



NoiStudiamo

DISPENSA DI LABORATORIO

Sistemi Operativi Laboratorio

Programmazione di sistema in C con le interfacce POSIX:
file, processi, thread e sincronizzazione.

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

I tredici capitoli della dispensa, dagli strumenti di base agli esempi svolti di programmi concorrenti.

1. Strumenti e convenzioni	3
2. I file e i descrittori	7
3. L'I/O bufferizzato con gli stream	12
4. Il file system: attributi, directory e link	15
5. La mappatura dei file in memoria	19
6. I processi	22
7. I thread POSIX	26
8. Mutex e semafori	29
9. Variabili condizione, lock lettori e scrittori, barriere	34
10. Progettare un programma concorrente	39
11. Esempio svolto: dimensione dei file in più directory	42
12. Esempio svolto: vettori in file binari con mmap	48
13. Esempio svolto: duplicazione di file a blocchi	54

Strumenti e convenzioni

Le chiamate di sistema e le funzioni di libreria, la documentazione, gli standard, la compilazione e le convenzioni comuni a tutti i programmi che seguono.

1.1 Chiamate di sistema e funzioni di libreria

Un programma C che lavora con il sistema operativo può usare due tipi di servizi. Le **chiamate di sistema** sono i servizi offerti direttamente dal kernel: il programma esegue un'istruzione di trap, il processore passa in **modalità kernel** e il sistema operativo esegue la richiesta per conto del processo. Le **funzioni di libreria** sono invece normali funzioni contenute nella libreria di sistema ed eseguite in **modalità utente**; molte di esse, per svolgere il proprio compito, invocano a loro volta una o più chiamate di sistema. La `printf`, ad esempio, prepara il testo in un buffer in modalità utente e poi usa la chiamata di sistema `write` per consegnarlo al kernel.

I sottosistemi su cui si lavora sono quattro: la gestione dei **file** e delle **directory** con il relativo I/O, la gestione dei **processi**, la gestione dei **thread** e la **comunicazione e sincronizzazione** fra processi e thread. Il riferimento classico per tutti questi argomenti è *Advanced Programming in the UNIX Environment* di Stevens e Rago; per i thread sono validi anche *Programming With POSIX Threads* di Butenhof e *PThreads Primer* di Lewis e Berg.

1.2 La documentazione: le pagine di manuale

La documentazione ufficiale di ogni comando e di ogni funzione è nelle **pagine di manuale**, consultabili dalla shell con il comando `man` seguito dal nome cercato, oppure in rete su siti come `man7.org`. Su Debian e Ubuntu le pagine dello standard POSIX si installano con il pacchetto `manpages-posix-dev`. Il manuale è diviso in **sezioni** numerate:

SEZIONE	CONTENUTO
1	Comandi utente
2	Chiamate di sistema
3	Funzioni di libreria
8	Comandi di amministrazione

Quando lo stesso nome compare in più sezioni bisogna indicare quella voluta: `man 1 chown` mostra il comando della sezione 1, `man 2 chown` la chiamata di sistema. Ogni pagina riporta il prototipo, gli header da includere, il significato del valore restituito e i possibili codici di errore: è la prima cosa da consultare quando non si ricorda l'uso preciso di una funzione.

1.3 Gli standard: ISO C e POSIX

Per scrivere codice **portabile**, cioè che si possa compilare su sistemi e architetture diversi, ci si attiene agli standard. Lo standard **ISO C** definisce il linguaggio e la sua libreria (ad esempio gli stream di `stdio.h`). Lo standard **IEEE POSIX** (*Portable Operating System Interface*, famiglia 1003.1) definisce le interfacce di programmazione dei sistemi di tipo UNIX: file, directory, processi, segnali e, con le estensioni real time e quelle per i thread, i **POSIX thread** e i semafori. Una parte di POSIX si sovrappone a ISO C.

GNU/Linux supporta POSIX in modo quasi completo, ma attiva per default anche alcune estensioni GNU. Per chiedere al compilatore di esporre esattamente le interfacce POSIX si definisce, prima di qualunque `#include`, la

macro seguente:

```
#define _POSIX_C_SOURCE 200809L
```

macOS offre un supporto quasi completo (mancano però i semafori numerici senza nome e le barriere), mentre Windows offre un supporto parziale, ampliabile con il *Windows Subsystem for Linux* (WSL).

