----------> FILE

/\* aprire \*/

fds = open("./nome\_file", O\_CREAT | O\_WRONLY | O\_RDONLY | O\_TRUNC | O\_APPEND, 0777);

/\* posizionarsi all'interno del file\*/

lseek(fds, 1, SEEK\_CUR); /\*Si posiziona alla fine del file\*/

/\* leggere \*/

read(fds, &buff, sizeof(char));

/\* scrivere \*/

write(fdd, &buff, sizeof(char));

/\* chiudere \*/

close(fds);

/\* Effettua una copia di fds sul file descriptor fd2\*/

fd2 = dup (fds);

/\* Duplica il canale 3 sul canale 2 \*/

dup2 (3, 2);

----------> DIR FILE PERMITS

/\* posizione corrente \*/

char \*cwd=getcwd(cwd,MAX\_PATH);

/\* cambio cartella lavoro \*/

chdir("/home/aulainfo/Scrivania/new\_folder");

/\* controllo permessi lettura|scrittura|esecizione|esistenza \*/

bool flag = access("./file.c", R\_OK | W\_OK | X\_OK | F\_OK);

/\* annulla i permessi che ci sono come argomento \*/

umask(S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH);

/\* crea un file generico \*/

creat("foo", S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH)

/\* aprire directory \*/

DIR \*cartella = opendir("/home/aulainfo/Scrivania/new\_folder");

/\* usata sequenzialmente legge i contenuti della cartella \*/

struct dirent \*element = readdir(cartella);

/\* usata identifica le info nella stat \*/

struct stat info;

lstat(path,&info);

----------> PROCESS FORK

/\* creare processo (vfork lo esegue prima del padre) \*/

int childpid= fork();

/\* attesa di un processo \*/

waitpid(pid\_process,&status,0);

/\* controllo che sia terminato normale \*/

bool flag = WIFEXITED(status)

/\* importo lo stato uscita del processo \*/

WEXITSTATUS(status);

/\* terminare processo \*/

exit(0);

/\* esecuzione exec con lista \*/

execl("/bin/ps", "ps", "-elf", NULL);

/\* esecuzione programma c con arg e path\*/

char \*env\_init[]={"USER=sconosciuto", "PATH=/tmp", NULL};

execle("/home/aulainfo/Scrivania/due.c", "arg 1", "arg 2", "arg 3", NULL, env\_init);

/\* stessa cosa di prima ma il path è quello usato dall'invocante\*/

execlp("/home/aulainfo/Scrivania/due.c", "arg 1", "arg 2", "arg 3", NULL);

----------> PIPE

/\* creare pipre anonima\*/

Int fd[2];

pipe(fd);

/\* leggere da una pipe \*/

close(fd[WRITE]);

(opportunamente) dup2(fd[READ],READ);

read (fd[READ], buffer, sizeof(buffer));

close(fd[READ]);

/\* scrivere da una pipe \*/

close(fd[READ]);

(opportunamente) dup2(fd[WRITE],WRITE);

write (fd[WRITE], buffer, sizeof(buffer));

close(FD[WRITE]);

----------> FIFO

/\* creare fifo \*/

mkfifo("./esempio\_fifo", 0777);

/\* aprire fifo \*/

int fifo\_fd = open("./esempio\_fifo", O\_WRONLY | O\_RDONLY);

/\* scrivere sulla fifo \*/

write(fifo\_fd, buffer, BUFFER\_SIZE);

/\* leggere sulla fifo \*/

read(fifo\_fd, buffer, BUFFER\_SIZE);

/\* chiudere fifo \*/

close(fifo\_fd);

----------> THREAD

/\* struttura thread \*/

struct struttura {

char var1;

int var2;

bool var3;

float var4;

};

/\* funzione che sarà eseguita dai thread \*/

void \*char\_print (void\* parameters) {

/\* cast del puntatore al tipo corretto essendo dato un void \*/

struct struttura \*p = (struct struttura\*) parameters;

/\* elabora i dati passati dalla struct \*/

p->var2++;

p->var4+=1.42455;

p->var3=false;

/\* temina l'esecuzione\*/

pthread\_exit((void \*) p->var2);

