Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 2° anno - 12 CFU

Laboratorio II

Professori:
Prof. Giovanni Manzini

Autori:
Filippo Ghirardini

Anno Accademico 2023/2024

Contents

1 Funzioni specifiche

1.1 perror

Come funziona *perror*?
È una funzione che stampa un messaggio di errore su *stderr* seguito da una descrizione. Si basa sulla variabile globale *errno*.

```
void perror(const char *msg) → msg:string
```

1.2 stderr e stdout

Qual è l'utilizzo di *stderr* e *stdout*?
stdout lo usiamo per l'output, di default i dati vengono stampati sul terminare (e.g. *printf*). Per ridirezionarlo usiamo

```
command > file_output
```

stderr è usato invece per l'output degli errori o messaggi diagnostici (e.g. *perror*). Per ridirezionarlo usiamo

```
command >&2 file_output
```

1.3 fscanf

Cosa significano i modificatori *%NS* e *%ms* in *fscanf*?

- **%Ns** è un modificatore che permette di leggere d massimo $n-1$ caratteri in input, riservando il carattere finale \0.

  ```
  fscanf(file, "%Ns", buffer);
  ```

  Con *buffer* già allocato.

- **%ms** è un modificatore che permette di allocare dinamicamente la memoria necessaria per memorizzare la stringa.

  ```
  fscanf(file, "%ms", &buffer);
  ```

  Con *buffer* un puntatore a *char**. La memoria è deallocata automaticamente.

1.4 snprintf e asprintf

Allocazione di stringhe tramite *snprintf* e *asprintf*.
Con *snprintf* si costruiscono stringhe in modo sicuro scrivendole in un buffer preallocato mentre *asprintf* alloca dinamicamente la memoria.

- ```
  int snprintf(char* str, size_t size, const char* format);
  ```

  Questa funzione restituisce il numero di caratteri scritti e usa i seguenti parametri:

  – *str*: puntatore al buffer dove verrà scritta la stringa
  – *size*: dimensione massima del buffer incluso \0
  – *format*: stringa di formato
  ```

```
int asprintf(char ** strp, const char * format, ...);
```

Restituisce il numero di caratteri scritti senza \0 e prende come parametro *strp* ovvero il puntatore dove verrà allocata la memoria per la stringa.

1.5 fscanf

Quali sono le limitazioni di *fscanf*?

- È sensibile al formato, quindi se passo *%d* e poi leggo un intero mi restituisce errore
- Non c'è la gestione delle righe, ignora i caratteri *\n*
- Legge fino al prossimo spazio, può causare **troncamenti**
- Non gestisce facilmente le virgole e non verifica i limiti dei buffer

1.6 getline

Spiega la funzione *getline*.
È una funzione per leggere intere righe di testo.

```
ssize_t  getline(char ** lineptr, size_t * n, FILE* stream);
```

Dove i parametri sono:

- *lineptr*: punta alla memoria dove verrà memorizzata la riga letta. Se è *NULL* la funzione alloca dinamicamente. Se è troppo piccolo il buffer viene riallocato.
- *n*: la dimensione del buffer, viene aggiornato automaticamente se viene ridimensionato
- *stream*: puntatore al file da cui leggere

1.7 fscanf vs getline

Qual è la differenza sostanziale tra *fscanf* e *getline*?
fscanf è ottima per leggere dati formattati direttamente ma richiede attenzione ai rischi di buffer overflow. *getline* è più flessibile e sicuro per leggere righe intere ma necessita di ulteriori elaborazioni per estrarre i dati interni.

1.8 strtok

Parsing di stringhe con *strtok*.
La funzione *strtok* è usata per dividere una stringa in token separati da una o più caratteri (delimitatori).

```
char* strtok(char* str, const char* delim);
```

Dove i parametri sono:

- *str* è la stringa da dividere
- *delim* è la stringa di delimitatori, ogni carattere è un separatore tra i token

La funzione restituisce un puntatore al prossimo token o *NULL* quando non ci sono più token. Inserisce \0 dopo ogni token nella stringa originale che quindi non è più utilizzabile.

La versione thread-safe è ***strtok_r***:

```
strtok\_r(char* str, const char * delim, char ** saveptr);
```

Dove i parametri sono:

- *str* stringa da analizzare
- *delim* stringa di delimitatori
- *saveptr* puntatore che memorizza lo stato tra le chiamate

1.9 calloc

Spiega la funzione *calloc*.
La funzione *calloc* è utilizzata per allocare memoria dinamicamente. È simile a *malloc* ma con inizializza la memoria allocata a 0.

```
void* calloc(size_t nitems, size_t size);
```

- *nitems* numero di elementi da allocare
- *size* dimensione di ciascun elemento

Restituisce un puntatore alla memoria allocata.

1.10 qsort

Come funziona *qsort*? Quali sono le sue problematiche?
qsort è una funzione di libreria per ordinare un array di elementi.

```
void qsort(void* base, size_t num, size_t size, int (*compar)(const void*, const void*));
```

Dove i parametri sono:

- *base* puntatore dell'array da ordinare
- *num* numero di elementi nell'array
- *size* dimensione di ciascun elemento nell'array
- *compar* funzione di confronto personalizzata

La funzione utilizza *void **. Se il tipo non viene gestito correttamente possono verificarsi comportamenti imprevedibili. Se l'array contiene strutture la funzione *compar* deve saperli gestire correttamente.

1.11 fread e fwrite

Lettura e scrittura dei file binari con *fread* e *fwrite*.