1.4 Compilare un programma

Un programma con un solo sorgente si compila e si collega in un solo passaggio; con più sorgenti si compila ogni file in un file oggetto e poi si collegano gli oggetti. Le librerie si aggiungono con l'opzione `-l` seguita dal nome della libreria (m per la matematica), e lo standard del linguaggio si sceglie con `-std`. Per i programmi con i thread si usa l'opzione `-pthread`, che imposta sia la compilazione sia il collegamento con la libreria dei thread.

COMANDI DI COMPILAZIONE

<code>gcc -Wall -o programma programma.c</code>	<code># un solo sorgente</code>
<code>gcc -c file1.c</code>	<code># compilazione separata</code>
<code>gcc -c file2.c</code>	
<code>gcc -o programma file1.o file2.o</code>	<code># collegamento</code>
<code>gcc -Wall -pthread -o programma programma.c</code>	<code># programma con thread</code>
<code>gcc -std=c99 -lm -o calcolo calcolo.c</code>	<code># standard C99 e libreria matematica</code>
<code>gcc -static -o programma programma.c</code>	<code># collegamento statico</code>

L'opzione `-Wall` attiva gli avvisi più utili e conviene usarla sempre: molti errori di concorrenza nascono da variabili non inizializzate o da tipi sbagliati che il compilatore sa già segnalare. Per progetti con molti sorgenti si usa `make` con un *makefile*, che descrive regole del tipo «obiettivo: dipendenze» seguite dal comando che produce l'obiettivo; `make` ricompila soltanto ciò che è cambiato.

UNA NOTA SUI LISTATI

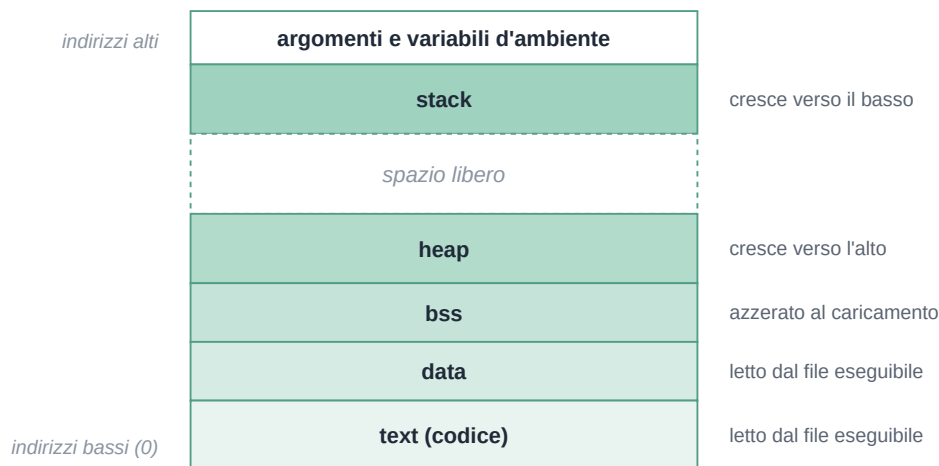
Nei listati di questa dispensa il segno meno e il trattino sono resi con il segno meno tipografico. Quando si ricopia il codice va digitato il normale segno meno della tastiera, ad esempio nell'operatore freccia `p->campo` o nelle opzioni del compilatore.

1.5 Argomenti dalla riga di comando

Un programma riceve gli argomenti della riga di comando attraverso i due parametri del `main`: `argc` è il numero di argomenti, compreso il nome del programma, e `argv` è un vettore di stringhe, terminato da un puntatore nullo, in cui `argv[0]` è il nome con cui il programma è stato invocato e gli elementi da `argv[1]` a `argv[argc - 1]` sono gli argomenti veri e propri. La prima cosa che fa un programma è controllare che il numero di argomenti sia quello atteso e, in caso contrario, stampare un messaggio d'uso sullo standard error e terminare con un codice di errore. Gli argomenti numerici si convertono con `atoi` o, controllando gli errori, con `strtol`.

1.6 Lo spazio di indirizzamento di un processo

Lo spazio degli indirizzi virtuali di un processo è organizzato in **segmenti**. Dagli indirizzi bassi verso quelli alti si trovano: il **codice** (*text*), mappato dal file eseguibile; i **dati inizializzati** (*data*), come una variabile globale dichiarata con un valore iniziale, letti anch'essi dal file; i **dati non inizializzati** (*bss*), come un vettore globale senza valore iniziale, che non occupano spazio nel file e vengono azzerati al caricamento; lo **heap**, da cui `malloc` preleva la memoria dinamica e che cresce verso l'alto; lo **stack**, con le variabili locali e i record di attivazione delle funzioni, che cresce verso il basso. In cima si trovano gli argomenti della riga di comando e le variabili d'ambiente.



