

Sistemi Operativi

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

Anno Accademico 2023-2024

Laboratorio di Sistemi Operativi (MZ)

Prof. Mario Pavone

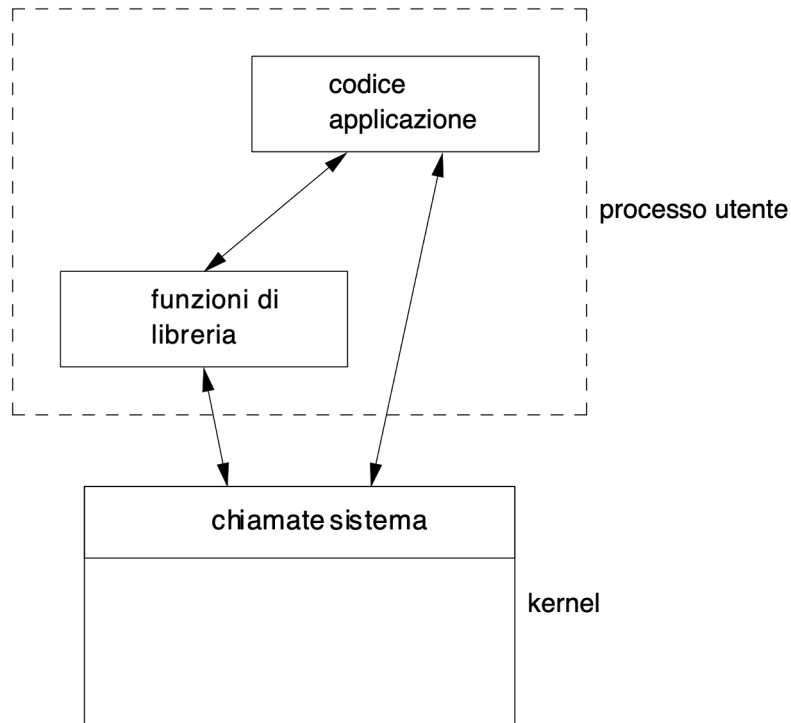
Dipartimento di Matematica e Informatica

Università degli Studi di Catania

mario.pavone@unict.it

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



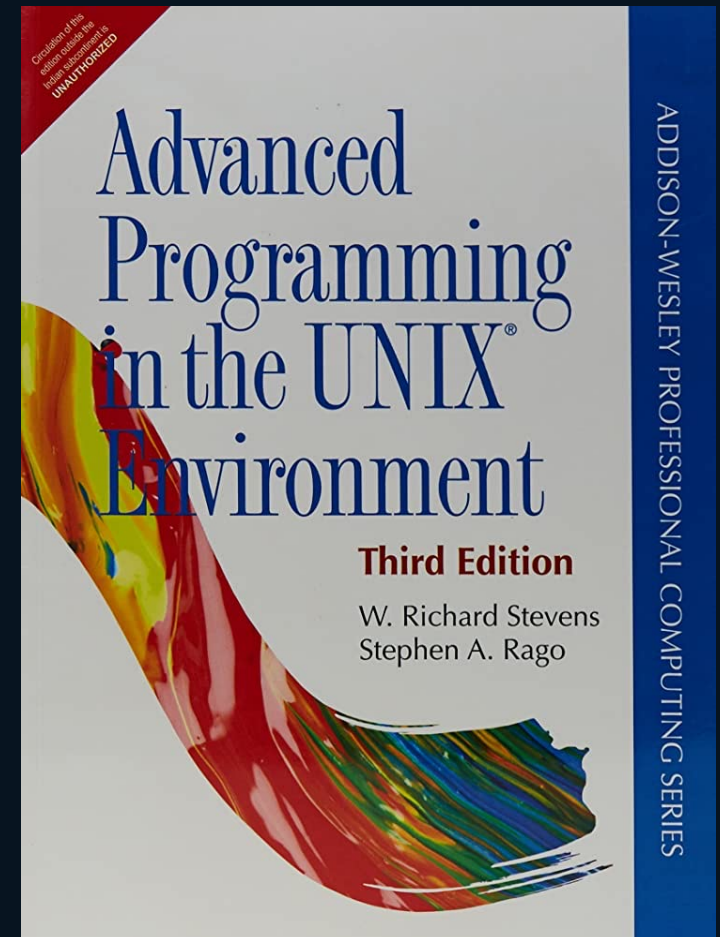


Chiamate di Sistema

- 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 sottosistemi:
 - 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