```
size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
size_t fwrite(const void* ptr, size_t size, size_t count, FILE* stream);
```

Dove i parametri sono:

- *ptr* puntatore al buffer dove leggere o scrivere
- *size* dimensione di ciascun elemento
- *count* numero di elementi
- *stream* puntatore al file aperto

Le funzioni restituiscono il numero di elementi letti o scritti. Se è inferiore a *count* potrebbe significare la presenza di errori.

1.12 fork

Creazione di processi con *fork*.
La chiamata alla funzione *fork* genera un nuovo processo figlio duplicando il processo padre. Il padre deve sempre aspettare il figlio invocando *wait*, altrimenti esso diventerà un processo zombie (non terminabile correttamente dal SO).

```
pit_t fork(void);
```

1.13 exit e wait

Terminazione di un processo figlio con *exit* e *wait*.

- *exit*: la funzione termina l'esecuzione del processo chiamante, chiude tutti i file aperti, esegue la pulizia e restituisce un codice di uscita al SO

  ```
  void exit(int status);
  ```

- *wait*: il processo genitore può utilizzare questa funzione (o in alternativa *waitpid*) per attendere che uno o più processi figli terminino. Questo gli permette di recuperare il codice di uscita ed evitare che nascano processi zombie.

  ```
  pid_t wait(int *status);
  pid_t wait_pid(pid_t pid, int* status, int options);
  ```

  Dove i parametri sono:

  - *pid* PID del processo figlio da attendere
    - > 0 il PID specifico
    - -1 qualsiasi figlio
    - 0 qualsiasi figlio nello stesso gruppo di processi
  - *status* puntatore dove viene memorizzato il codice di uscita
  - *options* opzioni aggiuntive

1.14 execl

Spiega l'istruzione *execl*.
Serve ad eseguire un determinato file.

```
int execl(const char* path, const char* arg, ..., NULL);
```

Dove i parametri sono:

- *path* percorso del programma da eseguire
- *arg* lista di argomenti passati al programma dove il primo è convenzionalmente il nome del programma
- *NULL* la lista di argomenti deve terminare con NULL per segnalare la fine

1.15 atexit

Descrivi la funzione *atexit*.
È utilizzata per registrare funzioni da eseguire automaticamente quando il programma termina (e.g. pulizia come chiusura di un file). Vengono eseguite in ordine inverso rispetto alla registrazione.

```
int atexit(void(*func)(void));
```

1.16 getopt

Come funziona *getopt*?
È una funzione per analizzare le opzioni e gli argomenti passati dalla riga di comando.

```
int getopt(int argc, const char* argv[], const char* opstring);
```

Dove i parametri sono:

- *argc* numero di argomenti passati da riga di comando
- *argv* array di stringhe contenenti i parametri passati
- *opstring* una stringa che definisce le opzioni accettate. Ogni carattere rappresenta un'opzione, se è seguito da : vuol dire che richiede un argomento

Restituisce il carattere dell'opzione trovata, -1 quando tutte le opzioni sono state elaborate e ? se trova un'opzione non valida.

1.17 pthread_create e pthread_join

Descrivi i prototipi di *pthread_create* e *pthread_join*.

- Creazione di thread

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

  Dove i parametri sono:

  - *thread* puntatore alla variabile che conterrà il thread creato
  - *attr* specifica gli attributi del thread (NULL di solito)
  - *(*start_routine)(void)* funzione che rappresenta il punto di inizio del thread. Deve accettare void* come parametro e restituire void*
  - *arg* puntatore dell'argomento da passare alla funzione di inizio

  Restituisce 0 se il thread è stato creato con successo.

- Attesa thread

  ```
  int pthread_join(pthread_t thread, void** retval);
  ```

  Dove i parametri sono:

  - *thread* identificatore del thread
  - *retval* puntatore al valore restituito dalla funzione eseguita dal thread

  Restituisce 0 se il thread ha terminato con successo.

1.18 signal

- *signal* è utilizzato per la gestione di segnali

  ```
  void* signal(int signum, void* handler(int));
  ```

  Dove i parametri sono:

  - *signum* il codice del segnale da gestire
  - *handler* la funzione che gestisce il segnale. Deve prendere in input un intero che corrisponde al valore del segnale.

1.19 sleep

Attesa di segnali con *sleep*.
Una volta configurato il gestore di segnali, il programma può attendere utilizzando la funzione *sleep*. Infatti, non appena il kernel riceve un segnale, interrompe l'attesa e chiama l'handler.

1.20 kill

Invio di segnale dalla riga di comando con *kill*.

```
kill -signal pid
```

Dove i parametri sono:

- *signal* nome o numero del segnale da inviare al processo
- *pid* process ID a cui inviare il segnale:
  - > 0 processo con PID specifico
  - $= 0$ tutti processi nel gruppo del chiamante
  - < -1 tutti i processi nel gruppo con PID$= -$PID
  - $= -1$ tutti i processi a cui l'utente può inviare segnali

Note 1.20.0.1. Utilizzare il parametro *-l* per vedere l'elenco dei segnali e i loro numeri.

Esiste anche una funzione che può essere chiamata all'interno di un programma e invia il segnale ad un singolo thread all'interno dello stesso processo.

