

Sistemi Operativi

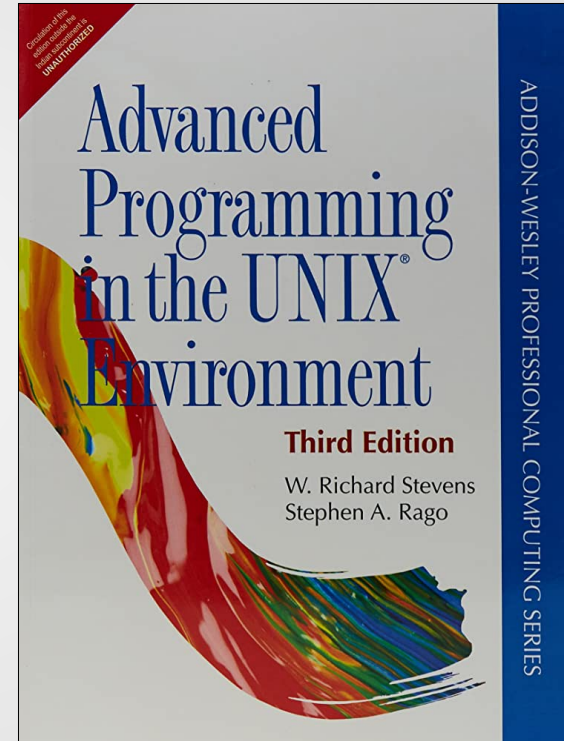
C.d.L. in Informatica (laurea triennale)
Anno Accademico 2022-2023

Laboratorio di Sistemi Operativi

Dipartimento di Matematica e Informatica – Catania

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



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`:

<code>S_IRUSR</code>	<code>S_IWUSR</code>	<code>S_IXUSR</code>
<code>S_IRGRP</code>	<code>S_IWGRP</code>	<code>S_IXGRP</code>
<code>S_IROTH</code>	<code>S_IWOTH</code>	<code>S_IXOTH</code>

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

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** posiziona 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 e valido solo su file
- ottenere la posizione attuale: `pos = lseek(fd, 0, SEEK_CUR);`
- esempio: **test-seek-on-stdin.c** 