}

int main() {

/\* variabile identificativa dei thread\*/

/\* assegno variabili all'interno della struttura tid1\_args e tid2\_args \*/

struct struttura tid1\_args;

struct struttura tid2\_args;

tid1\_args.var1 = '0';

tid1\_args.var4 = 304.400;

tid2\_args.var2 = '1';

tid2\_args.var4 = 242.500;

/\* creo e attivo il thread1 e thread2 (notare il passaggio del parametro)\*/

pthread\_t tid[2];

pthread\_create( &tid[0] , NULL , char\_print , (void\*) &tid1\_args);

pthread\_create( &tid[1] , NULL , char\_print , (void\*) &tid2\_args);

/\* aspetto che termino l'esecuzione \*/

void \*rit1, \*rit2;

pthread\_join(tid[0], &rit1);

pthread\_join(tid[1], &rit2);

----------> MUTEX

/\* inizializza il mutex globalmente\*/

pthread\_mutex\_t mymutex = PTHREAD\_MUTEX\_INITIALIZER;

/\* per bloccare il mutex \*/

pthread\_mutex\_lock( & mymutex); // lock il mutex

/\* per sbloccare il mutex \*/

pthread\_mutex\_unlock( & mymutex); // unlock mutex

----------> VARABILI CONDIZIONE

/\* inizializza la variabile di condizione globalmente\*/

pthread\_cond\_t mycond = PTHREAD\_COND\_INITIALIZER;

/\* bloccare con la varabile di condizione \*/

pthread\_cond\_wait( &mycond, &mymutex );

/\* per sbloccare la condizione \*/

pthread\_cond\_signal( &mycond);

--------> SEMAFORI Basati su nome

sem\_t \* sem= sem\_open(nome\_semaforo, O\_CREAT | O\_EXCL, S\_IRUSR | S\_IWUSR, max\_number);

sem\_wait(sem);

sem\_post(sem);

sem\_getvalue(sem, &val);

sem\_close(sem);

sem\_unlink(nome\_semaforo);

-------->SEMAFORI Basati su memoria

sem\_init( &nome\_semaforo, 0, max\_number);

sem\_wait(sem);

sem\_post(sem);

sem\_destroy( &nome\_semaforo );

----------------> CLIENT AF\_INET

/\* defiamo il descrittore alla soket | il client possa connettersi con il server \*/

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

/\* variabile usata per dare un nome alla socket \*/

struct sockaddr\_in address;

/\* definiamo il tipo -> usando una struct sockaddr\_in è logico che sia AF\_INET \*/

address.sin\_family = AF\_INET;

/\* non ha un nome, ma necessita di sapere l'id che si riferisce\*/

address.sin\_addr.s\_addr=htonl(IP\_ADD);

/\* non ha un nome, ma necessita di sapere la porta che si riferisce\*/

address.sin\_port=htons(PORT);

/\* connettiamo la nostra socket con quella del server e controlla la connessionee\*/

connect(sockfd, (struct sockaddr\*) &address, sizeof(address));

// DA NOTARE che si usa (struct sockaddr \*) &address per rendere compatibile il tipo passato, da struct sockaddr\_IN a sockaddr (anche avendo gli stessi campi)

/\* possiamo leggere e scrivere via sockfd \*/

send(sockfd, &var, sizeof(var));

recv(sockfd, &buf, sizeof(buf));

/\* una volta finito chiude la socket \*/

close(sockfd);

/\* termina \*/

exit(0);

-----------------> SERVER AF\_INET

/\* defiamo il descrittore alla soket | il client possa connettersi con il server \*/

int server\_sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

/\* variabile usata per dare un nome alla socket \*/

struct sockaddr\_in server\_address;

/\* definiamo il tipo -> usando una struct sockaddr\_in è logico che sia AF\_INET \*/

server\_address.sun\_family = AF\_UNIX;

/\* non ha un nome, ma necessita di sapere l'id che si riferisce\*/

address.sin\_addr.s\_addr=htonl(IP\_ADD);

/\* non ha un nome, ma necessita di sapere la porta che si riferisce\*/

address.sin\_port=htons(PORT);

