Sistemi Operativi

C.d.L. in Informatica (laurea triennale)

Anno Accademico 2025-2026

Laboratorio di Sistemi Operativi (MZ)

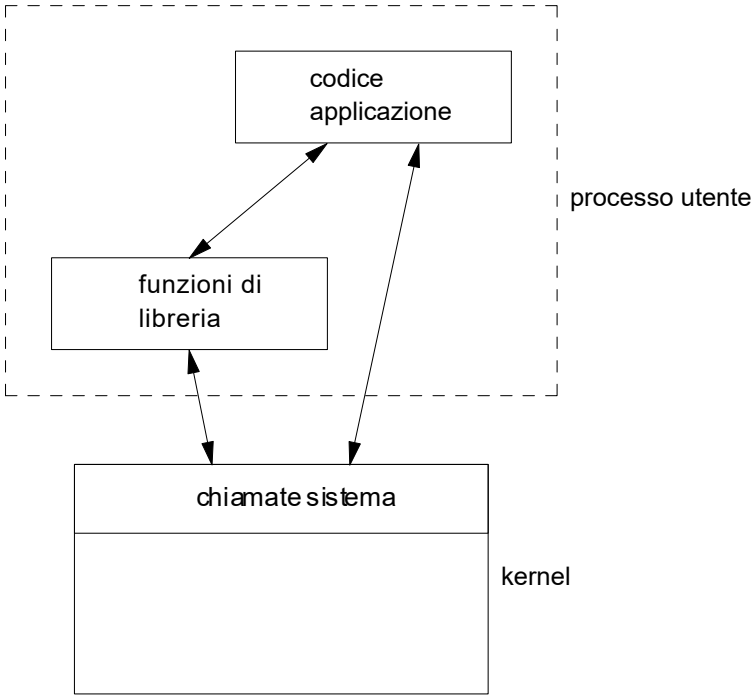
Prof. Mario Pavone

Dipartimento di Matematica e Informatica

Università degli Studi di Catania

mario.pavone@unict.it

http://ww.dmi.unict.it/mpavone/so.html



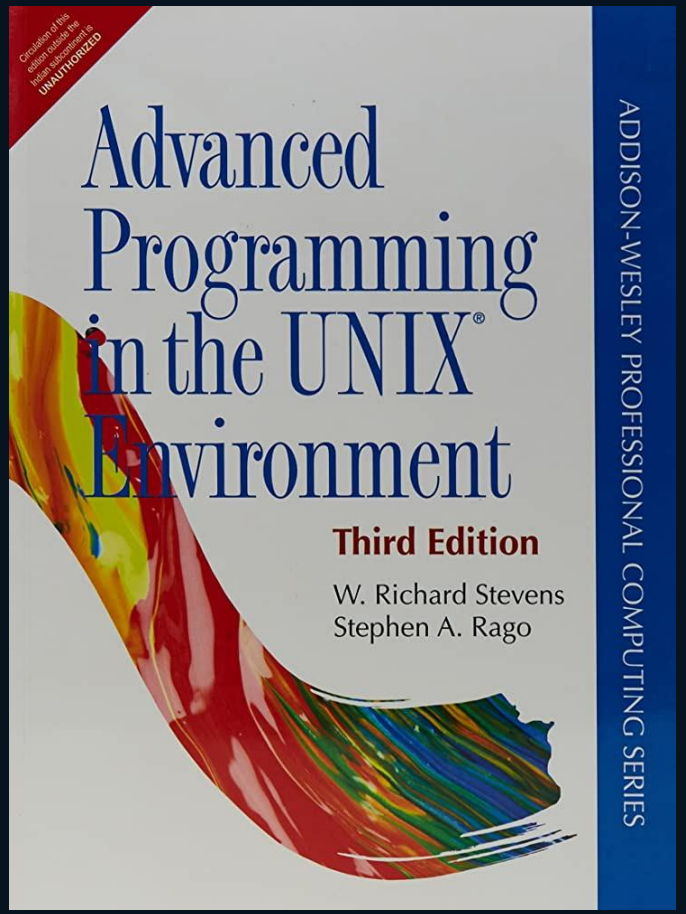
Chiamate di Sistema

- Nei nostri programmi possiamo impiegare:
  - **chiamate di sistema**: servizi offerti direttamente dal Sistema Operativo (TRAP, modalità kernel)
  - **chiamate di libreria**: funzioni incluse in libreria di sistema (modalità utente)

- nel corso ci occuperemo dei seguenti sotto-sistemi:
  - gestione dei **file system** (file e directory) e dell'**I/O**
  - gestione dei **processi**
  - gestione dei **thread**
  - **comunicazione** e **sincronizzazione** di processi e thread

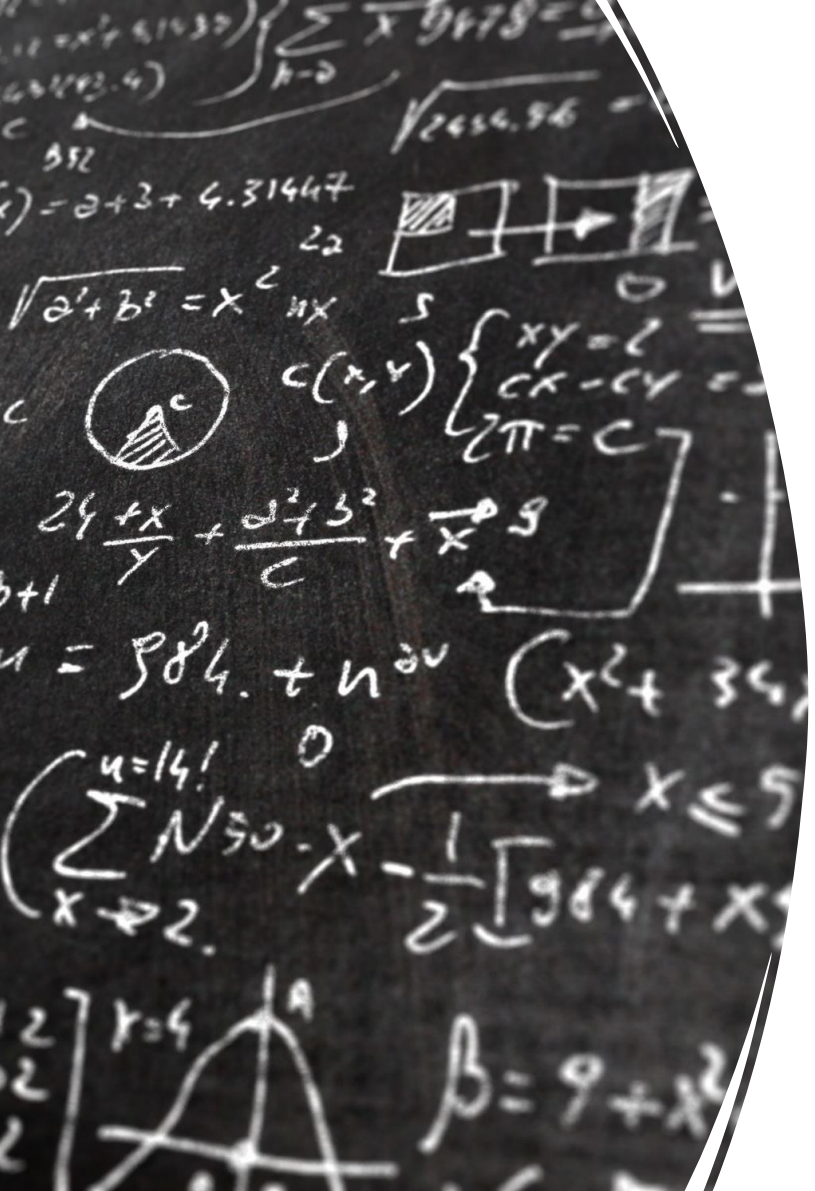
Materiale di Riferimento

- Manuale di riferimento:
  - ***Advanced Programming in the UNIX Environment*** (terza edizione – 2013) di Stevens e Rago
- è possibile utilizzare qualunque altro testo o risorsa web che tratti l'argomento
- altre valide alternative:
  - ***Programming With POSIX Threads***
    di Butenhof
  - ***PThreads Primer: A Guide to Multithreaded Programming***
    di Lewis e Berg



Documentazione

- Le pagine di manuale (**man pages**) UNIX rappresentano la documentazione ufficiale:
  - accessibile da:
    - **shell**: **man** ***comando-o-funzione***
      - pacchetto **manpages-posix-dev** su Debian/Ubuntu
    - **online:** man.cx , man7.org , …
  - **sezioni**:
    - n.1: comandi utente
    - n.2: chiamate di sistema
    - n.3: librerie di sistema
    - …
    - n.8: comandi di amministrazione
  - casi di omonimia: **man chown** *vs.* **man 3 chown**