Letture 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`

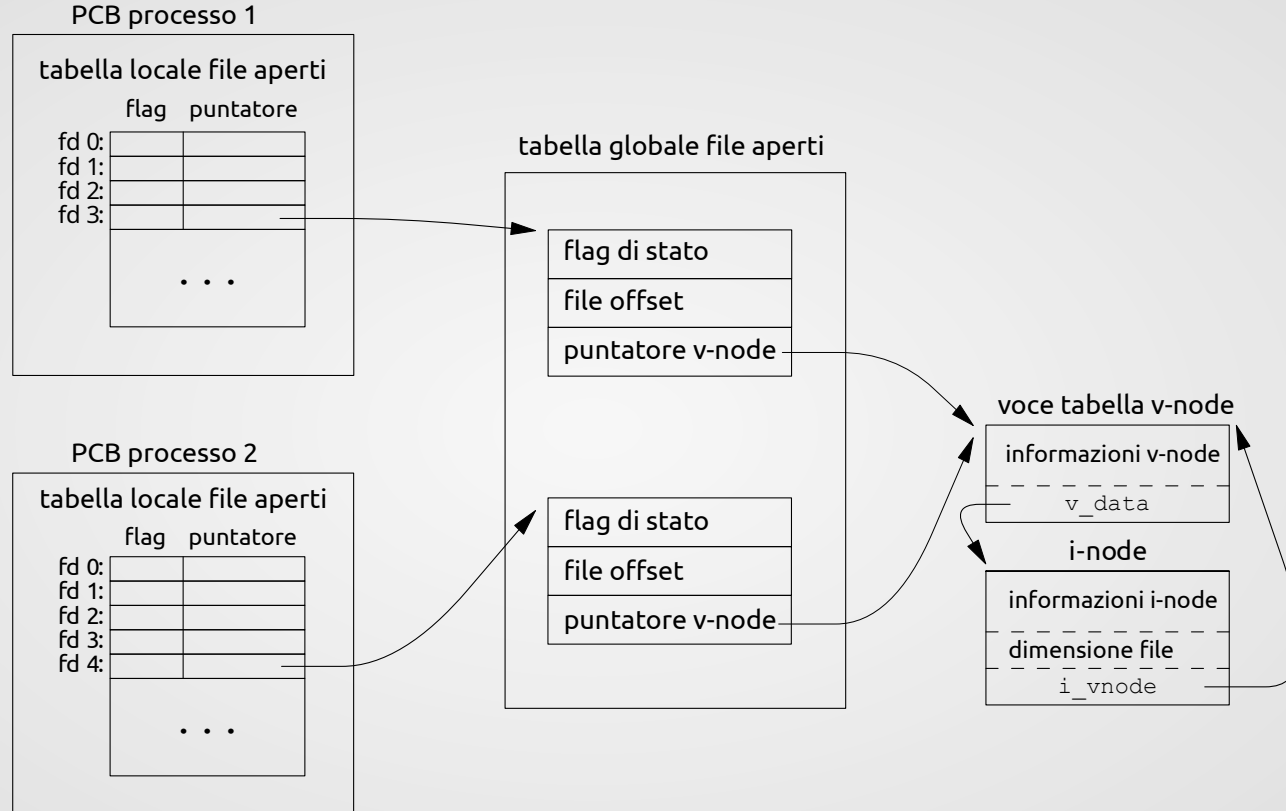
Efficienza dell'I/O su File

dimens. buffer	CPU utente (s)	CPU kernel (s)	clock time (s)	interazioni
1	20,03	117,50	138,73	516.581.760
2	9,69	58,76	68,60	258.290.880
4	4,60	36,47	41,27	129.145.440
8	2,47	15,44	18,38	64.572.720
16	1,07	7,93	9,38	32.286.360
32	0,56	4,51	8,82	16.143.180
64	0,34	2,72	8,66	8.071.590
128	0,34	1,84	8,69	4.035.795
256	0,15	1,30	8,69	2.017.898
512	0,09	0,95	8,63	1.008.949
1.024	0,02	0,78	8,58	504.475
2.048	0,04	0,66	8,68	252.238
4.096	0,03	0,58	8,62	126.119
8.192	0,00	0,54	8,52	63.060
16.384	0,01	0,56	8,69	31.530
32.768	0,00	0,56	8,51	15.765
65.536	0,01	0,56	9,12	7.883
131.072	0,00	0,58	9,08	3.942
262.144	0,00	0,60	8,70	1.971
524.288	0,01	0,58	8,58	986

Trasferendo un file da 500 MB, usando buffer di dimensione diversa e un file-system con blocchi da 4KB.

Condivisione di File e Strutture Dati di Supporto

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



Lecture 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>

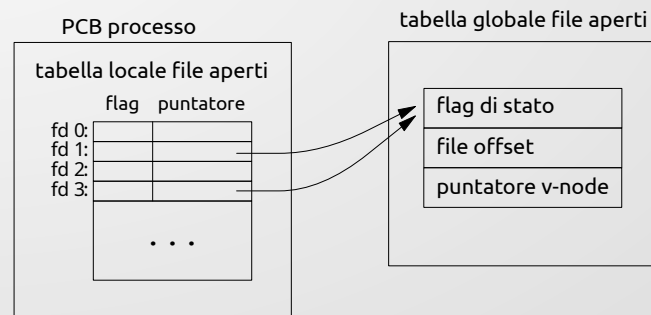
Duplicazione dei Descrittori di File

```
int dup(int fd);  
int dup2(int fd, int fd2);
```

<unistd.h>

- **dup** duplica una voce della tabella locale usando la prima voce libera
- **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**

Eredita le stesse modalità di accesso



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 30 s)
 - 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); 
```

<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);
```

associa uno stream ad un esistente file descriptor

<stdio.h>

- **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

NOTA: quando viene aperto uno stream viene creato il relativo descrittore del file nella tabella dei file aperti.

Lettura e Scrittura sugli Stream per Caratteri

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

<stdio.h>

- **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**

Lettura e Scrittura sugli Stream per Righe

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

<stdio.h>

- **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** 

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

<stdio.h>

Lettura e Scrittura sugli Stream per Blocchi

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

<stdio.h>

- **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** → No byte!!!
 - ♦ 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

```
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);  → "riavvolge lo stream", portando la posizione  
all'inizio dello stream
```

<stdio.h>

- **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

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 );
```

 → raccoglie alcune informazioni su un file
 → raccoglie le informazioni su un file già aperto
 → raccoglie le informazioni sul link simbolico stesso

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

- **stat** (e varianti) 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?
- i **timestamp** (`time_t`) sono interi che possono essere localizzati al fuso predefinito (`localtime` 
- ulteriori dettagli sull'**utente e gruppo** proprietario si possono ottenere usando `getpwnid`  e `getgrgid`  a partire dai rispettivi dai campi `st_uid` e `st_gid`
- esempio: `stat.c` 

Gestione Directory

```
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>

- **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**

Gestione Directory

```
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>

- **opendir** apre uno *directory stream* (DIR *) per la lettura di una directory
 - ritorna: NULL in caso di errore, altrimenti il puntatore allo 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** 

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 bufsiz);
```