/\* l'equivalente della connect del client ma in questo caso la sto creando non connettendo \*/

bind(server\_sockfd, (struct sockaddr \*) &server\_address, sizeof(server\_address));

---> ACCETTARE CONNESSIONE

/\* possono esserci massimo 5 processi richiedenti contemporaneamente \*/

listen(server\_sockfd, 5);

/\* ovvaimente fatto tutto in un ciclo \*/

while (1) {

/\* variabile usata per dare un nome alla socket \*/

struct sockaddr\_in client\_address;

/\* identifica la lunghezza della socket \*/

client\_len = sizeof(client\_address);

/\* accetta la connessione proveniente dalla socket server\_sockfd, e utilizza i canale di comunicazione client\_address avente lunghezza client\_len per connettersi e "parlare" con il client \*/

int client\_sockfd = accept(server\_sockfd, (struct sockaddr \*) &client\_address, &client\_len);

/\* possiamo leggere e scrivere via sockfd \*/

write(sockfd, &var, sizeof(var));

read(sockfd, &buf, sizeof(buf));

close(client\_sockfd); // chiude la socket aperta per il client

}

/\* una volta finito chiude la socket \*/

close(sockfd);

/\* termina \*/

exit(0);

----------------> CLIENT AF\_UNIX

/\* defiamo il descrittore alla soket | il client possa connettersi con il server \*/

int sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);

/\* variabile usata per dare un nome alla socket \*/

struct sockaddr\_un address;

/\* definiamo il tipo -> usando una struct sockaddr\_un è logico che sia AF\_UNIX \*/

address.sun\_family = AF\_UNIX;

/\* nome della socket che si utilizza per la connessione al server, ovviamente il server deve avere lo stesso \*/

strcpy(address.sun\_path, "./server\_socket");

/\* connettiamo la nostra socket con quella del server e controlla la connessionee\*/

connect(sockfd, (struct sockaddr\*) &address, sizeof(address));

// DA NOTARE che si usa (struct sockaddr \*) &address per rendere compatibile il tipo passato, da struct sockaddr\_UN a sockaddr (anche avendo gli stessi campi)

/\* possiamo leggere e scrivere via sockfd \*/

write(sockfd, &var, sizeof(var));

read(sockfd, &buf, sizeof(buf));

/\* una volta finito chiude la socket \*/

close(sockfd);

/\* termina \*/

exit(0);

-----------------> SERVER AF\_UNIX

/\* defiamo il descrittore alla soket | il client possa connettersi con il server \*/

int server\_sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);

/\* variabile usata per dare un nome alla socket \*/

struct sockaddr\_un server\_address;

/\* definiamo il tipo -> usando una struct sockaddr\_un è logico che sia AF\_UNIX \*/

server\_address.sun\_family = AF\_UNIX;

/\* nome della socket che si ha intenzione di dargli (il nome deve essere comune anche al client) \*/

strcpy(server\_address.sun\_path, "./server\_socket");

/\* l'equivalente della connect del client ma in questo caso la sto creando non connettendo \*/

bind(server\_sockfd, (struct sockaddr \*) &server\_address, sizeof(server\_address));

---> ACCETTARE CONNESSIONE

/\* possono esserci massimo 5 processi richiedenti contemporaneamente \*/

listen(server\_sockfd, 5);

/\* ovvaimente fatto tutto in un ciclo \*/

while (1) {

/\* variabile usata per dare un nome alla socket \*/

struct sockaddr\_un client\_address;

/\* identifica la lunghezza della socket \*/

client\_len = sizeof(client\_address);

/\* accetta la connessione proveniente dalla socket server\_sockfd, e utilizza i canale di comunicazione client\_address avente lunghezza client\_len per connettersi e "parlare" con il client \*/

int client\_sockfd = accept(server\_sockfd, (struct sockaddr \*) &client\_address, &client\_len);

/\* possiamo leggere e scrivere via sockfd \*/

write(sockfd, &var, sizeof(var));

read(sockfd, &buf, sizeof(buf));

close(client\_sockfd); // chiude la socket aperta per il client

}

/\* una volta finito chiude la socket \*/

close(sockfd);

/\* termina \*/

exit(0);