Creazione di Programmi Eseguibili

- Compilazione diretta di un solo sorgente:
  - **gcc -o *nome-eseguibile sorgente.c* - gcc *sorgente.c* -o *nome-eseguibile***
- compilazione da sorgenti multipli:
  - **gcc -c *file1.c* ; gcc -c *file2.c***
  - **gcc -o *nome-eseguibile file1.o file2.o***
- linking con librerie: opzione **-l *nome-libreria***
  - **gcc -l m -l pthread *sorgente.c***
  - linking statico: opzione aggiuntiva **–static**
- specifica dello standard C da utilizzare: opzione **-std=**
  - **gcc -std=c99 *sorgente.c***
- per progetti più articolati si può usare **make** e un progetto **makefile**
  - regole del tipo: **target ← dipendenze / comando**
  - esempio: **makefile.sample** 🗎

Gestione Standard degli Errori

- La maggior parte delle chiamate di sistema segnalano **errori** nell'esecuzione:
  - riportando un **valore di ritorno** anomalo (in genere -1)
  - impostando una **variabile globale** prestabilita:
    - `extern int errno;`
    - dichiarata nell'header `errno.h` (generalmente automaticamente inclusa)

```
#define EPERM           1       /* Operation not permitted */
#define ENOENT          2       /* No such file or directory */
#define ESRCH           3       /* No such process */
#define EINTR           4       /* Interrupted system call */
…
```

    - nota: in caso di successo `errno` non viene resettata!
  - **errori fatali** *vs.* **non fatali** (ad es. `EINTR`, `ENFILE`, `ENOBUFS`, `EAGAIN`, ...)

```
char *strerror(int errnum);
void perror(const char *s);
```

<string.h>, <stdio.h>

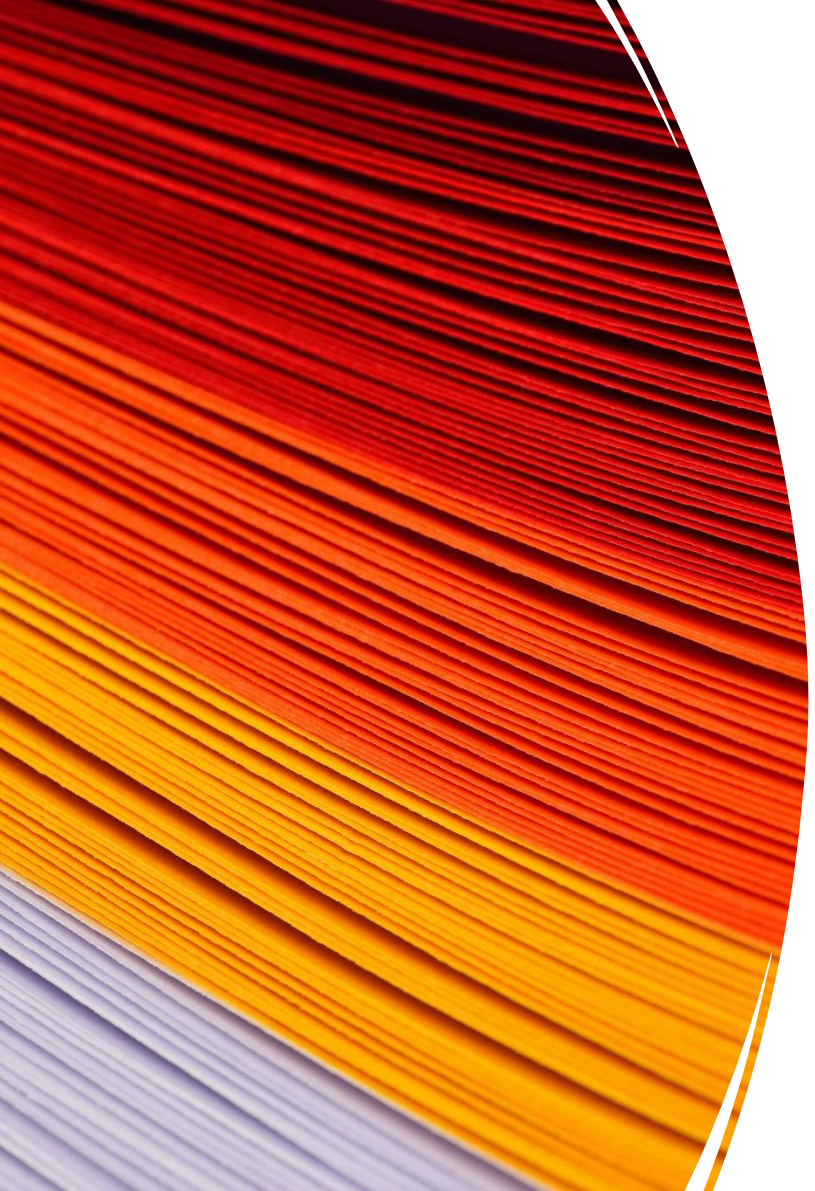
Terminazione del Processo

```
void exit(int status);
int atexit(void (*func)(void));
                                                    <stdlib.h>
```

- **exit** termina il processo con ***exit code*** pari a (status && 0xFF)
  - convenzione UNIX (quasi universale): “0” = “*tutto ok*”, “>0” = “*errore*”
  - costanti apposite per maggiore portatilità: **EXIT_SUCCESS**, **EXIT_FAILURE**
- un processo termina anche quando:
  - il main ritorna (si può pensare a qualcosa tipo: exit(main(argc, argv))
  - l'ultimo thread termina
- exit termina in “*modo pulito*” il processo:
  - scrivendo eventuali buffer in sospeso (vedi *stream* più avanti)
  - eseguendo eventuali procedure di chiusura registrate con **atexit**
- esempio: **at-exit.c**

Descrittori di File

- Ogni processo può aprire uno o più file ottenendo un intero non negativo detto **descrittore di file** (*file descriptor*) come riferimento
- esistono tre **canali predefiniti** a cui è associato già un descrittore:
  - **standard input** (0)
  - **standard output** (1)
  - **standard error** (2)
  - in **unistd.h** sono definite apposite costanti:
    - **STDIN_FILENO**, **STDOUT_FILENO** e **STDERR_FILENO**
- in esecuzioni dirette in genere sono associati al terminale ma non sempre:
  - **./my-prog > output-file.txt < input-file.txt**
  - **cat input.txt | ./my-prog | sort > output.txt**



Apertura, Creazione e Chiusura di un File

```
int open(const char *path, int oflag, [ mode_t mode ] );
int creat(const char *path, mode_t mode );
int close(int fd)                                    <fcntl.h>, <unistd.h>
```

- **open** apre (ed eventualmene crea) un file con percorso **path**
  - **oflag**: intero che può combinare alcuni flag per l'apertura:
    - **O_RDONLY** / **O_WRONLY** / **O_RDWR** (in modo mutuamente esclusivo)
    - **O_APPEND**: ogni scrittura avverrà alla fine del file
    - **O_CREAT**: crea il file se non esiste usando i permessi indicati in **mode**
    - **O_EXCL**: usato con O_CREAT, genera un errore se il file esiste già
    - **O_TRUNC**: se il file esiste, viene troncato ad una lunghezza pari a 0
  - ritorna: -1 in caso di errore o il descrittore del file appena aperto (≥0)
- **creat** equivale a: open(path, O_RDWR | O_CREAT | O_TRUNC, mode)
- **close** chiude un file aperto

Permessi sugli Oggetti del File-System UNIX

- I permessi sono di triplice natura: **lettura** (R) / **scrittura** (W) / **esecuzione** (X)
- il tipo **mode_t** è un intero che codifica una **maschera con permessi** per:
  - **utente proprietario** (USR)
  - **gruppo proprietario** (GRP)
  - **tutti gli altri utenti** (OTH)
- la maschera si può ottenere da costanti definite in **sys/stat.h**:

S_IRUSR	**S_IWUSR**	**S_IXUSR**
S_IRGRP	**S_IWGRP**	**S_IXGRP**
S_IROTH	**S_IWOTH**	**S_IXOTH**

- è anche prassi, non raccomandata, utilizzare direttamente la **rappresentazione numerica ottale**: ad esempio: 0640 ≈ S_IRUSR | S_IWUSR | S_IRGRP
- per le **directory**: X rappresenta il **diritto di attraversamento**