I segmenti dello spazio di indirizzamento di un processo, dagli indirizzi bassi (in basso) agli indirizzi alti (in alto).

Il comando `size` mostra le dimensioni dei segmenti di un eseguibile. Un programma collegato **staticamente** contiene al proprio interno il codice delle librerie che usa ed è quindi molto più grande della versione collegata dinamicamente, che carica le librerie condivise al momento dell'esecuzione.

1.7 La gestione degli errori

Quasi tutte le chiamate di sistema segnalano un errore in due modi: restituiscono un **valore anomalo**, in genere meno 1 (oppure NULL per le funzioni che restituiscono un puntatore), e scrivono un codice che descrive l'errore nella variabile globale `errno`, dichiarata in `errno.h`. I codici hanno nomi simbolici, come `ENOENT` (file o directory inesistente), `EACCES` (permesso negato) o `EINTR` (chiamata interrotta da un segnale). Due funzioni trasformano il codice in un messaggio leggibile: `perror`, che stampa sullo standard error il testo passato seguito dal messaggio corrispondente al valore attuale di `errno`, e `strerror`, che restituisce il messaggio relativo a un codice dato.

DEFINIZIONE

Il valore di `errno` ha significato **solo subito dopo una chiamata fallita**: una chiamata che va a buon fine non lo azzerava. Va quindi controllato il valore restituito dalla funzione e, solo se indica un errore, si legge `errno`. Fanno eccezione le funzioni dei **POSIX thread**, che non usano `errno` ma restituiscono direttamente il codice di errore (0 in caso di successo).

Alcuni errori sono **fatali** e conviene terminare il programma; altri, come `EINTR` o `EAGAIN`, sono **temporanei** e la chiamata si può ripetere. Nei programmi di questa dispensa ogni errore delle chiamate di sistema viene gestito con una piccola funzione che chiama `perror` e termina:

```
void errore(const char *msg) {
    perror(msg);          /* messaggio con la descrizione di errno */
    exit(EXIT_FAILURE);
}
```

1.8 La terminazione di un processo

Un processo termina quando chiama `exit`, quando il `main` ritorna (il ritorno dal `main` equivale a chiamare `exit` con il valore restituito) oppure quando termina il suo ultimo thread. L'argomento di `exit` è lo **stato di uscita**: al processo padre arrivano soltanto gli 8 bit meno significativi, cioè `status & 0xFF`. Per convenzione 0 indica il successo e un valore positivo un errore; le costanti `EXIT_SUCCESS` ed `EXIT_FAILURE` di `stdlib.h` rendono il codice più portabile.

La `exit` termina il processo in modo **ordinato**: chiama prima, in ordine inverso rispetto alla registrazione, le funzioni registrate con `atexit`, che servono a eseguire operazioni di chiusura (ad esempio rimuovere un file temporaneo), poi svuota i buffer degli stream ancora pendenti e chiude gli stream. Per terminare **senza** queste operazioni esiste la chiamata di sistema `_exit`, usata ad esempio da un processo figlio che non deve svuotare una seconda volta i buffer ereditati dal padre.

FUNZIONI DI CHIUSURA CON ATEXIT

```
#include <stdio.h>
#include <stdlib.h>

void saluta(void) {
    printf("chiusura: ultima funzione registrata, eseguita per prima\n");
}

void pulisci(void) {
    printf("chiusura: prima funzione registrata, eseguita per ultima\n");
}

int main(void) {
    atexit(pulisci);           /* registrazione delle funzioni di chiusura */
    atexit(saluta);
    printf("lavoro del programma...\n");
    exit(EXIT_SUCCESS);       /* chiama saluta, poi pulisci, poi svuota i buffer */
}
```

I file e i descrittori

L'I/O di basso livello: aprire, leggere, scrivere e posizionarsi in un file attraverso i descrittori, e le strutture con cui il kernel gestisce i file aperti.