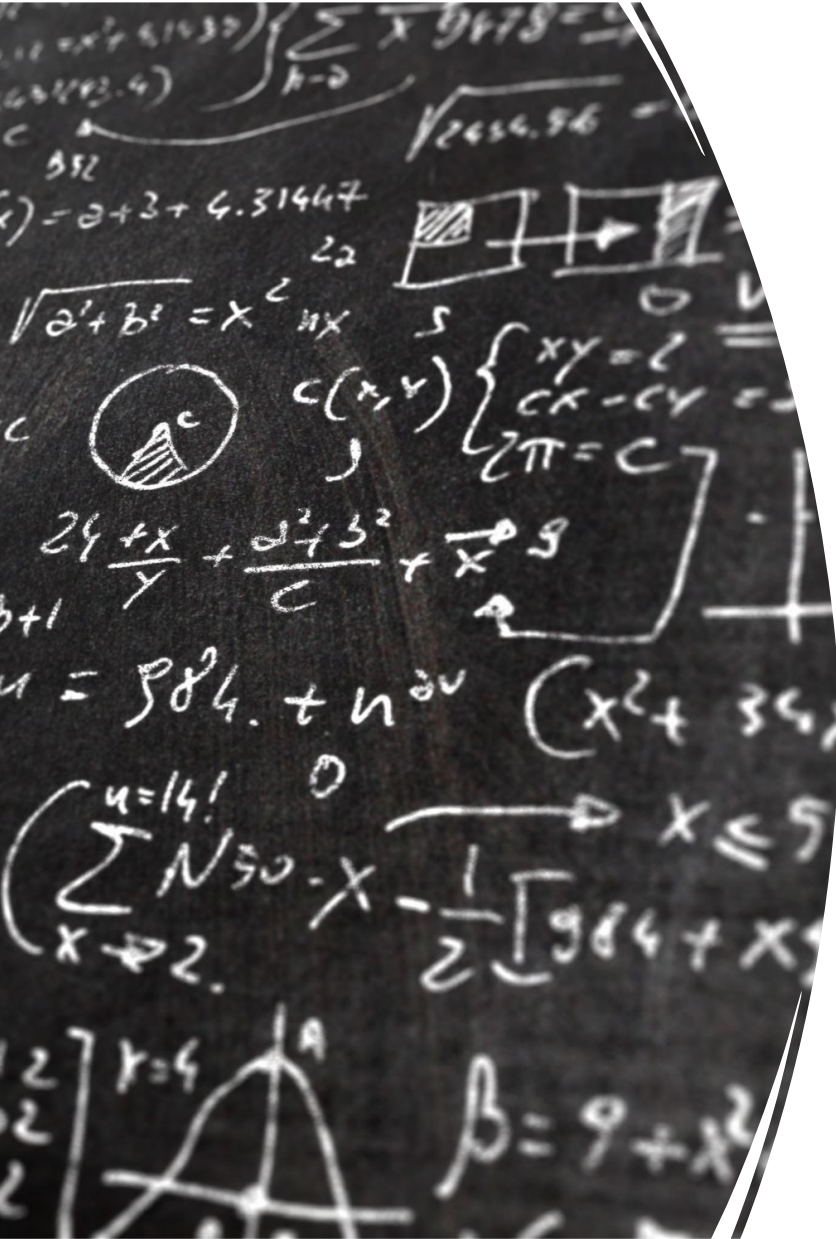


Documentazione

- 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**

Standard

- L'uso degli **standard** è importante per creare codice che sia **portatile** (previa compilazione) su molteplici piattaforme e architetture.
 - **ISO C**: linguaggio e funzioni di libreria
 - **IEEE POSIX** (Portable Operating System Interface) Std 1003.1 e estensioni:
 - POSIX.1: interfacce di programmazione con sintassi C (sovrapp. con ISO C)
 - POSIX.2: comandi e utilità sulla shell UNIX
 - POSIX.4: estensioni real-time (tra cui i thread)
 - POSIX.7: amministrazione di sistema
- **Supporto:**
 - **GNU/Linux**: alquanto completo (**piattaforma di riferimento** per il laboratorio)
 - con alcune estensioni **GNU** specifiche attive di default
 - si può forzare lo standard POSIX con: `#define _POSIX_C_SOURCE 200809L`
 - **Windows**: parziale ma ampliabile con **Windows Subsystem for Linux** (WSL)
 - **MacOS**: alquanto completo



Creazione di Programmi Esecuibili

- 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 eventualmente 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_IRGRP
S_IROTH

S_IWUSR
S_IWGRP
S_IWOTH

S_IXUSR
S_IXGRP
S_IXOTH

- 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**

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;