Maschera di Creazione per i Permessi

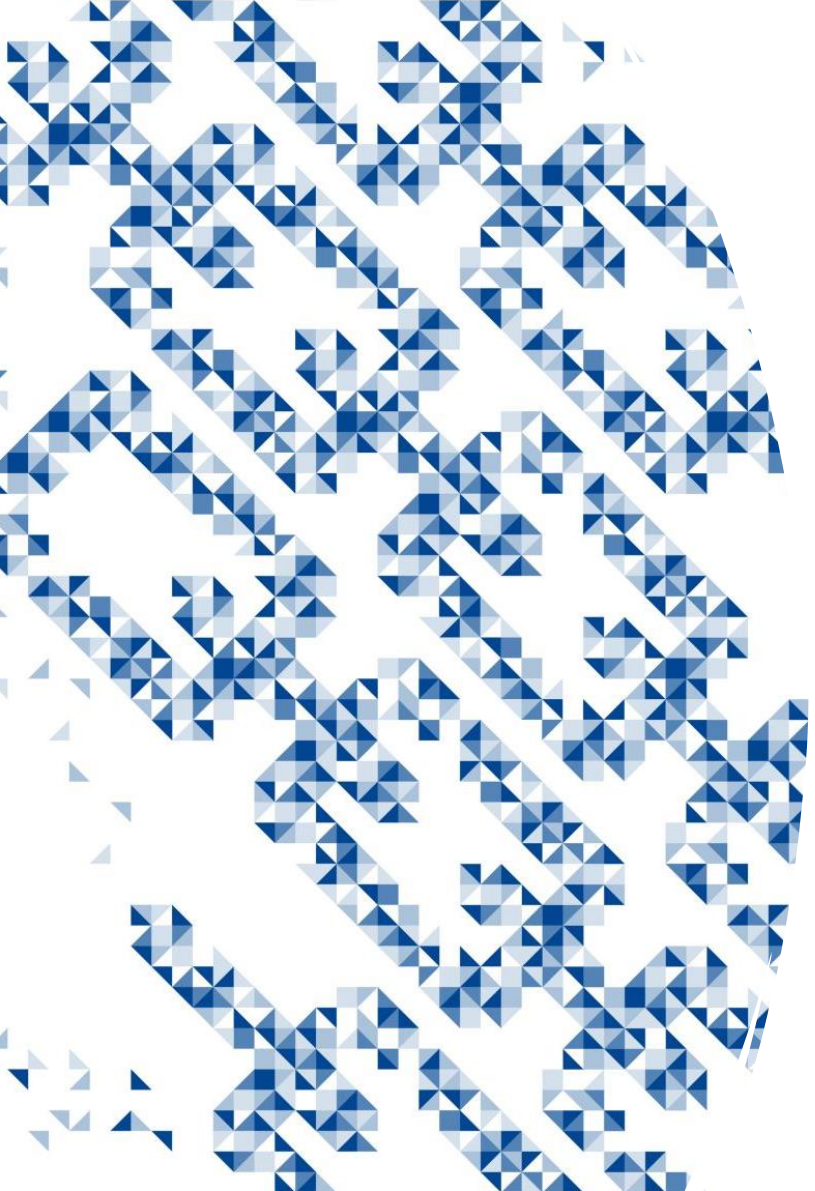
```
* Set the umask to RW for owner,group; R for other
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void )
  {
    mode_t omask;
    mode_t nmask;

    nmask = S_IRUSR | S_IWUSR | /* owner read write */
            S_IRGRP | S_IWGRP | /* group read write */
            S_IROTH;            /* other read */
    omask = umask( nmask );
    printf( "Mask changed from %o to %o\n",
             omask, nmask );
    return EXIT_SUCCESS;
```

Posizionamento

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

<unistd.h>

- ogni file aperto ha un **file offset** che simula l'accesso sequenziale
  - posto a 0 in apertura se non si è usato O_APPEND
  - aggiornato ad ogni operazione
- **lseek** effettua uno spostamento di **offset** byte rispetto a **whence**:
    - **SEEK_SET**: rispetto all'inizio del file
    - **SEEK_CUR**: rispetto alla posizione attuale (offset può essere negativo)
    - **SEEK_END**: rispetto alla fine del file (offset può essere negativo)
  - ritorna: -1 in caso di errore o la nuova posizione rispetto all'inizio del file (≥0)
  - non comporta alcuna operazione di I/O; valido solo su file
- *ottenere la posizione attuale*: pos = lseek(fd, 0, SEEK_CUR);
- esempio: **test-seek-on-stdin.c**



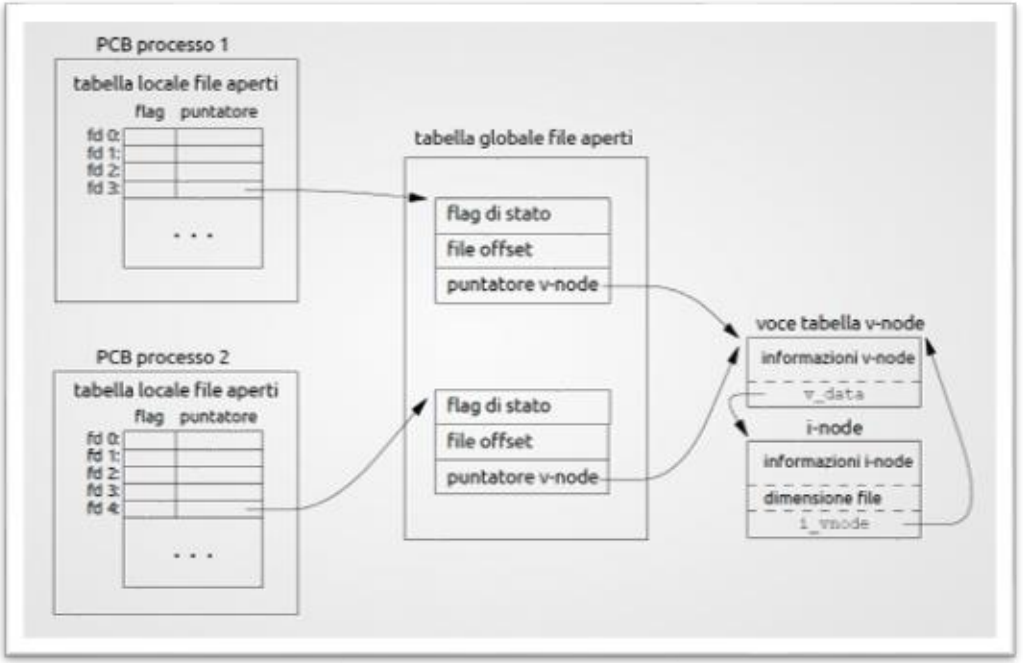
Lettura e Scrittura

```
ssize_t read(int fd, void *buf, size_t nbytes);
ssize_t write(int fd, const void *buf, size_t nbytes);        <unistd.h>
```

- **read** legge **nbytes** byte dal descrittore **fd** mettendoli su buffer **buf**
  - ritorna: -1 in caso di errore, 0 se siamo alla fine del file, altrimenti il numero di byte effettivamente letti (>0)
    - può leggere meno dati se il file sta finendo, se leggendo da terminali, pipe, socket di rete, a causa dei segnali, ...
- **write** legge **nbytes** byte dal buffer **buf** e li scrive sul descrittore **fd**
  - ritorna: -1 in caso di errore o il numero di byte trasferiti (≥0)
- esempi: **count.c**, **hole.c**, **copy.c**

Condivisione di File e Strutture Dati di Supporto

La gestione dei file del Sistema Operativo richiede diverse **strutture dati**:



Letture e Scritture Atomiche

- Ragionando in uno scenario **multi-processo/multi-thread**:
  - ogni voce della tabella globale ha il proprio file offset
  - lo stesso file può essere aperto da più processi
  - i thread condividono la tabella locale del processo e quindi i file offset
- esempio: **accodamento concorrente** di dati (***log file***)
  - multi-processo: i file offset potrebbero non sempre puntare alla fine
    - il flag ***O_APPEND*** garantisce che ogni scrittura avvenga alla fine del file
- esempio: **letture e scritture concorrenti** ad **accesso diretto** sullo stesso file
  - multi-thread: ci possono essere corse critiche interlacciando lseek/read-write
    - è possibile rendere atomiche tali operazioni:

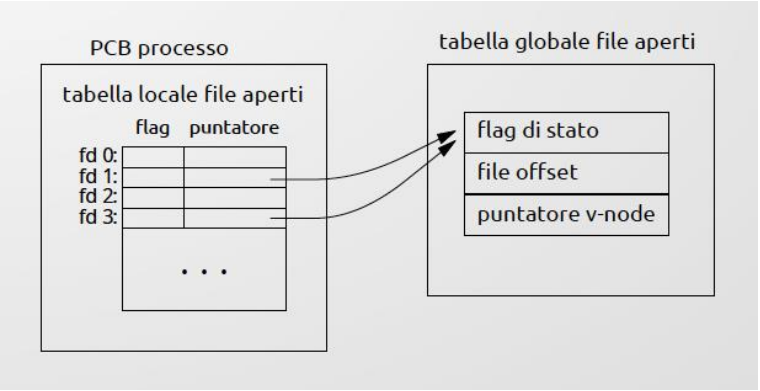
```
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```

<unistd.h>

Duplicazione dei Descrittori di File

```
int dup(int fd);
int dup2(int fd, int fd2);                    <unistd.h>
```

- **dup** duplica una voce della tabella locale usando la prima voce libera
  - descrittore di file disponibile con numero più basso
- **dup2** duplica la voce **fd** usando la voce occupata da **fd2**
  - **fd2** viene chiusa se usata
  - è una **operazione atomica**
- utilizzabili per manipolare i **canali input/output/error standard** in un processo
- esempio: **redirect.c**



Cache del Disco

- Il Sistema Operativo usa la RAM libera come **cache del disco**, anche in **scrittura**
  - ritarda le scritture per ragioni di efficienza (in genere per massimo 30s)
  - questo può creare problemi indesiderati
- è sempre possibile forzare la mano al S.O. tramite:
  - **flag O_SYNC** in fase di apertura di un file
  - **chiamate di sistema** per forzare la scrittura:
    - ritornano solo a scrittura avvenuta