2.1 I descrittori di file

Quando un processo apre un file, il kernel gli restituisce un **descrittore di file** (*file descriptor*): un intero non negativo con cui il processo si riferirà al file in tutte le operazioni successive. Ogni processo parte con tre descrittori già aperti, definiti in `unistd.h`:

DESCRITTORE	COSTANTE	CANALE
0	STDIN_FILENO	Standard input
1	STDOUT_FILENO	Standard output
2	STDERR_FILENO	Standard error

Quando un programma viene lanciato da un terminale, i tre canali sono di solito collegati al terminale stesso, ma non è detto: la shell può redirigerli su file (`./programma < ingresso.txt > uscita.txt`) o collegarli ad altri programmi con le pipe (`cat dati.txt | ./programma | sort`). Il programma non se ne accorge, perché continua a leggere dal descrittore 0 e a scrivere sul descrittore 1.

2.2 Aprire, creare e chiudere un file

PROTOTIPI (FCNTL.H, UNISTD.H)

```
int open(const char *path, int oflag, ... /* mode_t mode */);
int creat(const char *path, mode_t mode);
int close(int fd);
```

La `open` apre il file indicato da `path` e restituisce il descrittore, che è sempre il **più basso numero libero**; in caso di errore restituisce meno 1. Il parametro `oflag` combina con l'operatore `|` alcuni flag: esattamente uno fra i tre modi di accesso, più eventuali opzioni.

FLAG	SIGNIFICATO
O_RDONLY	Apertura in sola lettura
O_WRONLY	Apertura in sola scrittura
O_RDWR	Apertura in lettura e scrittura
O_CREAT	Crea il file se non esiste, con i permessi indicati dal terzo argomento <code>mode</code>
O_EXCL	Insieme a <code>O_CREAT</code> , fa fallire la chiamata se il file esiste già
O_TRUNC	Se il file esiste, lo tronca a lunghezza zero
O_APPEND	Ogni scrittura avviene in fondo al file

La combinazione `O_CREAT | O_EXCL` è utile quando si vuole creare un file soltanto se non esiste: la verifica e la creazione avvengono in un'unica operazione atomica, senza il rischio che un altro processo crei il file fra le due. La `creat` è una forma abbreviata che equivale a `open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)`. La `close` chiude il descrittore e lo rende di nuovo disponibile; alla terminazione del processo tutti i descrittori ancora aperti vengono chiusi automaticamente.

2.3 I permessi e la maschera di creazione

Ogni oggetto del file system ha tre gruppi di permessi, per l'**utente proprietario**, per il **gruppo proprietario** e per **tutti gli altri**, e per ognuno tre diritti: **lettura** (r), **scrittura** (w) ed **esecuzione** (x). Per le directory il diritto di esecuzione indica il permesso di **attraversarle**, cioè di accedere ai file che contengono. Il tipo `mode_t` codifica la maschera dei permessi, che si compone con le costanti di `sys/stat.h`:

	LETTURA	SCRITTURA	ESECUZIONE
Utente	S_IRUSR	S_IWUSR	S_IXUSR
Gruppo	S_IRGRP	S_IWGRP	S_IXGRP
Altri	S_IROTH	S_IWOTH	S_IXOTH

È molto diffuso anche l'uso diretto della notazione **ottale**, in cui ogni cifra rappresenta un gruppo (r vale 4, w vale 2, x vale 1): 0644 equivale a `S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH`, cioè lettura e scrittura per il proprietario e sola lettura per gli altri.

Quando un file o una directory viene creato, ai permessi richiesti viene applicata la **maschera di creazione** del processo, che toglie per sicurezza alcuni permessi: i permessi effettivi sono `mode & ~maschera`. Con la maschera tipica 022 un file creato con 0666 ottiene 0644. Ogni processo ha la propria maschera, che eredita dal padre e può cambiare con la funzione `umask`; anche la shell ha la sua, modificabile con l'omonimo comando.

2.4 Leggere e scrivere

PROTOTIPI (UNISTD.H)

```
ssize_t read(int fd, void *buf, size_t nbytes);
ssize_t write(int fd, const void *buf, size_t nbytes);
```

La `read` legge al massimo `nbytes` byte dal descrittore e li copia nel buffer `buf`. Restituisce il numero di byte effettivamente letti, **0 alla fine del file** e meno 1 in caso di errore. Il numero di byte letti può essere **minore** di quello richiesto, e non è un errore: succede quando il file sta finendo, quando si legge da un terminale (che restituisce una riga alla volta), da una pipe o da un socket, o quando la lettura viene interrotta da un segnale. Per questo una lettura completa si fa sempre con un **ciclo**, che termina quando la `read` restituisce 0. La `write` scrive `nbytes` byte presi da `buf` e restituisce il numero di byte scritti, oppure meno 1; con i file regolari, se non ci sono errori, scrive tutti i byte richiesti.