```
int pthread_kill(pthread_t thread, int sig);
```

1.21 sigwait

Attesa dei segnali con *sigwait*.
sigwait è una funzione che sospende il thread chiamante fino a quando uno dei segnali specificati non viene ricevuto. Il segnale viene poi rimosso dalla coda dei segnali pendenti del processo e il controllo ritorna al thread chiamante, consentendo di gestire il segnale in modo sincrono.

```
int sigwait(const sigset_t * set, int* sig);
```

Dove i parametri sono:

- *set* puntatore ad un insieme di segnali che il thread è disposto ad attendere
- *sig* puntatore dove verrà memorizzato il segnale ricevuto

Restituisce 0 se è corretto.

1.22 sigqueue e sigwaitinfo

Invio di segnali con *sigqueue* e *sigwaitinfo*.

- *sigqueue* consente di inviare segnali real time ad un thread specifico con dati personalizzati

  ```
  int sigqueue(pthread_t thread, int sig, const union signal value);
  ```

  Dove i parametri sono:

  - *thread* identificazione del thread destinatario
  - *sig* codice del segnale
  - *value* dato associato al segnale

- *sigwaitinfo* permette di attendere un segnale e ricevere le informazioni aggiuntive

```
int sigwaitinfo(const sigset_t * set, siginfo_t * info);
```

Dove i parametri sono:

  - *set* insieme di segnali da attendere
  - *info* struttura che riceve le informazioni del segnale. È di tipo *siginfo_t* che contiene:
    * *si_signo* numero del segnale
    * *si_code* codice del segnale
    * *si_value* dato personalizzato associato al segnale

1.23 syscall e library function

Qual è la relazione tra *syscall* e le lybrary functions?
Le *syscall* sono funzioni fornite dal sistema operativo per interagire direttamente con il kernel.

```
int open(const char* pathname, int flags, mode_t mode);
size_t read(int fd, void* buf, size_t count);
size_t write(int fd, const void* buf, size_t count);
```

Queste funzioni operano su un file descriptor senza buffering e con gestione manuale delle risorse.
Le *funzioni di libreria* invece offrono un livello superiore , semplificando le operazioni di lettura e scrittura grazie al **buffering**.

```
FILE* fopen(const char* pathname, const char* mode);
size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
size_t fwrite(const void* ptr, size_t size, size_t count, FILE* stream);
```

Queste operano su puntatori al file (struttura che incapsula il file descriptor e il buffer) e offrono un sistema di buffering, riducendo le chiamate al kernel.

2 Strutture dati

2.1 Array statici e dinamici

Quando bisogna evitare gli array statici?
Nei seguenti casi:

- Quando la dimensione dell'array dipende dall'**utente**
- Quando l'array deve essere **ridimensionato**
- Quando è molto **grande**
- Quando lavoriamo in **parallelo**

Differenza tra matrici statiche e dinamiche.
Una matrice statica è dichiarata con una dimensione fissa e acceduta con indici:

```
int matrice[3][4];
matrice[1][2] = 45;
```

Una matrice dinamica è allocata ad un certo punto dell'esecuzione del programma permettendo di usare dimensioni variabili. Esistono due tipi di implementazione:

- Doppio puntatore

```
int righe=3, colonne=3;
int** matrice=malloc(righe*sizeof(int*));
for(int i=0; i<righe; i++)
matrice[i] = malloc(colonne*sizeof(int));
```

- Array lineare, è migliore

```
int righe=3, colonne=4;
int* matrice = malloc(righe*colonne*sizeof(int));
```

2.2 Caratteri vs stringhe

Qual è la differenza tra caratteri e stringhe?
Un carattere è un elemento che occupa 1 byte di memoria.

```
char C = 'A';
```

Può rappresentare un simbolo o un valore del codice ASCII. La stringa è una **sequenza** di caratteri, rappresentato come array con un carattere aggiuntivo di terminazione \0.

```
char* str = "hello";
```

2.3 struct e typedef

Spiega la struct e il typedef.
Le struct sono strutture utili per raggruppare più variabili di natura diversa sotto un unico nome. Per semplificarne la definizione utilizziamo *typedef*:

```
typedef struct {
   char nome[50];
   int eta;
} Persona;
```

```
Persona p = {"Pippo", 20};
p.nome;
p.eta;

Persona p2 = malloc(sizeof(Persona));
p2->nome = "Pluto";
```

Note 2.3.0.1. Se la struct contiene array dinamici interni, questi vanno allocati e deallocati di conseguenza.

2.4 Array di struct

Come funzionano gli array di struct?
Avendo rinominato la struttura con *typedef*, per creare un array di struct:

```
Persona * array = malloc(num_struct * sizeof(Persona));
```

e posso accedervi nel seguente modo:

```
array[0].nome;
```

2.5 Liste

Parla delle liste.
La struttura di una lista è composta da nodi. Ogni nodo contiene un campo **data** che memorizza l'informazione e un campo **next** che memorizza il puntatore al nodo successivo.