<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>

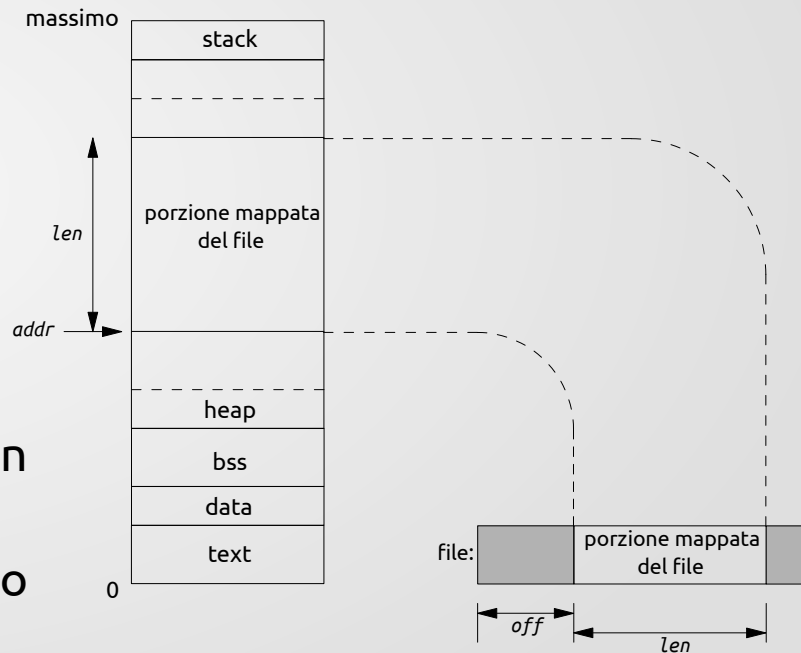
- **truncate** e **ftruncate** troncano un file esistente alla dimensione specificata
 - può anche aumentare la dimensione dei file (vedi **hole**)
- **chmod** cambia la maschera dei permessi di un oggetto sul file-system
- **chown** cambia l'utente proprietario e il gruppo proprietario specificati tramite 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 → *modifiche non condivise e volatili*
 - 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

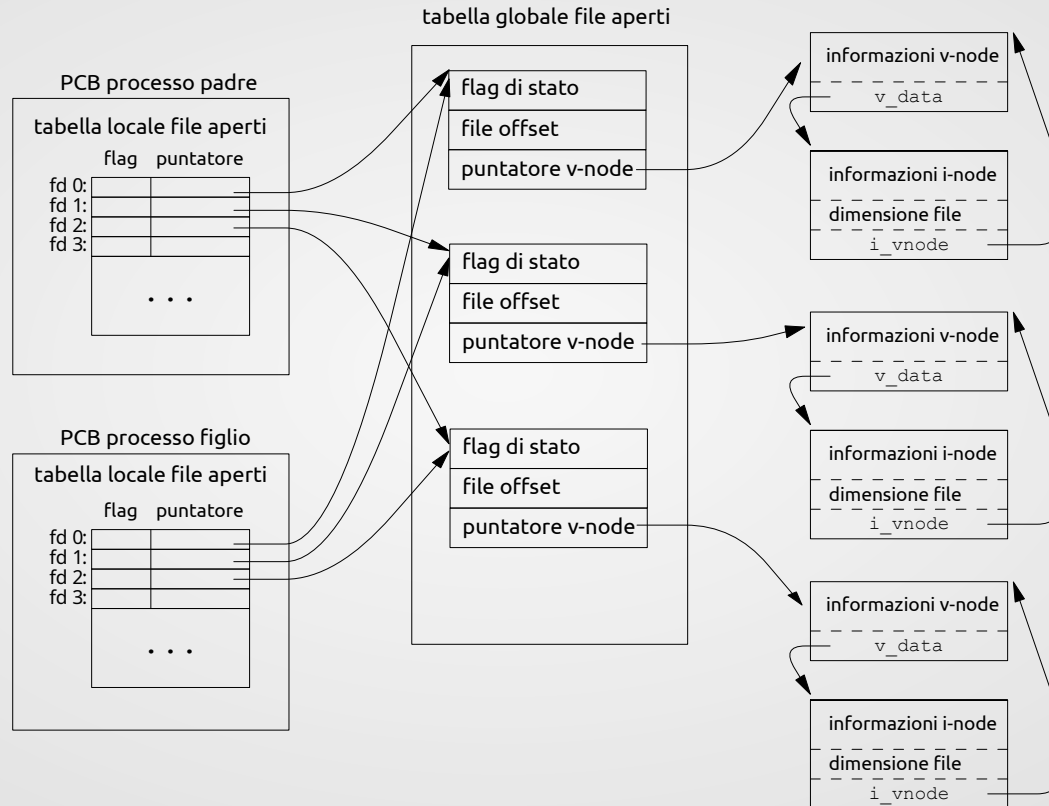
```
pid_t getpid(void);  
pid_t getppid(void);  
pid_t fork(void);
```

<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**

Fork e Tabelle dei File Aperti

- Il processo padre e quello figlio condividono le voci della **tabella globale dei file aperti** e quindi i **flag di apertura** e i **file offset**:



Coordinamento Semplice tra Processi

```
pid_t wait(int *statloc);
```

```
pid_t waitpid(pid_t pid, int *statloc, int options);
```

coordinare il lavoro del processo padre
con quello dei processi figli

<sys/wait.h>

- **wait** e **waitpid** permettono al padre può 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**

estrapolare il solo
exit status