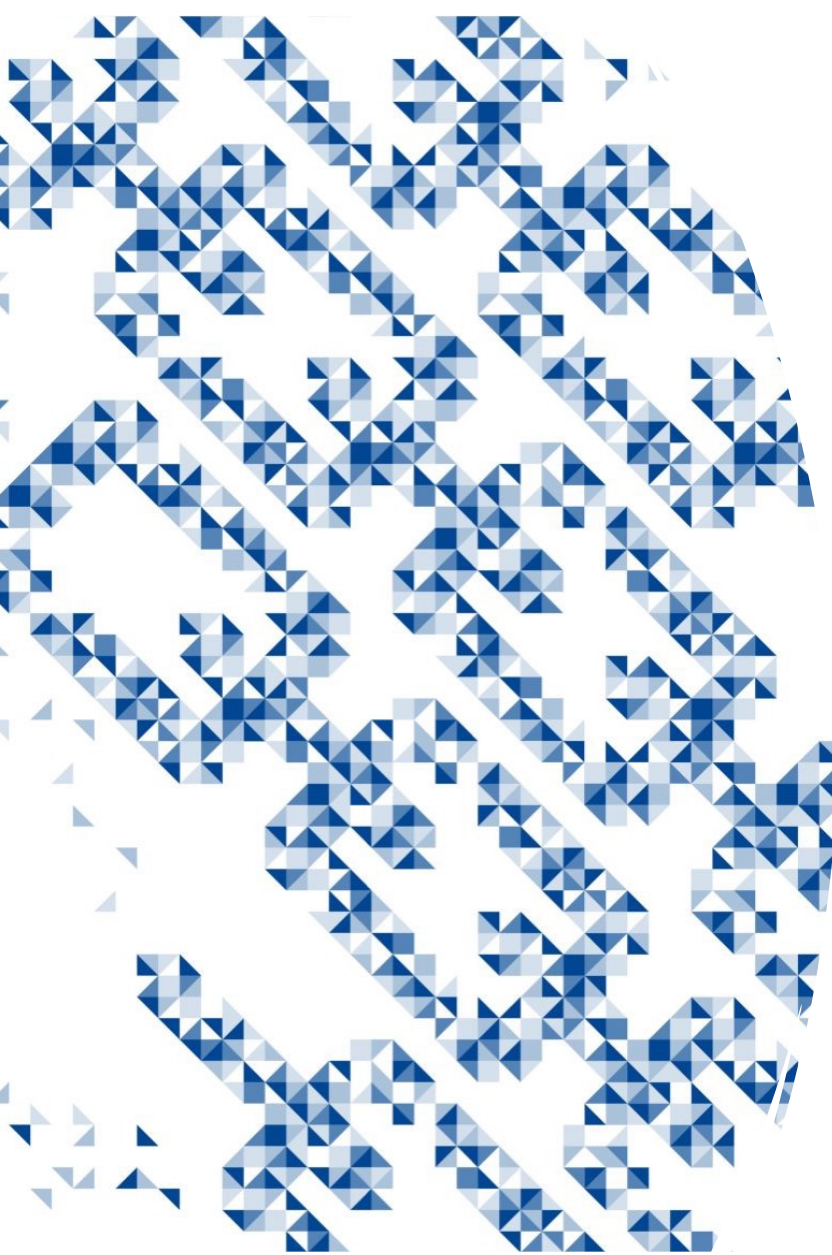
    nmask = S_IRUSR | S_IWUSR | /* owner read write */
           S_IRGRP | S_IWGRP | /* group read write */
           S_IROTH;           /* other read */
    omask = umask( nmask );
    printf( "Mask changed from %o to %o\n",
           omask, nmask );
    return EXIT_SUCCESS;
}
```

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** 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** 



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** 📄

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	26,47	31,07	129.145.440

**Domanda: Come scegliamo
il valore BUFSIZE?**

85.550	0,01	0,58	0,12	7.885
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

Efficienza dell'I/O su File

dimens. buffer	CPU utente (s)	CPU kernel (s)	clock time (s)	interazioni