```
typedef struct Nodo {
   int data;
   struct Nodo* next;
} Nodo;
```

Vediamo le operazioni che possono essere eseguite:

- **Creazione** di lista **vuota**

```
Nodo * crea_lista(){
   return NULL;
}
```

- **Distruzione** della lista

```
void distruggi_lista(Nodo* testa) {
   Nodo * corrente = testa;
   while (corrente != NULL) {
      Nodo * temp = corrente;
      corrente = corrente->next;
      free(temp);
   }
}
```

- **Stampa**

```
void stampa_lista(Nodo* testa) {
   Nodo * corrente = testa;
   while (corrente != NULL) {
      printf("%d ->", corrente->data);
      corrente=corrente->next;
```

```
    }
}
```

- **Inserimento** in **testa**
  - *Iterativo*

```
Nodo* inserisci_testa(Nodo* testa, int valore) {
   Nodo * nuovo_nodo = malloc(sizeof(Nodo));
   nuovo_nodo->data = valore;
   nuovo_nodo->next = testa;
   return nuovo_nodo;
}
```

  - *Ricorsivo*

```
Nodo* inserisci_testa_ricorsivo(Nodo* testa, int valore) {
   if(testa == NULL) {
      Nodo* nuovo = malloc(sizeof(Nodo));
      nodo->data = valore;
      nodo->next = NULL;
      return nuovo;
   }
   Nodo* nuovo = malloc(sizeof(Nodo));
   nuovo->data = valore;
   nuovo->next = testa;
   return nuovo;
}
```

- **Cancellazione di un elemento**
  - *Iterativo*

```
Nodo* cancella_elem(Nodo* testa, int valore){
   if(testa==NULL) return NULL;
   if(testa->data == valore) {
      Nodo* tmp = testa;
      testa = testa->next;
      free(tmp);
      return testa;
   }
   Nodo* corrente = testa;
   while (corrente->next != NULL && corrente->next->data != valore)
   corrente = corrente->next;
   if(corrente->next == NULL){
      Nodo* temp = corrente->next;
      corrente->next = temp->next;
      free(temp);
   }
   return testa;
}
```

  - *Ricorsivo*

```
Nodo* cancella_ric(Nodo* testa, int valore){
   if(testa==NULL) return NULL;
   if(testa->data==valore){
      Nodo* tmp = testa;
      testa = testa->next;
      free(tmp);
```

```
        return testa;
    }
    testa->next = cancella_ric(testa->next, valore);
    return testa;
}
```

2.6 Union

Descrivi union.
Union è un tipo di dato che consente di memorizzare variabili di diversi tipi nello stesso spazio di memoria. A differenza delle struct, in cui ogni membro ha il proprio spazio, in una union tutti i membro condividono la stessa area di memoria e di conseguenza solo un membro alla volta può contenere un valore. La dimensione di union è uguale alla dimensione del suo membro più grande.

```
union Data{
    int i;
    float f;
    char str[20];
};

union Data data;
data.i = 10;
data.f = 220.5;
```

3 Teoria

3.1 Aritmetica dei puntatori

L'aritmetica dei puntatori ti consente di eseguire operazioni matematiche utilizzando direttamente i puntatori:

- Incremento con offset: incrementando un puntatore esso punterà all'elemento successivo
- Decremento con offset: decremento all'elemento precedente

```
int* p = array;
p++;
p--;
```

Note 3.1.0.1. Quando si eseguono queste operazioni è importante prestare attenzione ai limiti dell'array.

3.2 Buffer overflow

Problematica del buffer overflow.
Quando creiamo un buffer, stiamo associando ad esso un arco di memoria. Se i dati scritti in questo buffer superano la capacità i dati eccedenti sovrascrivono la memoria adiacente. Per evitarlo possiamo usare funzioni come *fgets* e *snprintf* o usare allocazione dinamica.

3.3 Variabili statiche

Qual è l'uso delle variabili statiche?
Una variabile statica ha una durata di memoria globale, ovvero dura per tutta l'esecuzione del programma. Sono visibili all'interno dello scope della funzione dichiarata. Vengono inizializzate una sola volta e occupano memoria per tutta l'esecuzione, anche se non utilizzate.

3.4 MAKEFILE

Parla del MAKEFILE e della sua compilazione.
Il MAKEFILE è uno strumento per gestire la compilazione di progetti, specialmente quelli composti da più file sorgente.

```
target: dipendenze
   comando
```

Si compone di:

- *target*: nome del file da creare
- *dipendenze*: file da cui il target dipende
- *comando*: comando da eseguire

Esempio 3.4.1 (MAKEFILE)**.** Dato *main.c* il file sorgente, *funzioni.c* il sorgente con funzioni aggiuntive e *funzioni.h* il suo header.