Il programma seguente copia un file in un altro: è lo schema di base di ogni elaborazione con i descrittori.

COPIA.C

```
/* copia.c: copia un file in un altro usando read e write */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

COPIA.C (CONTINUA)

```
#define DIM_BUFFER 4096

int main(int argc, char *argv[]) {
    int fd_in, fd_out;
    ssize_t letti;
    char buffer[DIM_BUFFER];

    if (argc != 3) {
        fprintf(stderr, "uso: %s <sorgente> <destinazione>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* apertura del file sorgente in sola lettura */
    if ((fd_in = open(argv[1], O_RDONLY)) < 0) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    /* creazione (o troncamento) del file destinazione con permessi rw r r */
    if ((fd_out = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644)) < 0) {
        perror(argv[2]);
        exit(EXIT_FAILURE);
    }

    /* ciclo di copia: read restituisce 0 alla fine del file */
    while ((letti = read(fd_in, buffer, DIM_BUFFER)) > 0) {
        if (write(fd_out, buffer, letti) != letti) {
            perror("write");
            exit(EXIT_FAILURE);
        }
    }
    if (letti < 0) {
        /* errore durante la lettura */
        perror("read");
        exit(EXIT_FAILURE);
    }

    close(fd_in);
    close(fd_out);
    exit(EXIT_SUCCESS);
}
```

2.5 Il posizionamento

PROTOTIPO (UNISTD.H)

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

Ogni file aperto ha una **posizione corrente** (*file offset*), che vale 0 all'apertura e avanza a ogni lettura o scrittura del numero di byte trasferiti; è ciò che rende sequenziale l'accesso. La `lseek` sposta la posizione di `offset` byte rispetto a un riferimento indicato da `whence`: `SEEK_SET` (inizio del file), `SEEK_CUR` (posizione attuale) o `SEEK_END` (fine del file); negli ultimi due casi l'offset può essere negativo. Restituisce la nuova posizione misurata dall'inizio del file, oppure meno 1. Non esegue nessun I/O: modifica soltanto un valore nel kernel. Per conoscere la posizione attuale si usa `lseek(fd, 0, SEEK_CUR)`.

La `lseek` vale solo per gli oggetti che ammettono l'accesso diretto: su una pipe, una FIFO o un socket fallisce con l'errore `ESPIPE`. Se ci si posiziona **oltre la fine** del file e si scrive, il file si allunga e la zona intermedia, mai scritta, diventa un **buco** (*hole*): si legge come una sequenza di zeri e, sulla maggior parte dei file system, non occupa blocchi sul disco.

2.6 L'efficienza dell'I/O

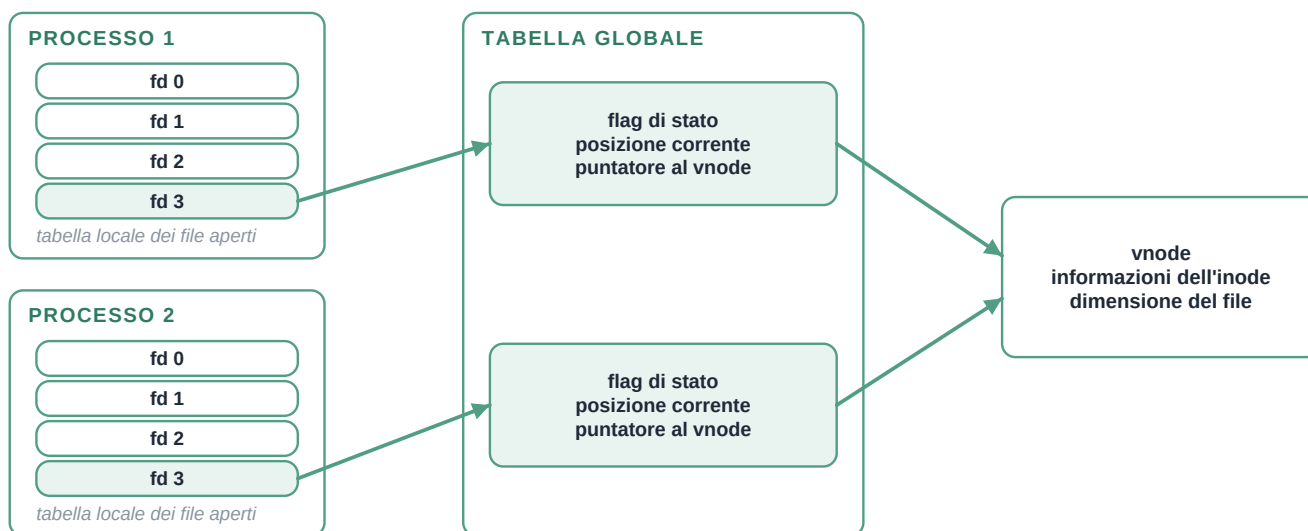
Ogni `read` e ogni `write` è una chiamata di sistema, con il relativo passaggio in modalità kernel, quindi la **dimensione del buffer** incide molto sulle prestazioni. La tabella riporta i tempi, misurati su Linux, per copiare lo stesso file grande con buffer di dimensioni diverse.

BUFFER (BYTE)	CPU UTENTE (S)	CPU KERNEL (S)	TEMPO TOTALE (S)
1	20,03	117,50	138,73
16	1,07	7,93	9,38
256	0,15	1,30	8,69
4096	0,03	0,58	8,62
65536	0,01	0,56	9,12