<pre>#include "apue.h" #define BUFSIZE 4096 int main(void) { int n; char buf[BUFSIZE]; while ((n = read(STDIN_FILENO, buf, BUFSIZE)) > 0) if (write(STDOUT_FILENO, buf, n) != n) err_sys("write error"); if (n < 0) err_sys("read error"); exit(0); }</pre>				
524.288	0,01	0,58	8,58	986

Efficienza dell'I/O su File

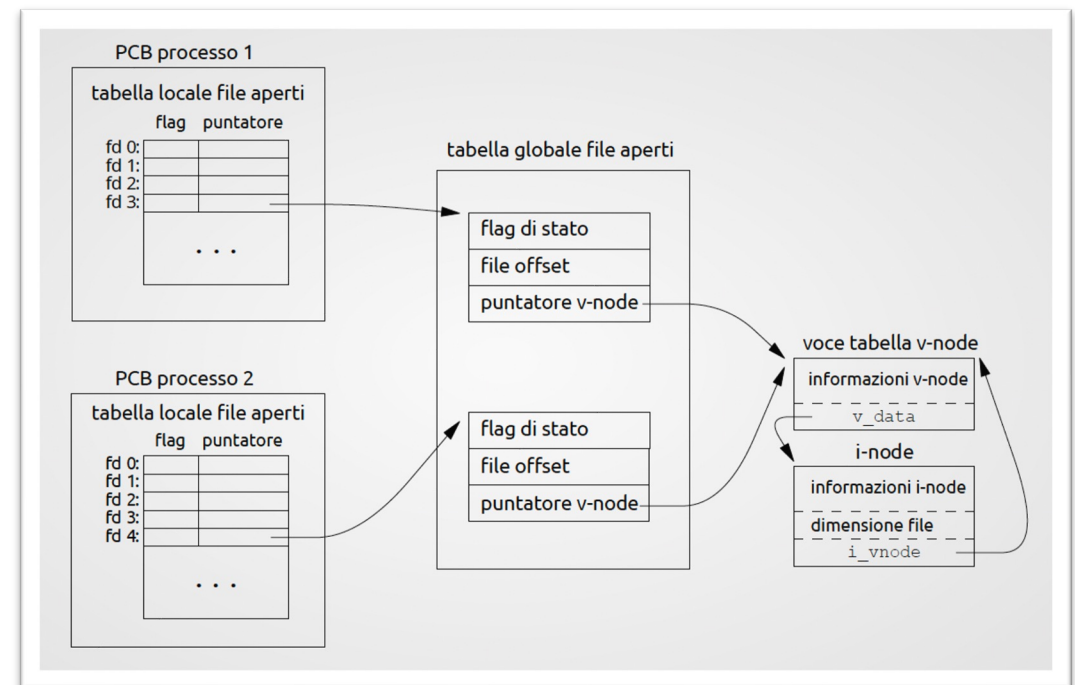
dimens. buffer	CPU utente (s)	CPU kernel (s)	clock time (s)	interazioni
1	20,03	117,50	138,73	516.581.760
2	9,69	58,76	68,60	258.290.880
4	4,60	36,47	41,27	129.145.440
8	2,47	15,44	18,38	64.572.720
16	1,07	7,93	9,38	32.286.360
32	0,56	4,51	8,82	16.143.180
64	0,34	2,72	8,66	8.071.590
128	0,34	1,84	8,69	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

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

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>

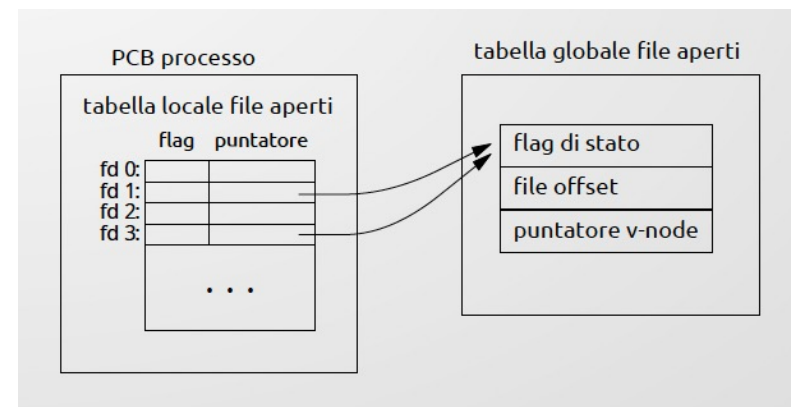
Duplicazione dei Descrittori di File