```
int fsync(int fd);
void sync(void);
```
(Demone *update*)

`<unistd.h>`

  - omonimo comando della **shell**: **sync**

I/O Bufferizzato

- Lo standard ISO C fornisce una libreria per l'I/O bufferizzato basato su **stream**
  - cerca di ridurre il numero di chiamate di sistema **read** e **write**
  - tipo di riferimento: **FILE ***
  - stream predefiniti: **stdin**, **stdout** e **stderr**
- esistono vari tipi di buffering:
  - **fully buffered**: in genere usato per i file
  - **line buffered**: in genere usato per i terminali interattivi (stdin e stdout)
  - **unbuffered**: in genere usato per lo standard error (stderr)
- è sempre possibile forzare **scritture pendenti** nel buffer con **fflush**
  - questo non assicura la scrittura su disco: vedi **cache**

Apertura e Chiusura di Stream

```
FILE *fopen(const char *pathname, const char *type);
FILE *fdopen(int fd, const char *type);
int fclose(FILE *fp);
```

- **fopen** e **fdopen** creano uno stream aprendo un file specificato o già aperto
  - **type**: specifica la modalità di apertura usando una stringa
    - **r** – apertura in sola lettura (O_RDONLY)
    - **r+** – apertura in lettura e scrittura (O_RDWR)
    - **w** – creazione/troncatura per scrittura (O_WRONLY | O_CREAT | O_TRUNC)
    - **w+** – creazione/troncatura per lettura/scrittura (O_RDWR | O_CREAT | O_TRUNC)
    - **a** – creazione/apertura in accodamento (O_WRONLY | O_CREAT | O_APPEND)
  - ritorna: NULL in caso di errore o lo stream creato
  - **type** in **fdopen** deve essere coerente con la modalità di apertura di **fd**
- **fclose** chiude lo stream e svuota il buffer

Lettura e Scrittura sugli Stream per Caratteri

- **fgetc** legge un carattere dallo stream
  - ritorna: **EOF** (-1) in caso di errore o fine file, oppure il carattere appena letto (inserito in un int)
    - se interessati, bisogna usare **ferror** e **feof** per disambiguare
- **fputc** scrive un carattere sullo stream
- il loro uso è reso efficiente dal buffering
- esempi:
  - **copy-stream.c**,
  - **streams-and-buffering.c**

```
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
int ferror(FILE *fp);
int feof(FILE *fp);
```

Lettura e Scrittura sugli Stream per Righe

- **fgets** legge una riga dallo stream **fd** e lo scrive come stringa su **buf** di **n** byte
  - **una riga termina con un ritorno a capo ('\n') o dalla fine del file**
  - **vengono effettivamente trasferiti al più (n-1) byte (ritorno a capo incluso)**
  - ritorna: NULL in caso di fine-file/errore o buf in caso di successo
- **fputs** scrive la stringa in **buf** sullo stream **fp**
  - ritorna: **EOF** in caso di errore, un valore non-negativo in caso di successo
- esempio: **my-cat.c**

```
char *fgets(char *buf, int n, FILE *fp);
int fputs(const char *str, FILE *fp);
```

```
int fprintf(FILE *fp, const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
```

Lettura e Scrittura sugli Stream per Blocchi

```
size_t fread(void *buf, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *buf, size_t size, size_t nobj, FILE *fp);
```

- **fread** e **fwrite**, rispettivamente, leggono e scrivono **nobj** record, ciascuno di dimensione **size** byte, sullo stream **fp** dal buffer **buf**
  - ritorna: il numero di record effettivamente trasferiti
  - può riportare meno di nobj record: fine file o errore

funzione	CPU utente (s)	CPU kernel (s)	clock time(s)
read & write con buffer ottimale	0,05	0,29	3,18
fgets & fputs	2,27	0,30	3,49
fgetc & fputc	8,16	0,40	10,18
read & write un byte alla volta	134,61	249,94	394,95

Posizionamento sugli Stream

- **fseek** e **fseeko** spostano il file offset sullo stream
  - parametri coerenti con lseek
- **ftell** e **ftello** riporta direttamente l'attuale file offset associato allo stream
- le **varianti *o** sono suggerite per implementazioni recenti a supporto di grandi file

```
int fseek(FILE *fp, long offset, int whence);
int fseeko(FILE *fp, off_t offset, int whence);
long ftell(FILE *fp);
off_t ftello(FILE *fp);
void rewind(FILE *fp);
```

Raccolta Informazioni sugli Oggetti del File-System

```
int stat(const char *pathname, struct stat *buf );
int fstat(int fd, struct stat *buf );
int lstat(const char *pathname, struct stat *buf );

<sys/stat.h>, <sys/types.h>
```

- **stat** (e vantianti) riporta in **buf** (tipo **stat**) informazioni sull'oggetto riferito
  - **lstat** evita di attraversare i link simbolici
- alcune informazioni che possiamo trovare nella struttura:
  - **st_mode**: informazioni sui permessi di accesso e sul tipo di file
  - **st_uid**, **st_gid**: l'UID dell'utente proprietario e il GID del gruppo proprietario
  - **st_atime**, **st_ctime**, **st_mtime**: il momento (data e orario) dell'ultimo accesso, ultima modifica globale (attributi o contenuto), ultima modifica al contenuto
  - **st_ino**: l'*i-number*, ovvero il numero dell'i-node
  - **st_nlink**: il numero di hardlink all'i-node
  - **st_size**: la dimensione del file in byte

Raccolta Informazioni sugli Oggetti del File-System

- il campo **st_mode** può essere ispezionato in vari modi:
  - la **maschera dei permessi** può essere isolata con: (st_mode & 0777)
  - i flag che denotano il **tipo** di oggetto tramite alcune predicati (macro):
    - **S_ISREG()**: è un file regolare?
    - **S_ISDIR()**: controllo per directory?
    - **S_ISBLK()**: è un dispositivo speciale a blocchi?
    - **S_ISCHR()**: è un dispositivo speciale a caratteri?
    - **S_ISLNK()**: controllo per link simbolico?

Raccolta Informazioni sugli Oggetti del File-System

Per quel che riguarda il campo `st_mode`, si possono usare i flag visti per `creat()` ed `open()` congiuntamente con:

- `S_IFBLK`: dispositivo speciale a blocchi;
- `S_IFDIR`: cartella;
- `S_IFCHR`: dispositivio speciale a caratteri;
- `S_IFFIFO`: una coda FIFO (named pipe);
- `S_IFREG`: file regolare;
- `S_IFLNK`: link simbolico.

Esistono delle maschere per filtrare solo alcuni bit:

- `S_IFMT`: maschera per i flag sul tipo;
- `S_IRWXU`: lettura/scrittura/esecuzione per il proprietario;
- `S_IRWXG`: lettura/scrittura/esecuzione per il gruppo;
- `S_IRWXO`: lettura/scrittura/esecuzione per gli altri.

Raccolta Informazioni sugli Oggetti del File-System

- il c
- la **masch**
- i flag che
- i **ti** pre

Ad esempio, per controllare che un file non sia una directory e che il proprietario abbia solo il diritto di esecuzione (e nessun altro):

```
struct stat statbuf;
stat("filename",&statbuf);

if (!S_ISDIR(statbuf.st_mode) && ( (statbuf.st_mode & S_IRWXU) == S_IXUSR ))...
```

- ulterio
- esempio: **stat.c**

Raccolta Informazioni sugli Oggetti del File-System

- i **timestamp** (time_t) sono interi che possono essere localizzati al fuso predefinito (**localtime**) e convertiti in stringa (**asctime**) con :

```
printf("ultimo accesso: %s\n",asctime(localtime(&(buf.st_atime))));
```

- ulteriori dettagli sull'**utente** e **gruppo** proprietario si possono ottenere usando **getpwuid** e **getgrgid** a partire dai rispettivi dai campi **st_uid** e **st_gid**

- esempio: **stat.c**

Gestione Directory

- **mkdir** crea una cartella con maschera dei permessi **mode**
  - viene applicata anche qui la maschera di **umask**
  - ritorna: -1 in caso di errore, 0 altrimenti
- **rmdir** cancella una directory
  - deve essere vuota (nessun effetto ricorsivo)
  - ritorna: -1 in caso di errore, 0 altrimenti
- **chdir** cambia la ***current working directory*** del processo chiamante
- **getcwd** la riporta nel buffer **buf** di dimensione **size**

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int chdir(const char *pathname);
char *getcwd(char *buf, size_t size);
```

<sys/stat.h>, <unistd.h>

Gestione Directory

- **opendir** apre uno ***directory stream*** (**DIR ***) per la lettura di una directory
  - ritorna: NULL in caso di errore, altrimenti il puntatore all stream creato
- **readdir** legge il prossimo record (**struct dirent ***) dallo stream
  - ritorna: NULL in caso di errore o file elenco, altrimenti il puntatore al record
  - contenuto del record: in numero di i-node **d_ino** e il nome **d_name**
- si possono fare accessi diretti usando **rewinddir**, **telldir** e **seekdir**
- esempio: **list-dir.c**

```
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
long telldir(DIR *dp);
void seekdir(DIR *dp, long loc);
int closedir(DIR *dp);
```

<dirent.h>

Gestione Directory

Per leggere le varie voci di una cartella si deve usare la chiamata:

`struct dirent *readdir(DIR *dir)`

Riporta un puntatore ad una struttura `struct dirent` che contiene le informazioni relative alla voce correntemente letta. Riporta `NULL` in caso di errore e di fine della lista.

