Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 2° anno - 12 CFU

Laboratorio II

Domande esame orale

Professori:
Prof. Giovanni Manzini

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 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.

1.16 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).

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

1.17 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.

1.18 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.

1.19 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.

1.20 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.

1.21 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

Note 1.21.0.1. Utilizzare il parametro *-l* per vedere l'elenco dei segnali e i loro numeri.

1.22 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.

1.23 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