Con un byte alla volta il tempo è dominato dal costo delle chiamate di sistema. Già con poche centinaia di byte il tempo totale si stabilizza, perché il collo di bottiglia diventa il disco; il tempo di CPU nel kernel smette di scendere intorno alla dimensione del blocco del file system, tipicamente **4096 byte**, che è quindi una buona scelta per il buffer.

2.7 Le strutture dati del kernel per i file aperti

Il kernel gestisce i file aperti con tre livelli di strutture. Ogni processo ha una **tabella locale dei file aperti**, collegata al suo PCB, in cui il descrittore è l'indice: ogni voce contiene alcuni flag del descrittore e un puntatore a una voce della **tabella globale dei file aperti**, unica nel sistema. Ogni voce della tabella globale corrisponde a un'apertura e contiene i **flag di stato** (modo di accesso, `O_APPEND`), la **posizione corrente** e un puntatore al **vnode** del file, la struttura che descrive il file vero e proprio e contiene, fra l'altro, le informazioni dell'inode come la dimensione.



Le tre strutture del kernel: tabella locale del processo, tabella globale dei file aperti e vnode del file.

Questa organizzazione spiega diversi comportamenti. Se due processi aprono lo **stesso file**, ottengono due voci distinte nella tabella globale, ciascuna con la propria posizione, che puntano allo stesso vnode. Se invece due descrittori puntano alla **stessa voce** globale, come accade dopo una dup o fra padre e figlio dopo una fork, condividono la posizione corrente: una lettura con uno dei due sposta la posizione anche per l'altro. I **thread** di un processo, infine, condividono l'intera tabella locale.

2.8 Scritture concorrenti: O_APPEND, pread e pwrite

Quando più processi accodano dati allo stesso file (il caso tipico è un file di log) la sequenza «lseek alla fine, poi write» non è sicura: fra le due chiamate un altro processo può scrivere, e la posizione calcolata non è più la fine del file. Con il flag O_APPEND è il kernel a posizionarsi alla fine **in modo atomico** prima di ogni scrittura.

Un problema analogo si presenta con i thread, che condividono le posizioni correnti: due thread che alternano lseek e read sullo stesso descrittore possono intrecciarsi e leggere dalla posizione sbagliata. Le chiamate pread e pwrite leggono e scrivono **a una posizione indicata come argomento**, in un'unica operazione atomica e **senza modificare** la posizione corrente:

PROTOTIPI (UNISTD.H)

```
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);
```

2.9 Duplicare i descrittori

PROTOTIPI (UNISTD.H)

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

La dup crea un nuovo descrittore, il più basso libero, che punta alla stessa voce della tabella globale di fd. La dup2 fa lo stesso ma usa come nuovo descrittore proprio fd2, chiudendolo prima se era aperto; chiusura e duplicazione avvengono in modo atomico. Il loro uso tipico è la **redirezione** dei canali standard, ed è così che la shell realizza > e <:

REDIREZIONE DELLO STANDARD OUTPUT SU UN FILE