Tra le informazioni utili reperibili nella struttura abbiamo:

- `ino_t d_ino`: il numero dell'inode del file;
- `char *d_name`: il nome del file.

- esempio: **list-dir.c** 🗎

Gestione Directory

- si possono fare accessi diretti usando **rewinddir**, **telldir** e **seekdir**

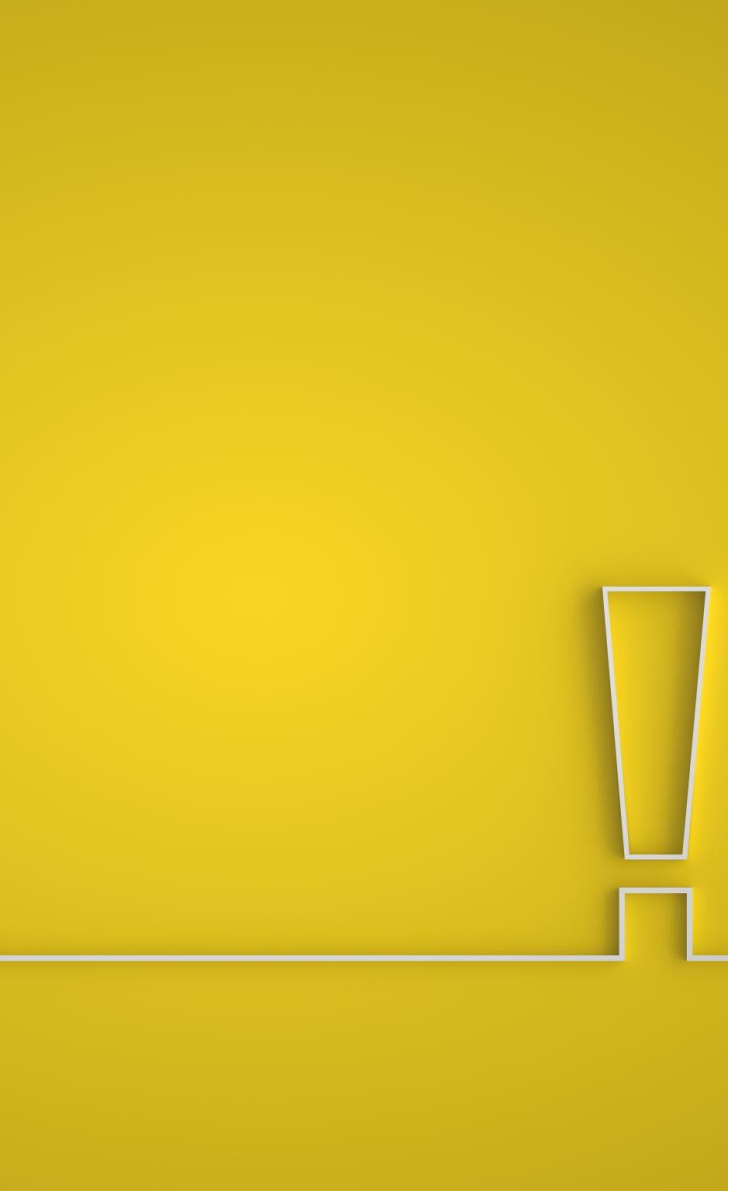
- Alla fine è necessario chiudere la directory con:

  int closedir(DIR *dp)

- esempio: **list-dir.c** 🗎

```
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
void rewinddir(DIR *dp);
long telldir(DIR *dp);
void seekdir(DIR *dp, long loc);
int closedir(DIR *dp);
```

<dirent.h>



Gestione dei Link Simbolici e Fisici

```
int link(const char *existingpath, const char *newpath);
int unlink(const char *pathname);
int remove(const char *pathname);
int rename(const char *oldname, const char *newname);
int symlink(const char *actualpath, const char *sympath);
ssize_t readlink(const char*pathname, char *buf, size_t bufsize);
```

<unistd.h>, <stdlib.h>

- **link** crea un link fisico di un file esistente; **unlink** lo rimuove
- **remove** funziona usa **unlink** su file e **rmdir** su cartelle (vuote)
- **rename** rinomina file e directory
- **symlink** crea un link simbolico a file e cartelle
- **readlink** legge il percorso interno di un link simbolico e lo scrive su **buf**

- esempio: **move.c**

Varie ed Eventuali su File

```
int truncate(const char *path, off_t length);
int ftruncate(int fildes, off_t length);
int chmod(const char *path, mode_t mode);
int chown(const char *path, uid_t owner, gid_t group);
```

<unistd.h>, <sys/stat.h>

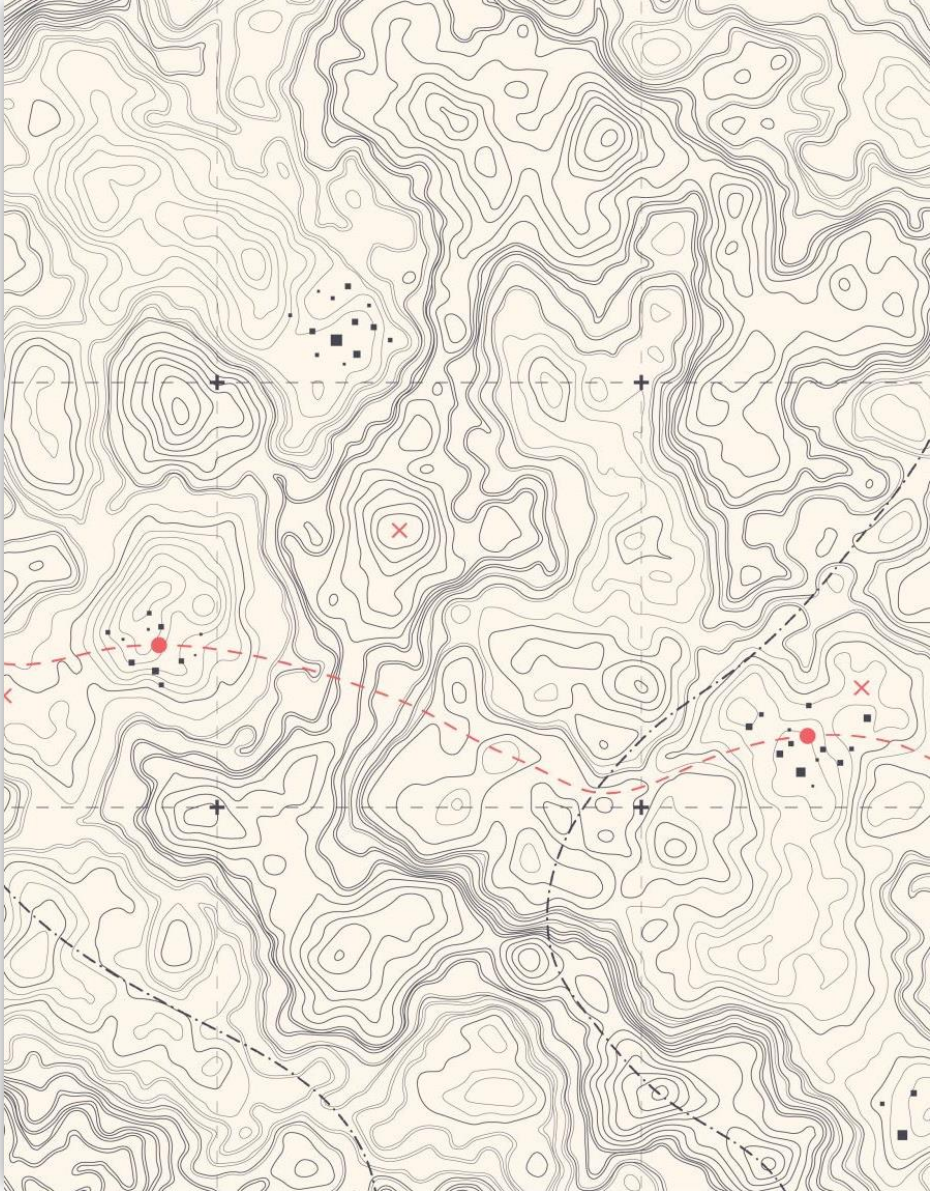
- **truncate** e **ftruncate** troncano un file esistente alla dimensione specificata
  - può anche aumentare la dimensione dei file (vedi ***hole***)
- **chmod** cambia la maschera dei permessi di un oggetto sul file-system
- **chown** cambia l'utente proprietario e il gruppo proprietario specificati tramite i rispettivi identificativi numerici
  - su Linux e altri sistemi UNIX (non tutti), solo l'amministratore può usare chown

Mappatura dei File

```
void *mmap(void *addr, size_t len, int prot, int flag,
int fd, off_t off);
```