```
# Compilatore
CC=gcc
# Opzioni di compilazione
CFLAGS= -wall -g
# Eseguibile
TARGET = programma
# File sorgenti
SRCS = main.c funzioni.c
# File oggetto
OBJS = main.o funzioni.o
```

```
# Regola principale
all: $(TARGET)
# Regola per creare eseguibile
$(TARGET): $(OBJS)
   # $< significa primo file nella lista dipendenze
   # $@ significa nome del target .o
   $(CC) $(CFLAGS) -c $< -o $@
# Pulizia
clean:
   rm -f $(OBJS) $(TARGET)
```

3.5 void*

Necessità ed utilizzo del tipo *void.**
Il puntatore *voidé* è un puntatore generico che può essere utilizzato per rappresentare un puntatore ad un qualsiasi dato. Un esempio di applicazione sono le funzioni parametrizzate che prendono un input valori diversi a seconda della chiamata.

3.6 Programma vs processo

Qual è la differenza tra un programma ed un processo?
Un **programma** è un insieme di istruzioni scritte in un linguaggio di programmazione che risiede su un dispositivo come file eseguibile o sorgente.
Un **processo** è un'entità dinamica, rappresenta un'istanza in esecuzione di un programma. È gestito dal sistema operativo che gli assegna risorse per la sua esecuzione.

3.7 PID

PID di un processo.
Il PID è un identificatore assegnato dal sistema operativo ad ogni processo in esecuzione. È utile per la gestione dei processi, il loro controllo e la comunicazione con essi tramite segnali.

```
getpid(); // Restituisce il PID del processo chiamante
getppid(); // Restituisce il PID del processo genitore
```

3.8 Processi in background

In Linux è possibile avviare un processo in background aggiungendo il simbolo *&* alla fine del comando.

```
./programma &
```

Il terminale visualizza il PID del processo avviato.

Qual è la relazione tra *syscall* e le lybrary functions?
Le *syscall* sono funzioni fornite dal sistema operativo per interagire direttamente con il kernel.

```
int open(const char* pathname, int flags, mode_t mode);
size_t read(int fd, void* buf, size_t count);
size_t write(int fd, const void* buf, size_t count);
```

Queste funzioni operano su un file descriptor senza buffering e con gestione manuale delle risorse.
Le *funzioni di libreria* invece offrono un livello superiore , semplificando le operazioni di lettura e scrittura grazie al **buffering**.

```
FILE* fopen(const char* pathname, const char* mode);
size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
size_t fwrite(const void* ptr, size_t size, size_t count, FILE* stream);
```

Queste operano su puntatori al file (struttura che incapsula il file descriptor e il buffer) e offrono un sistema di buffering, riducendo le chiamate al kernel.

3.9 Variabili statiche

Qual è l'uso delle variabili statiche?
Una variabile statica ha una durata di memoria globale, ovvero dura per tutta l'esecuzione del programma. Sono visibili all'interno dello scope della funzione dichiarata. Vengono inizializzate una sola volta e occupano memoria per tutta l'esecuzione, anche se non utilizzate.

3.10 UMASK

Significato e uso di UMASK.
UMASK è un parametro in molti sistemi che definisce i **permessi** assegnati ai file o alle directory appena creati. Sono usate per migliorare la sicurezza e si lavora in un ambiente e se si lavora in un ambiente con i file condivisi può permettere di scrivere nel file di gruppo.
Ogni cifra rappresenta 3 bit: lettura, scrittura ed esecuzione. Lo 0 rappresenta la codifica ottale del numero. Le tre cifre dopo lo 0 rappresentano rispettivamente i permessi per l'utente, per il gruppo e per tutti.
Ad esempio 0666 indica che un file è senza restrizioni: in binario abbiamo 0 110 110 110, dove tutti (utente, gruppo e chiunque) hanno permesso di lettura, scrittura ma non esecuzione.

3.11 SIGINT, SIGSTOP, SIGQUIT

- **SIGINT(2)** (CTR+C) interrompe il processo in modo controllato dando al programma l'opportunità di gestire l'uscita.
- **SIGSTOP(19)** (CTRL+Z) sospende il processo e non può essere né intercettato né ignorato dal programma. Per farlo riprendere successivamente si invia il segnale **SIGCONT**.
- **SIGQUIT(3)** (CTRL+\\) termina il programma e, se abilitato, genera un core dump per diagnosticare eventuali problemi

3.12 MT-Safe

Concetto di Multi Thread safe.
Una funzione è MT-Safe se può essere chiamata in modo sicuro da più thread senza causare comportamenti indesiderati o corruzione dei dati. Deve avere:

- Accesso protetto ai dati condivisi attraverso meccanismi di protezione
- Non modificare variabili statiche o globali senza protezione
- Evitare race conditions

3.13 Duplicazione di stringhe

Come si duplica una stringa?

```
size_t len = strlen(str);
char* str2 = malloc((len+1)*sizeof(char));
strcpy(str2, str);
```

3.14 Manipolazione dei file

Per elencare i file:

```
ls -l # Dettagliato
ls -a # Mostra anche quelli nascosti
ls -R # Elenca ricorsivamente il contenuto delle sottodirectory
ls -lh # Specifica anche la dimensione dei file
```

L'output di *ls -l* ci specifica anche se un determinato oggetto è un file o una directory in quanto l'ultima avrà in aggiunta la lettera *d* mentre il primo avrà *-*.

4 Inter Process Communication

4.1 Memoria condivisa

Definizione ed esempi di memoria condivisa.
La memoria condivisa è un'area di memoria accessibile da più processi in un sistema operativo. È un meccanismo per consentire a più processi di comunicare e condividere dati direttamente, evitando di passare messaggi o usare file. Riserva un pezzo di RAM ai processi.
L'idea è che abbiamo a disposizione un file condiviso con due problemi:

- I file sono lenti specialmente con grandi dimensioni
- Le operazioni sui file sono ad accesso sequenziale tranne la *seek* che però è scomoda

Per risolvere il problema usiamo *shm_open* che apre un oggetto di memoria condivisa e restituisce un file descriptor e *truncate* che ne imposta la dimensione. L'oggetto in questione viene poi mappato con *mmap*. Al termine useremo *unmap* per annullare la mappatura nella memoria condivisa, *close* per chiudere il file descriptor e *shm_unlink* per rimuovere l'oggetto dalla memoria condivisa.

