mode); int **close** (int fd);

int **dup** (int oldfd); int **dup2** (int oldfd, int newfd);

Restituisce il numero di byte che sono stati letti da Fd

File descriptor buffer di count byte

int **read** (int fd, void *buf, size_t count); int **write** (int fd, void *buf, size_t count);

Restituisce il numero di Byte che sono stati scritti in Fd, o "-1" se c'è stato un errore

Imposta la posizione corrente di lettura e scrittura in corrispondenza del valore di offset All'interno del Fd

off_t **lseek** (int fd, off_t offset, int whence);

Restituisce informazioni sul File referenziato da Fd, salva il risultato nel buf

int **stat** (const char *filename, struct stat *buf); int **fstat** (int fd, struct stat *buf);

È l'opposto di Link() che rimuove un hard link

int **unlink** (const char *filename); //Delete file (solo se è l'ultimo descrittore)

GESTIONE PROCESSI

Un processo padre crea un processo figlio chiamando la primitiva fork()

int **fork** (void); void **exit**(int status); — chiude tutti i File aperti

es.

```
if ( fork() == 0 ){ /* Codice eseguito dal figlio */ }
else { /* Codice eseguito dal padre */ }
```

Restituisce il PID del processo corrente

pid_t **getpid**(void); pid_t **getppid**(void); → Restituisce il pid del processo padre del processo corrente

Attende la terminazione di ciascun processo figlio

pid_t **wait** (int *status); **waitpid** (pidfiglio, &status, NULL); — Sospende l'esecuzione del processo chiamante, finché il processo figlio identificato da pidFiglio termina

Creazione e modifica di nuovi processi, guarda ES: 4

```
int execl ( const char *path, const char *arg, ...);
int execlp ( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char * const envp[]);
/*UTILIZATE*/
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

SEGNALI

Gestione NON AFFIDABILE dei segnali

Identifica il comportamento di un processo alla ricezione di uno specifico segnale.

void ***signal** (int signumber, void (*handler))

int **kill** (pid_t pid, int sig); ← invia un segnale ad un processo PID il segnale viene definito in <signal.h> "SIGUSR1 → 10 in Linux"

unsigned int **alarm**(unsigned int nseconds); → Dopo n secondi invia un segnale al processo chiamante

int pause(void); /*NON UTILIZZARE, utilizzare sigsuspend*/

unsigned int **sleep**(unsigned int nseconds); → Sospende il processo per n secondi

Ritarda l'esecuzione del processo

int **nanosleep**(const struct timespec *req, struct timespec *rem);