<sys/mman.h>

- **mmap** mappa una porzione (definita da **off** e **len**) del file aperto dal descrittore **fd** sull'indirizzo virtuale **addr** abilitando i permessi **prot** sulle relative pagine
  - se **addr** è NULL il Sistema Operativo trova un indirizzo idoneo
  - **prot** è una combinazione di **PROT_READ**, **PROT_WRITE** e **PROT_EXEC**
  - **flag**:
    - **MAP_SHARED**: scritture applicate sul file e condivise con altri processi
    - **MAP_PRIVATE**: scritture private (*CoW*) e non persistenti
  - ritorna: **MAP_FAILED** in caso di errore, l'indirizzo di mappatura altrimenti



Mappatura dei File

```
int msync(void *addr, size_t len, int flag);
int munmap(void *addr, size_t len);
```

<sys/mman.h>

- **msync** forza il Sistema Operativo a scrivere su disco eventuali modifiche in sospeso nell'area mappata specificata da **addr** e **len**
  - **flag**:
    - **MS_ASYNC**: richiesta asincrona
    - **MS_SYNC**: richiesta sincrona (bloccante)
- **munmap** annulla la mappatura del file, salvando le eventuali modifiche in caso di mappatura condivisa (MAP_SHARED)
- effetti comunque applicati alla terminazione del processo
- esempi: **mmap-read.c** 🗎, **mmap-copy.c** 🗎, **mmap-reverse.c** 🗎

Creazione di Processi

```
pid_t getpid(void);
pid_t getppid(void);
pid_t fork(void);

<unistd.h>
```

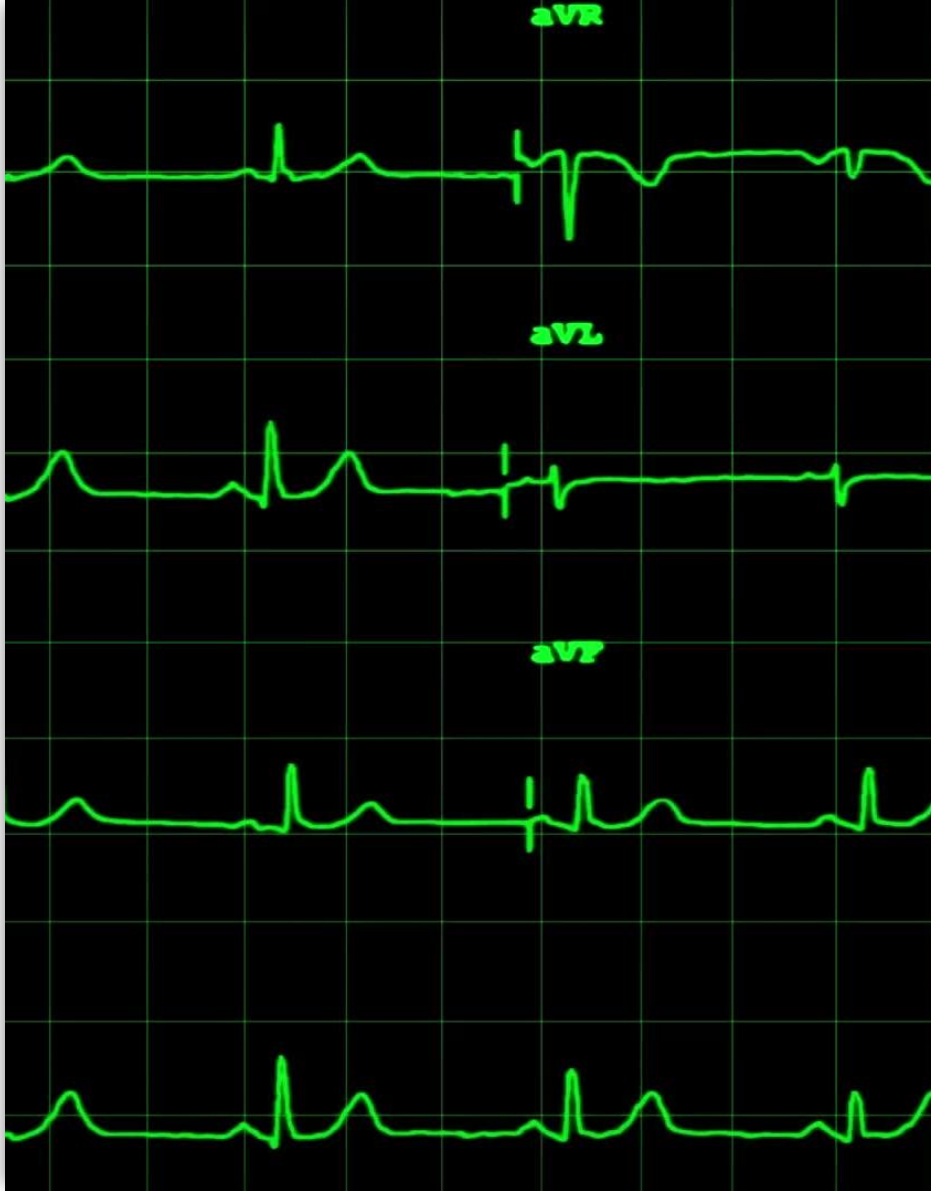
- **getpid** e **getppid** ritornano, rispettivamente, il ***process id*** (PID) del *processo chiamante* e del *processo padre*
- **fork** duplica il processo chiamante
  - **ritorna**:
    - nel padre: -1 in caso di errore, il PID del processo figlio appena creato altrimenti
    - nel figlio: il valore 0
- un processo figlio, rimasto **orfano**, viene comunque adottato

- esempi: **fork.c** 🗎, **fork-buffer-glitch.c** 🗎, **multi-fork.c** 🗎

Coordinamento Semplice tra Processi

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
                                                <sys/wait.h>
```

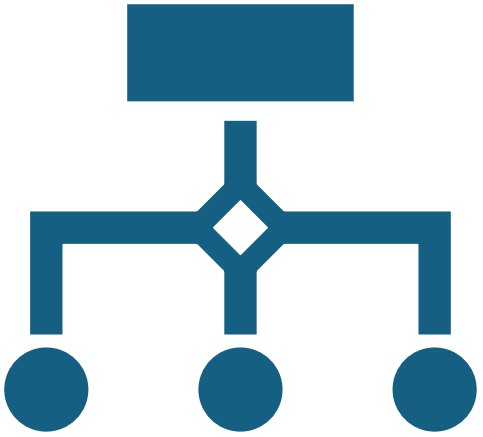
- **wait** e **waitpid** permettono al padre di bloccarsi in attesa della terminazione di, rispettivamente, un qualunque figlio o uno specifico indicandone il **pid**
  - **statloc**, se diverso da NULL, deve puntare ad un intero su cui sarà scritto l'**exit status** del figlio appena terminato:
    - **exit code** nella parte bassa (estraibile con la macro **WEXITSTATUS**)
    - vari flag ispezionalibili sul motivo esatto dell'uscita
  - **options** può specificare modalità particolari che possiamo ignorare (0)
  - ritornano: -1 in caso di errore, il PID del figlio altrimenti
- un figlio rimane nello stato di **zombie/defunct** se termina prima del padre: quest'ultimo può, potenzialmente, richiederne l'exit status con wait/waitpid
- esempio: **multi-fork-with-wait.c**



Esecuzione di un Programma

```
int execl(const char *pathname, const char *arg0, …, (char *)0);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *pathname, const char *arg0, …, (char *)0);
int execvp(const char *pathname, char *const argv[]);
                                                          <unistd.h>
```

- una chiamata **exec*** esegue il programma specificato da **pathname** su una lista di argomenti **arg*** usando il processo chiamante come ambiente
  - la variante **execl** passa gli argomenti come parametri della funzione con l'obbligatoria sentinella (char *)0
  - la variante **execv** usa un vettore di stringhe con una sentinella nell'ultimo slot
- le **varianti *p**, su pathname non assoluti, ricercano l'eseguibile nei percorsi previsti nella variabile d'ambiente **PATH**
  - ritorna: -1 in caso di errore, altrimenti… **non torna!**
- esempi: **exec.c** 🗎, **nano-shell.c** 🗎



Esecuzione per Interpretazione e Esecuzione Semplificata

- Sui sistemi UNIX un **file testuale** può essere reso eseguibile per **interpretazione**
  - deve avere l'apposito flag/**permesso di esecuzione** (x) attivo
  - deve specificare nella **prima riga** l'interprete da usare
    - convenzione: **#! interprete [eventuali argomenti]**
      - molto usato:
        - #! /usr/bin/bash
        - #! /usr/bin/python3
        - #! /usr/bin/perl
        - ...
  - gestito direttamente dal kernel
- per eseguire un comando in un sotto-processo si può anche usare **system**: tramite fork/waitpid/exec esegue una **shell** a cui viene passata la richiesta