```
int shm_size = sizeof(int) + sizeof(long);
inf fd = shm_open(sommamem, O_RDWR | O _CREAT, 0660, NULL);
xftruncate(fd, shm_size, HERE);
char* tmp = mmap(shm_size, fd, NULL);
xshm_unlink(sommamem, NULL);
close(fd);
munmap(tmp, shm_size);
```

4.2 shm_open, ftruncate e mmap

Uso e significato di *shm_open*, *ftruncate* e *mmap*.

- *shm_open* serve a creare o aprire un oggetto di memoria condivisa

  ```
  int shm_open(const char* nome, int oflag, mode_t mode);
  ```

  Dove i parametri sono:

  - *nome* nome dell'oggetto nella memoria condivisa
  - *oflag* flag che indica se creare, leggere o scrivere (*O_CREAT*, *O_RDONLY* o *O_RDWR*)
  - *mode* permessi di accesso

  Ritorna un file descriptor associato all'oggetto di memoria condivisa.

- *ftruncate* è usato per impostare la dimensione di un file o di un oggetto di memoria condivisa.

  ```
  int ftruncate(int fd, off_t length);
  ```

  Dove i parametri sono:

  - *fd* file descriptor ottenuto con *shm_open*
  - *length* nuova dimensione del file

  Restituisce 0 in caso di successo, -1 altrimenti.

- *mmap* serve a mappare un file o una regione di memoria (e.g. oggetto creato da *shm_open*) nello spazio di indirizzamento del processo

  ```
  void* nmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset);
  ```

Dove i parametri sono:

- *addr* indirizzo suggerito per il mapping, solitamente NULL
- *length* dimensione della regione da mappare
- *prot* permessi di accesso
- *flags* specifica opzioni
- *fd* file descriptor
- *offset* offset inizio del file

Restituisce la memoria mappata o *MAP_FAILED* in caso di errore.q

4.3 Async-signal-safe

Una funzione si definisce async signal safe se può essere chiamata in modo sicuro all'interno del signal handler. Questo significa che non fa affidamento su risorse che potrebbero essere lasciate in uno stato incoerente a causa dell'interruzione provocata dal segnale:

- Non usare mutex o risorse che potrebbero essere bloccate all'arrivo del segnale
- Non modificare dati globali o statici in modo atomico

Alcuni esempi sono: *write*, *read*, *close*, *signal*, *exit*, *kill*.

4.4 Maschera dei segnali bloccati

La maschera dei segnali bloccati viene gestita utilizzando le funzioni della libreria POSIX. Per i thread in ambienti MT si utilizza:

```
int pthread_sigmask(int how, const sigset_t* set, sigset_t* oldset);
```

Dove i parametri sono:

- *how* indica l'operazione da eseguire tra
  - *SIG_BLOCK* : aggiunge i segnali di un set alla maschera
  - *SIG_UNBLOCK*: rimuove i segnali di un set dalla maschera
  - *SIG_SETMASK*: imposta la maschera da un set
- *set*: insieme di segnali da aggiungere, rimuovere o impostare
- *oldset*: se non NULL viene riempito con la vecchia maschera

4.5 Segnali real time

I segnali si suddividono in:

- **Standard signal**: quando vengono inviati più segnali dello stesso tipo ad un processo o thread il sistema li accorpa in un unico segnale. Se il segnale non è ancora gestito ulteriori invii dello stesso segnale non verranno accodati e andranno persi.
- **Real time**: sono quelli con il codice che va da 32 a 64. Alcuni di essi sono utilizzabili solo dal SO. Quelli per l'utente vanno da *SIGRTMIN* e *SIGRTMAX*. Le istanze di segnali vengono accodate. Ogni invio viene mantenuto in una coda separata con la possibilità di includere dati aggiuntivi tramite una *union*.

4.6 Pipe

Spiega le pipe.
Le pipe sono un meccanismo di comunicazione tra processi che consente il trasferimento di dati tra due processi attraverso un buffer gestito dal sistema operativo. Ne esistono di due tipi:

- **Named**: una pipe con nome (chiamata anche FIFO) permette la comunicazione tra processi non correlati. Si crea con la funzione

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

  Dove i parametri sono:

  – *pathname* dove la pipe verrà create
  – *mode* specifica i permessi di accesso al file

  Se ha successo restituisce 0, altrimenti -1 e imposta *errno*. Al termine dell'utilizzo si deve fare