```
struct timespec {  time_t tv_sec;
                   long  tv_nsec;  };
```

es4-5.c

Gestione AFFIDABILE dei segnali

int **sigemptyset** (sigset_t *set); → sigemptyset(&zeromask) // prepara una maschera di segnali vuota
int **sigfillset** (sigset_t *set); → sigemptyset(&action.sa_mask) // azzera tutti i Flag della maschera di segnali sa_mask, presente all'interno della struttura action *
int **sigaddset** (sigset_t *set, int signo);
int **sigdelset** (sigset_t *set, int signo);
int **sigismember** (sigset_t *set, int signo);

Un processo può esaminare o modificare la propria signal mask

int **sigprocmask** (int how, const sigset_t *set, sigset_t *oset);
/* *how:* *SIG_BLOCK* *SIG_SETMASK* *SIG_UNBLOCK* */

int **sigpending**(sigset_t *set);

Primitiva fondamentale →

int **sigaction**(int signo, const struct sigaction *act, const struct sigaction *oact);
struct sigaction {
void (***sa_handler**)(); /* indirizzo del gestore o SIG_IGN o SIG_DFL */ // assegna la funzione da chiamare alla ricezione del segnale
void (***sa_sigaction**)(int, siginfo_t *, void *);
* sigset_t **sa_mask**;
int **sa_flags**; /* SA_NODEFER per ricevere più segnali uguali senza blocco*/
/* *SA_RESTART per far ripartire il programma da dove si era fermato*/
};

int **sigsuspend**(const sigset_t *sigmask) // attendiamo la ricezione di un segnale

PIPE => (es 7-8-9 esercitaz. 5)

int **pipe**(int fd[2]); // Apre la pipe creando un file descriptor, uno per la lettura e l'altro per la scrittura.

/* fd[0] lettura; fd[1] scrittura */
/*Dalla pipe si legge e scrive con le read e write sopra citate. La read fatta su una pipe fa scomparire per tutti gli altri processi il dato presente. La grandezza massima della pipe è PIPE_BUF = 4096 bytes*/

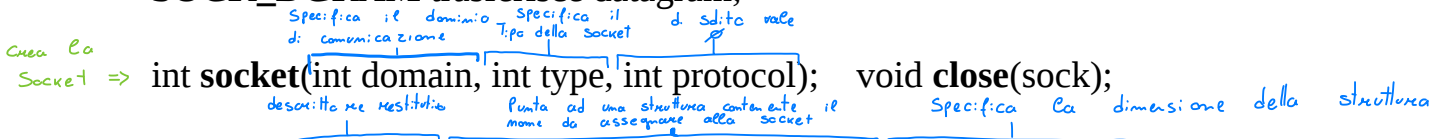
FIFO

int **mkfifo** (const char *pathname, mode_t mode);

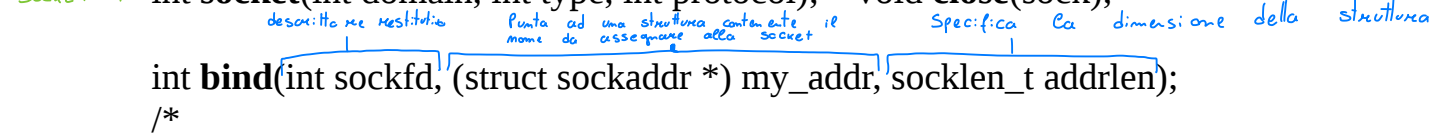
Vanno utilizzate le normali SysCall che si usano per file e pipe (open, read/write, close,.)
Una FIFO deve essere aperta con open dopo che è stata creata con mkfifo.

SOCKET

Le socket utilizzate sono:
SOCK_STREAM orientata alla connessione, trasferisce byte stream. Attenzione allo spacchettamento dei file letti;
SOCK_DGRAM trasferisce datagram;



int **socket**(int domain, int type, int protocol); void **close**(sock);



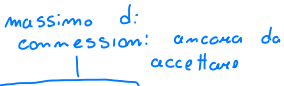
int **bind**(int sockfd, (struct sockaddr *) my_addr, socklen_t addrlen);
/*
on = 1; rval = **setsockopt**(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
*/

struct **sockaddr_in** {
sa_family_t sin_family; /* address family: AF_INET */
u_int16_t sin_port /* port in network byte order */
struct in_addr sin_addr; /* internet address, anche questa è una struttura “s_addr” */
};

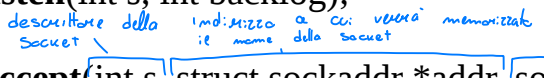
Client

int **connect**(int sockfd, (const struct sockaddr *) serv_addr, socklen_t addrlen);
/* E poi usare le write e read */

Server



int **listen**(int s, int backlog);



int **accept**(int s, struct sockaddr *addr, socklen_t *addrlen);

/*SERVER NON CONCORRENTE: server che gestisce le richieste dei client in modo seriale, quando una richiesta viene soddisfatta può essere effettuata un’altra;
SERVER CONCORRENTE: server che crea un figlio ad ogni richiesta di connessione dei client, perciò può sfruttare la funzionalità di gestire più richieste in parallelo*/

Socket datagram (senza listen, accept e connect)**:**

int **sendto** (int s, const void *msg, size_t len, int flags,const struct sockaddr *to, socklen_t tolen);
int **recvfrom**(int s, void *buf, size_t len, int flags, structsockaddr *from, socklen_t *fromlen);

SELECT

int **select**(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

FD_ZERO (fd_set *set) //azzera un fd_set
FD_CLR (int fd, fd_set *set) //rimuove un fd da un fd_set
FD_SET (int fd, fd_set *set) //inserisce un fd in un fd_set
FD_ISSET (int fd, fd_set *set) //predicato che verifica se un certo
//fd è membro di un fd_se