```
int system(const char *cmdstring);          <stdlib.h>
```

- esempio: system(“comando argomento1 argomento2 > output.txt“);

Segnali

- Usati dal sistema per notificare ai processi **eventi** di varia natura
  - reazioni: **ignorare**, **terminare** o **eseguire** una procedura apposita
- segnali principali:
  - **hangup** + (SIGHUP): perdita del terminale locale/remoto
  - **interrupt** + (SIGINT): interruzione interattiva da terminale (CTRL+C)
  - **termination** + (SIGTERM): richiesta di terminazione
  - **kill** x (SIGKILL): interruzione forzata
  - **illegal instruction** x (SIGILL), **segment. violation** x (SIGSEGV), … : errori fatali
  - **child death** * (SIGCHLD): figlio terminato
  - …
    - +: porta alla terminazione ma è ignorabile/gestibile
    - x: porta inevitabilmente alla terminazione
    - *: non implica terminazione (ignorato)

Invio Segnali

```
int kill(pid_t pid, int signo);
int raise(int signo);
```

<signal.h>

- **kill** invia un segnale, con codice **signo**, al processo di identificativo **pid**
  - diversamente da quanto suggerito dal nome, serve ad inviare un qualunque segnale
  - deve essere un nostro processo o dobbiamo usare i diritti di amministratore
- **raise** invia un segnale al processo chiamante stesso

Invio Segnali

```
int kill(pid_t pid, int signo);
```

pid > 0	The signal is sent to the process whose process ID is *pid*.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term *all processes* excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and `init` (pid 1).
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.
pid == –1	The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

I POSIX Thread (a.k.a. Pthread)

- Forniscono una **interfaccia standard** per interagire con le varie implementazioni disponibili sui sistemi POSIX compatibili
- **identificativo** di un thread
  - tipo: **pthread_t** (intero non negativo)
  - univoco solo nel contesto del processo contenitore
  - significato specifico alla piattaforma (intero, puntatore, ...)
  - incapsulamento tramite funzioni elementari:

```
pthread_t pthread_self(pvoid);
int pthread_equal(pthread_t tid1, pthread_t tid2);
                                                    <pthread.h>
```

- oltre all'inclusione dell'header **pthread.h**, è necessario effettuare il **linking** all'apposita libreria:
  - **gcc -l pthread -o eseguibile *sorgente.c***

Creazione Thread

```
int pthread_create(pthread_t *tidp, const pthread_attr_t *attr,
void *(*thread_func)(void *), void *thread_arg);
```

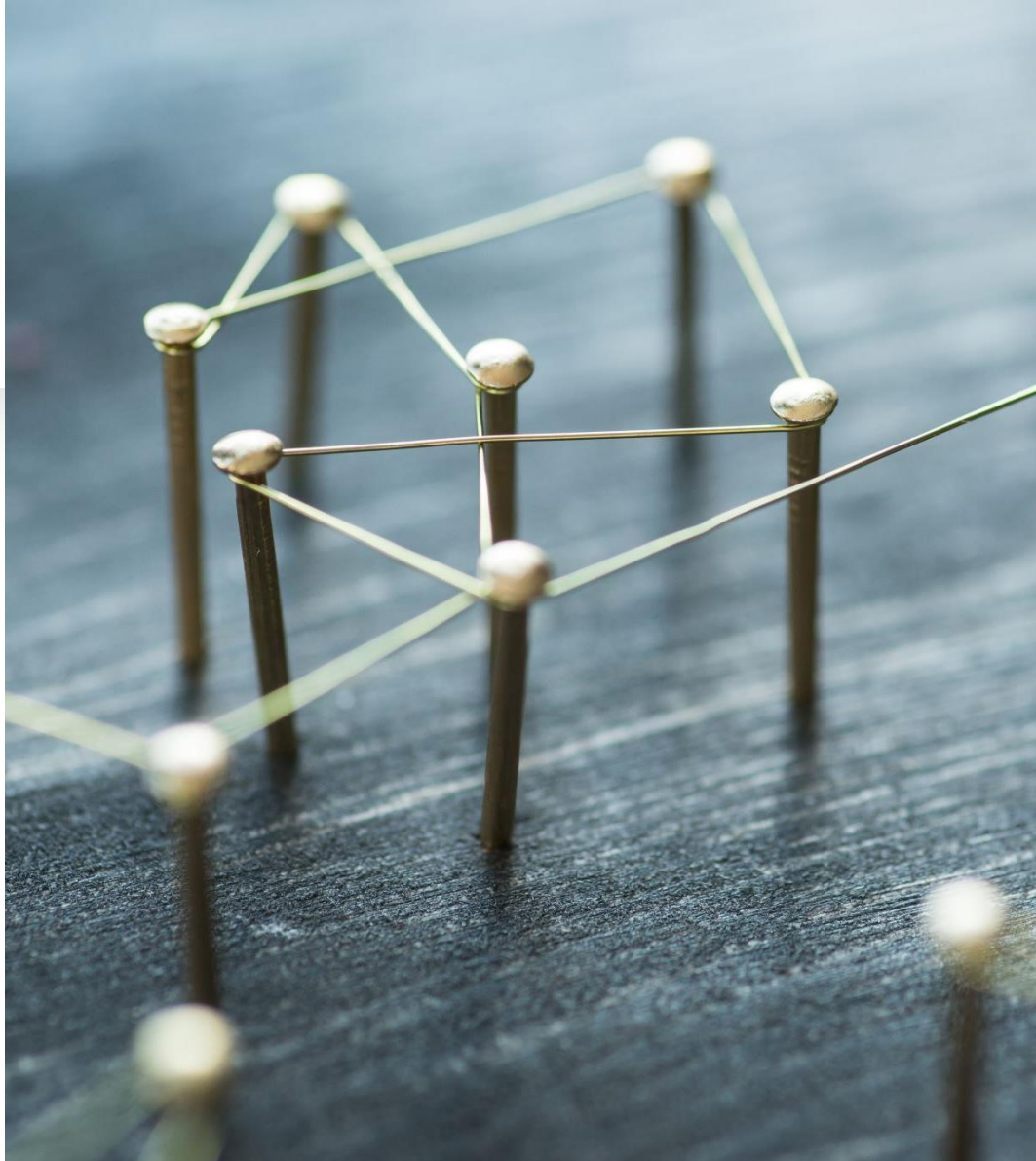
<pthread.h>

- **pthread_create** crea un nuovo thread che eseguirà la funzione **thread_func** con argomento **thread_arg**:
  - prototipo standard: **void *_funzione_(void *_argomento_) { /* corpo funz. */ }**
  - **stack** dedicato creato automaticamente
  - identificativo del thread depositato su ***tidp**
  - attributi particolari opzionali in **attr** (vedremo dopo): al momento NULL
  - ritorna: 0 in caso di successo, il codice d'errore (>0) altrimenti
    - questa è una **convenzione** delle funzioni in pthread: non usa **errno**
- esempio: **thread-ids.c**

Coordinamento Semplice tra Thread

```
void pthread_exit(void *rval_ptr);
int pthread_join(pthread_t thread, void **rval_ptr);
                                                    <pthread.h>
```

- **pthread_exit** termina il thread chiamante
  - equivale ad un **return** dalla funzione principale del thread
  - il processo rimane attivo finché c'è un thread o qualcuno chiama **exit**
  - il valore **rval_ptr** codifica il ***return value*** del thread (contenuto libero)
- **pthread_join** attende la terminazione di uno specifico thread (simile a **wait**)
  - diventa importante conservare il ***thread id*** ottenuto da pthread_create
  - *return value* del thread appena terminato depositato in ***rval_ptr**
  - ritorna: 0 in caso di successo, il codice d'errore (>0) altrimenti
- esempi: **multi-thread-join.c** 🗎**, thread-memory-glitch.c** 🗎



Mutex Lock

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

<pthread.h>

- la struttura **pthread_mutex_t** va usata da tutti i thread come riferimento al lock
- **pthread_mutex_init** inizializza dinamicamente **mutex** con eventuali attributi **attr** (vedremo dopo, per ora NULL)
  - per inizializzare istanze statiche si può usare **PTHREAD_MUTEX_INITIALIZER**
- **pthread_mutex_destroy** è necessario se inizializzato dinamicamente con _init
- **pthread_mutex_lock** e **phtread_mutex_unlock** acquisiscono e rilasciano il lock
- **pthread_mutex_trylock** è non bloccante e ritorna subito con **EBUSY** se non riesce
- esempio: **thread-conc-problem-fixed-with-mutex.c**

Semafori Numerici

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
```

<semaphore.h>

- la struttura dati di riferimento è **sem_t**
- **sem_init** inizializza il semaforo sem con il valore iniziale value
  - **pshared** dichiara il tipo di utilizzatori per adattare il meccanismo di sincronizzazione:
    - **PTHREAD_PROCESS_PRIVATE (0)**: tra thread dello stresso processo
    - **PTHREAD_PROCESS_SHARED (1)**: tra processi distinti tramite memoria condivisa
- **sem_wait** decrementa il semaforo dove **sem_trywait** lo fa senza bloccarsi
- **sem_post** incrementa il semaforo
- esempio: **thread-prod-cons-with-sem.c**