  ```
  int unlink(const char* pathname);
  ```

  che rimuove il link al file. Quando tutti i link sono stati rimossi, il file viene eliminato.

- **Unnamed**: utilizzato per la comunicazione tra processi correlati. Utilizza:

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

  Dove *pipefd* è un array di due elementi:

  – *pipefd[0]* è l'estremità di lettura della pipe
  – *pipefd[1]* è l'estremità di scrittura della pipe

  In caso di successo la funzione restituisce 0, altrimenti -1 e imposta *errno*. Se il lettore tenta di leggere i dati prima che siano stati inviati, rimane in attesa.

In entrambi i casi al termine dell'utilizzo bisogna utilizzare *close*.

5 Metodi di sincronizzazione

5.1 Race condition

Una race condition è un comportamento che si verifica quando due o più thread accedono ad una risorsa contemporaneamente senza un'adeguata sincronizzazione. Il risultato del programma può variare a seconda dell'ordine in cui i thread accedono alle risorse.

5.2 Semafori POSIX

I semafori sono strumenti di sincronizzazione utilizzati per condividere l'accesso concorrente a risorse condivise tra processi. Si dividono in:

- **Named**: hanno un nome associato nel file system. Si aprono con:

```
sem_t* sem_open(const char* name, int oflag, mode_t mode, unsigned int value);
```

  Dove i parametri sono:

  - *name* nome del semaforo
  - *oflag* flag per specificare la modalità
    - *O_CREAT* se il semaforo non esiste viene creato
    - *O_EXECL* se il semaforo esiste fallisce e restituisce un errore
  - *mode* permessi di accesso
  - *value* valore iniziale del semaforo

  Per chiuderlo e poi rimuoverlo:

```
int sem_close(sem_t* sem);
int sem_unlink(sem_t* sem);
```

- **Unnamed**: non hanno un nome associato nel file system e sono usati genericamente per la sincronizzazione di thread in uno stesso processo o tra processi con memoria condivisa. Si inizializza con:

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

  Dove i parametri sono:

  - *sem* è il puntatore alla variabile *sem_t* che rappresenta il semaforo
  - *pshared*
    - 0 il semaforo è usato da thread di uno stesso processo
    - 1 il semaforo sarà condiviso tra processi diversi e vive quindi in una regione di memoria condivisa
  - *value* è il valore iniziale del semaforo

  Ritorna 0 in caso di successo, -1 altrimenti settando *errno*.
  Per distruggerlo:

```
int sem_destroy(sem_t* sem);
```

Entrambi i tipi di semafori prevedono due operazioni:

- **Wait**: decrementa il valore del semaforo. Se il valore è 0 il processo è bloccato fino ad un nuovo incremento.

```
int sem_wait(sem_t* sem);
```

- **Post**: incrementa il valore del semaforo e sblocca eventuali processi in attesa

```
int sem_post(sem_t* sem);
```

5.3 Mutex

Il mutex è un meccanismo di sincronizzazione utilizzato per garantire che un solo thread alla volta possa accedere ad una risorsa condivisa. Garantisce la mutua esclusione. Di seguito le funzioni rilevanti:

- **Inizializzazione** può essere fatta in due modi

```
//DINAMICA, attr opzionale
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutex_attr_t* attr);

// STATICA
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- **Lock**: blocca il mutex. Se è già bloccato il thread rimane in attesa fino allo sblocco.

```
int pthread_mutex_lock(pthread_mutex_t mutex);
```

- **Unlock**: rilascia il mutex eventualmente sbloccando thread in attesa

```
int pthread_mutex_unlock(pthread_mutex_t mutex);
```

- **Distruzione**

```
int pthread_mutex_destroy(pthread_mutex_t mutex);
```

5.4 Produttore consumatore

Il problema del produttore-consumatore è un esempio della sincronizzazione tra processi o thread. Un produttore genera dei dati e li mette in un buffer condiviso mentre il consumatore li preleva per elaborarli. Il problema principale è garantire che il produttore non inserisca dati quando il buffer è pieno e che il consumatore non li prelevi quando è vuoto. Per risolvere questo problema usiamo dei **semafori** che regolano l'accesso al buffer e gestiscono le condizioni di attesa e i **mutex** garantiscono l'accesso esclusivo al buffer durante le operazioni di inserimento e prelievo.

5.5 Condition variables

Le condition variables sono strumenti di sincronizzazione tra thread. Gli permettono di attendere determinati eventi e di essere notificati quando questi accadono. Si utilizzano in combinazione con i mutex. Le operazioni eseguibili sono:

- **Creazione**

```
// STATICA
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
```

```
// DINAMICA, attr opzionale
int pthread_cond_init(&condvar, pthread_condattr_t* attr);
```

- **Attesa**

```
int pthread_cond_wait(pthread_cond_t* condvar, pthread_mutex_t* mutex);
```

- **Segnalazione**
  - Thread singolo

```
int pthread_cond_signal(pthread_cond_t* condvar);
```

  - Tutti i thread

```
int pthread_cond_broadcast(pthread_cond_t* condvar);
```

Note 5.5.0.1. La differenza di utilizzo tra *signal* e *broadcast* la abbiamo quando alcuni dei thread potrebbero non essere ancora pronti. In questo caso la broadcast li risveglia tutti e quelli che possono procedere lo faranno mentre la signal ne sveglia solo uno (si rischia che non sia quello corretto).

6 Python

6.1 Python vs C

Quali sono le differenze principali tra Python e C?