```
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); 

<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

Letture 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 *ptr, size_t size, size_t nobj, FILE *fp);
```

```
size_t fwrite(const void *ptr, 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

- **stat** (e vantiati) 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

```
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>

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`: dispositivo 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

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 ))...
```

- il campo `st_mode` può essere ispezionato in vari modi: [link](#)
- la maschera `S_IRWXU` (cioè `S_IRUSR | S_IWUSR | S_IXUSR`) rappresenta i permessi di lettura, scrittura e esecuzione per l'utente proprietario (e nessun altro): [link](#)
- i flag che compongono `st_mode` sono definiti in `sys/stat.h` e possono essere usati per controllare i permessi di lettura, scrittura e esecuzione per l'utente proprietario, per il gruppo proprietario e per gli altri utenti. 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 ))...
```
- i timestamp (`time_t`) sono interi che possono essere localizzati al fuso orario (UTC) o al fuso orario locale (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 campi `st_uid` e `st_gid`
- 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 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** 📄

```
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.

```
int closedir(DIR *dp);
```

`<dirent.h>`

• `opendir`
di una

• `readdir`
ritorna
puntatore

• `readdir_r`
stream

• `readdir64`
ritorna
al

• `closedir`
co
no

• `seekdir`
si pos
e

• 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

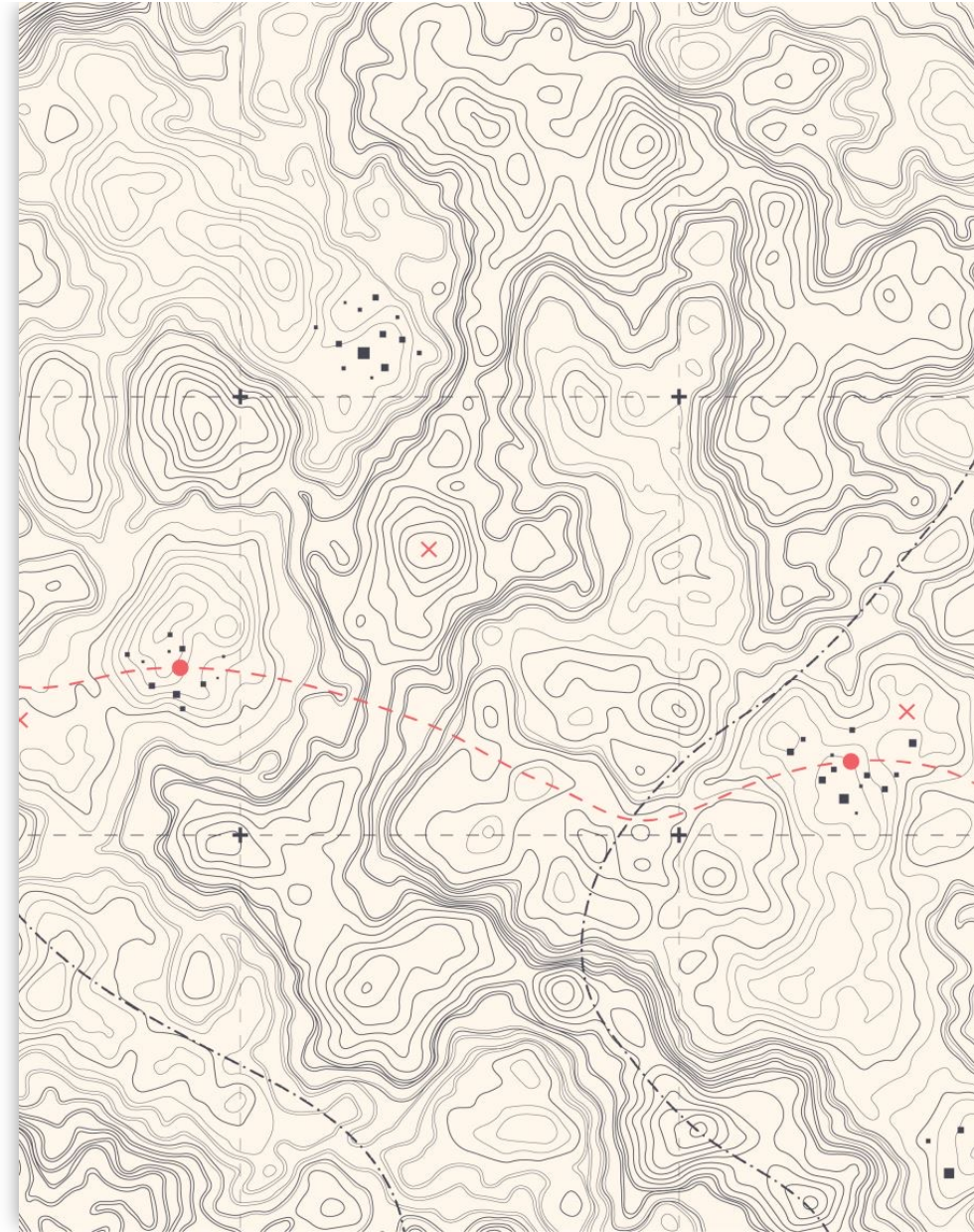
Mappatura dei File

```
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**