Lock per Lettori/Scrittori

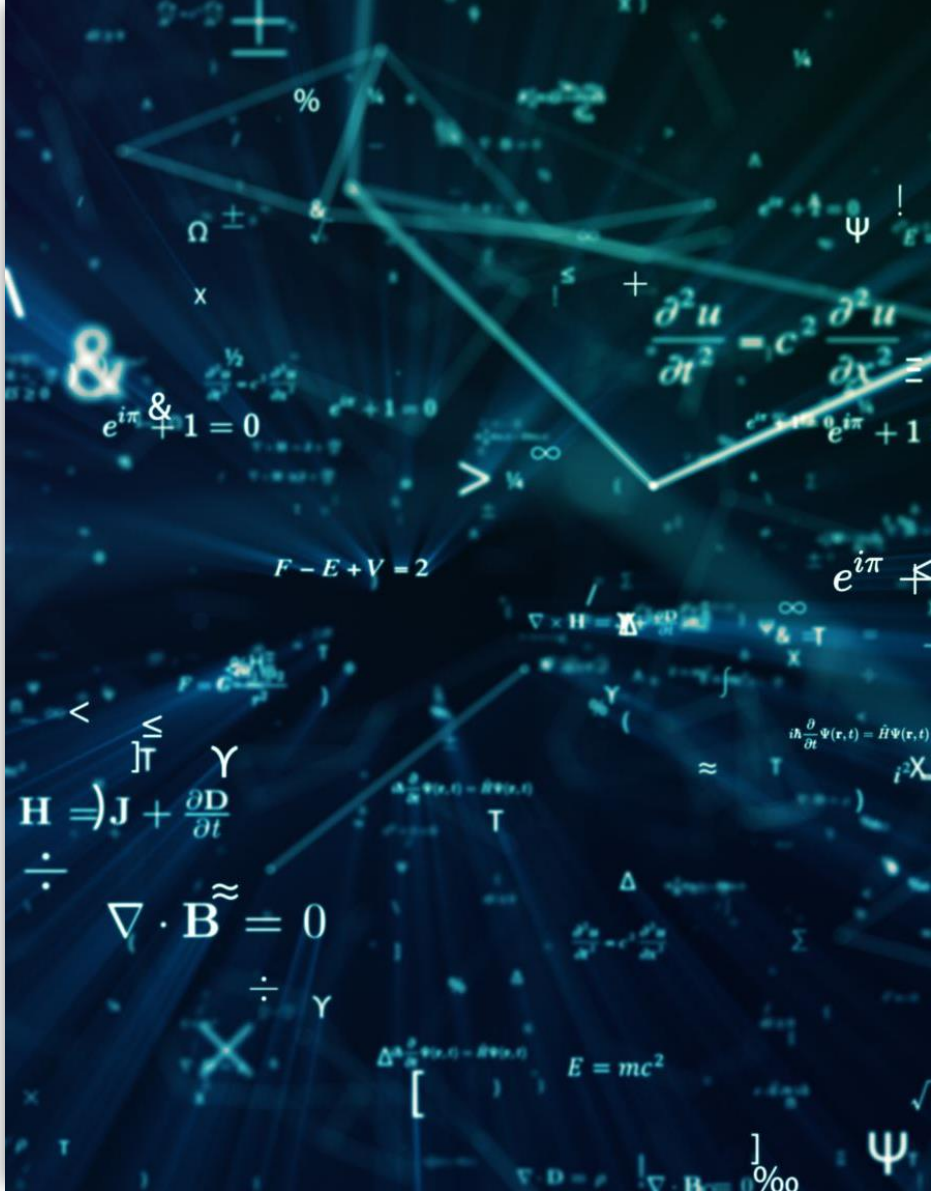
- la struttura di riferimento è **pthread_rwlock_t**
- **pthread_rwlock_{init,destroy}** sono simili agli analoghi visti prima per i mutex
  - anche qui per le istanze statiche esiste **PTHREAD_RWLOCK_INITIALIZER**
- **pthread_rwlock_[try]{rd|wr}lock** acquisiscono il lock condiviso o esclusivo
- **pthread_rwlock_unlock** rilascia il lock precedentemente acquisito
- esempi: **thread-number-set-with-rwlock.c** 🗎, **thread-safe-number-set-with-rwlock.c** 🗎

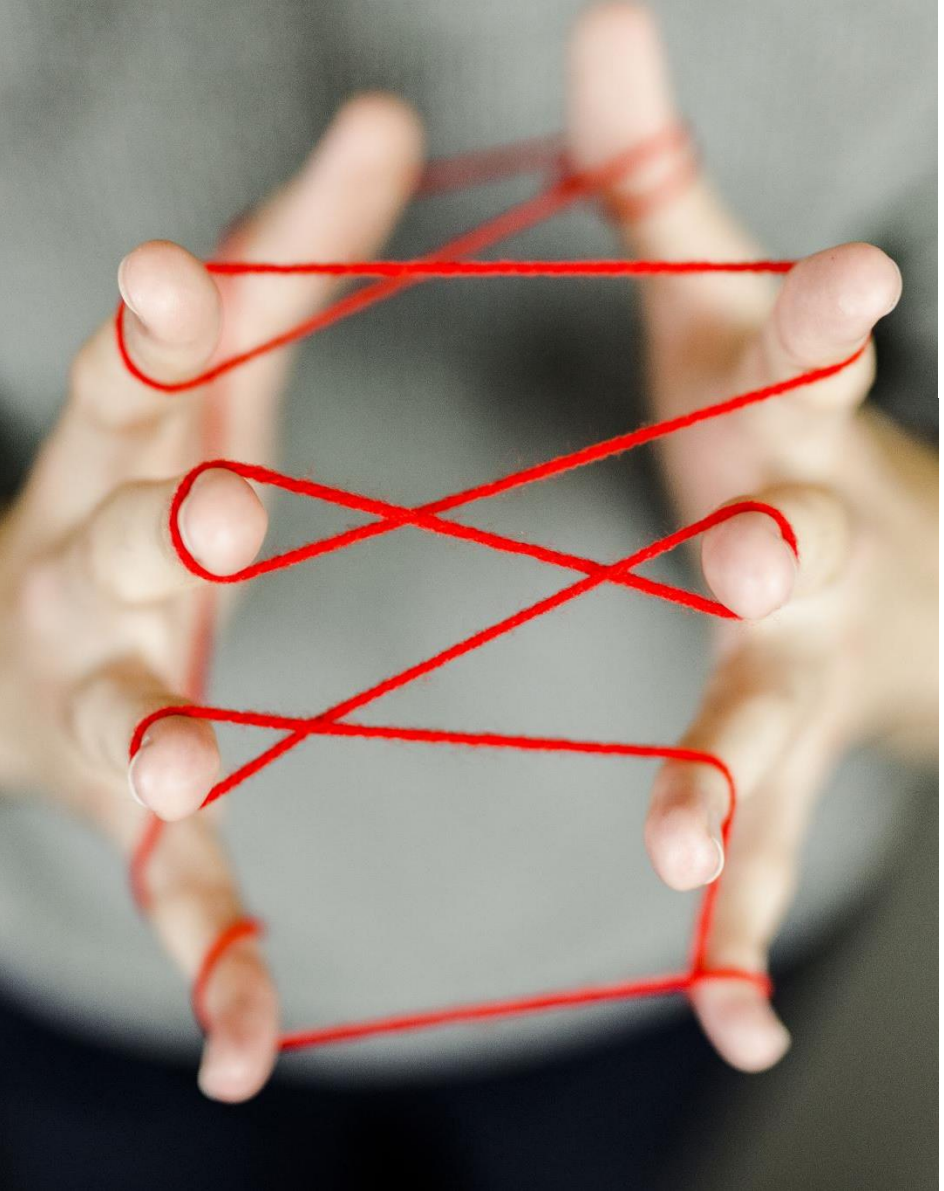
```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

                                                <pthread.h>
```

Variabili Condizione dei Monitor

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t
*attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);                <pthread.h>
```

- la struttura di riferimento è **pthread_cond_t**
- l'uso è sempre contestuale ad un mutex lock acquisito (come nei monitor)
- **pthread_cond_{init,destroy}** crea e distrugge una variabile condizione
  - anche qui per le istanze statiche esiste **PTHREAD_COND_INITIALIZER**
- **pthread_cond_wait** si blocca il chiamante sulla variabile condizione cond
- **pthread_cond_{signal,broadcast}** risvegliano uno o più thread bloccati
  - la condizione di blocco va **sempre** ricontrollata alla ripresa!
- esempio: **thread-safe-number-queue-as-monitor.c**





Barriere

```
int pthread_barrier_init(pthread_barrier_t *barrier,
const pthread_barrierattr_t *attr, unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
                                                <pthread.h>
```

- la struttura di riferimento è **pthread_barrier_t**
- **pthread_barrier_init** crea una barriera per **count** thread
- **pthread_barrier_wait** diventa bloccante per il chiamante finché non si raggiunge la soglia prestabilita di thread bloccati
  - ritorna: 0 o PTHREAD_BARRIER_SERIAL_THREAD a sblocco avvenuto, o errore (>0)
  - solo un thread riceverà **PTHREAD_BARRIER_SERIAL_THREAD** (-1): può essere usato come coordinatore per le fasi successive
- la barriera è **riutilizzabile** ma mantenendo il numero di thread
- esempi: **thread-barrier.c**, **thread-sort-with-barrier.c**