- Python gestisce gli **interi** con una precisione arbitraria, ovvero il limite è la quantità di memoria disponibile, mentre C li ha di dimensione limitata.
- Le **liste** in Python sono oggetti complessi che supportano operazioni avanzate mentre gli **array** in C sono limitati e meno flessibili.
- Python semplifica molto la gestione dei **file**
- Python è preferibile per script, prototipi e applicazioni rapide mentre C per applicazioni embedded, applicazioni ad alte prestazioni e sistemi operativi

6.2 Navigazione file system

Grazie al package *os*.

6.2.1 Elenco file

Per elencare i file in una directory:

```
path = "."
for entry in os.listdir(path):
   print(entry)
```

6.2.2 Directory

Per verificare se un file è una directory:

```
for entry in os.listdir(path):
   full_path = os.path.join(path, entry)
   if os.path.isdir(full_path):
      print("Its a directory")
```

6.2.3 Attraversare FS

Per attraversare ricorsivamente il file system:

```
for root, dirs, files in os.walk(path)
```

6.2.4 Manipolazione

Per manipolare le directory uso i seguenti comandi:

```
os.mkdir(path, mode=0o666) # Creo
os.rename(source, destination) # Rinomino
os.chdir(path) # Cambio directory corrente
os.rmdir(path) # Elimino
```

6.3 Subprocess

Per invocare comandi della shell da Python si utilizza dal pacchetto *subprocess*:

```
result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
print(result.stdout)
```

Dove i parametri sono:

- Lista che specifica il comando da eseguire e i suoi argomenti
- *capture_output* specifica se catturare *stderr* e *stdout*
- *text* specifica se convertire l'output in stringhe testuali

6.4 Classi

Le classi in Python possono avere metodi speciali:

- *__init__* viene chiamato automaticamente alla creazione di una nuova istanza della classe
- *__eq__* definisce il comportamento di ==
- *__str__* fornisce una rappresentazione dell'oggetto in stringa
- *__repr__* fornisce una rappresentazione stampabile dell'oggetto, utile per il debug
- *__lt__* definisce il comportamento di <
- *__hash__* consente di usare l'oggetto come chiave in un dizionario e di inserirlo in un set. In questo caso è importante anche definire *__eq__*

6.5 Metodo hash

Il metodo hash in Python viene usato per calcolare l'hash di un oggetto ed è un elemento chiave nella gestione di oggetti immutabili e strutture dati come dizionari e set. Di default ogni classe eredita da *Object* l'implementazione hash:

```
class MyClass:
   pass
obj1 = MyClass()
obj2 = MyClass()

# Sono diversi
assert obj1 != obj2
```

Ma se viene ridefinita dall'utente:

```
class Persona:
   def __init__(self, nome, eta):
      self.nome = nome
      self.eta = eta
   def __eq__(self, other):
      return self.nome == other.nome and self.eta == other.eta
   def __hash__(self):
      return hash((self.nome, self.eta))

p1 = Persona("Mario", 30)
p2 = Persona("Mario", 30)

# Sono uguali
assert p1 == p2
```

6.6 Multithreading

In Python non esiste un vero multithreading per via dell'esistenza del **Global Interpreter Lock**. Infatti quest'ultimo limita l'utilizzo dell'interprete Python ad un solo thread per processo e quindi alla fine è un falso multithreading. L'unico modo per aggirare questa problematica è creare più processi tramite il modulo *multiprocessing*.
L'unico caso in cui il multithreading in Python è utile è in presenza di operazioni di I/O, dove quindi i thread rimangono comunque in attesa.
Esistono vari metodi per gestire i thread in Python:

- *threading.Thread*

```
def funzione_thread(nome, ritardo):
   print(f"Thread {nome}: Inizio")
   time.sleep(ritardo) # Simula un'operazione che richiede tempo
   print(f"Thread {nome}: Fine")
thread1 = threading.Thread(target=funzione_thread, args=("Uno", 2))
thread2 = threading.Thread(target=funzione_thread, args=("Due", 1))

thread1.start()
thread2.start()

thread1.join() # Attende la terminazione di thread1
thread2.join() # Attende la terminazione di thread2
```

  Dove i parametri sono:

  - *target* è la funzione che deve essere eseguita dal thread
  - *args* è una tupla contenente gli argomenti da passare alla funzione che deve essere eseguita

- Sottoclasse di *threading.Thread*

```
class MioThread(threading.Thread):
   def __init__(self, nome, ritardo):
      super().__init__()
      self.nome = nome
      self.ritardo = ritardo

def run(self):
   print(f"Thread {self.nome}: Inizio")
   time.sleep(self.ritardo)
   print(f"Thread {self.nome}: Fine")

thread1 = MioThread("Uno", 2)
thread2 = MioThread("Due", 1)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

- *concurrent.futures.threadPoolExecutor* permette di gestire un pool di thread e facilita la creazione, la partenza e la terminazione

```
def funzione_thread(numero):
   print(f"Thread che elabora il numero: {numero}")
   time.sleep(1) # Simula un'operazione
   return numero * 2

numeri = [1, 2, 3, 4, 5]
```

```
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
   risultati = list(executor.map(funzione_thread, numeri))

print(f"Risultati: {risultati}")

# In alternativa
with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
   futures = [executor.submit(funzione_thread, n) for n in numeri]

for future in concurrent.futures.as_completed(futures):
   print(f"Risultato di un future: {future.result()}")
```

Dove *max_workers* indica il numero massimo di thread che possono coesistere.