```
int fd = open("uscita.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd < 0)
    errore("open");
dup2(fd, STDOUT_FILENO); /* da qui in poi il descrittore 1 punta al file */
close(fd);                /* il descrittore originale non serve più */
printf("questa riga finisce nel file\n");
```

2.10 La cache del disco

Il sistema operativo usa la RAM libera come **cache del disco**, anche in scrittura: una write che termina con successo ha copiato i dati nella cache, ma la scrittura fisica sul disco può avvenire diversi secondi dopo (su Linux al massimo circa 30). Di norma è un vantaggio, ma in caso di crash i dati non ancora scritti vanno persi. Quando serve la certezza che i dati siano sul disco si può aprire il file con il flag O_SYNC, che rende **sincrona** ogni scrittura, oppure forzare la scrittura con fsync(fd), che ritorna solo dopo aver scritto su disco i dati del file, o con sync(), che chiede la scrittura di tutti i buffer del sistema (esiste anche il comando sync della shell).

L'I/O bufferizzato con gli stream

La libreria standard del C per l'I/O: gli stream, i tipi di buffering e le funzioni per leggere e scrivere per caratteri, per righe e per blocchi.

3.1 Stream e buffering

Lo standard ISO C definisce una libreria di I/O di più alto livello, basata sugli **stream**, il cui scopo principale è **ridurre il numero di chiamate di sistema**. Uno stream è rappresentato da un puntatore di tipo `FILE *`, che contiene il descrittore sottostante e un **buffer** in memoria utente: le letture prelevano i dati dal buffer e chiamano `read` solo quando il buffer è vuoto, le scritture accumulano i dati nel buffer e chiamano `write` solo quando serve. All'avvio sono disponibili tre stream predefiniti, `stdin`, `stdout` e `stderr`, associati ai descrittori 0, 1 e 2.

BUFFERING	QUANDO IL BUFFER VIENE SVUOTATO
Completo (<i>fully buffered</i>)	Quando è pieno; è il modo usato di norma per i file
Per riga (<i>line buffered</i>)	A ogni ritorno a capo; è il modo usato per <code>stdin</code> e <code>stdout</code> collegati a un terminale
Assente (<i>unbuffered</i>)	Subito, a ogni operazione; è il modo usato per <code>stderr</code> , perché i messaggi di errore compaiano immediatamente

Con `fflush(fp)` si forza la scrittura immediata del contenuto del buffer, che passa però alla cache del kernel e non necessariamente al disco. Il buffering spiega un comportamento a prima vista strano: un messaggio stampato con `printf` senza ritorno a capo può non comparire subito, perché resta nel buffer.

3.2 Aprire e chiudere uno stream

PROTOTIPI (STDIO.H)

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

La `fopen` apre il file indicato e crea lo stream; la `fdopen` crea uno stream a partire da un descrittore già aperto, con una modalità coerente con quella del descrittore. Entrambe restituiscono `NULL` in caso di errore. La modalità è una stringa:

MODALITÀ	SIGNIFICATO	FLAG EQUIVALENTI DI OPEN
"r"	Sola lettura	<code>O_RDONLY</code>
"r+"	Lettura e scrittura	<code>O_RDWR</code>
"w"	Scrittura; crea il file o lo tronca	<code>O_WRONLY</code> <code>O_CREAT</code> <code>O_TRUNC</code>
"w+"	Lettura e scrittura; crea o tronca	<code>O_RDWR</code> <code>O_CREAT</code> <code>O_TRUNC</code>
"a"	Scrittura in coda; crea se non esiste	<code>O_WRONLY</code> <code>O_CREAT</code> <code>O_APPEND</code>

La `fclose` svuota il buffer e chiude lo stream insieme al descrittore sottostante.

3.3 Lettura e scrittura per caratteri

La `fgetc` legge un carattere dallo stream e la `fputc` lo scrive; grazie al buffer, leggere un carattere alla volta con queste funzioni è molto più efficiente che farlo con `read`. La `fgetc` restituisce il carattere letto convertito in `int`, oppure la costante `EOF` sia alla fine del file sia in caso di errore: per distinguere i due casi si usano `feof(fp)` e `ferror(fp)`.

DA RICORDARE

Perché il risultato di `fgetc` va memorizzato in un `int` e non in un `char`?

Perché `fgetc` deve poter restituire tutti i 256 valori possibili di un byte **più** il valore speciale `EOF`, che è negativo. Memorizzandolo in un `char` si perde la distinzione: un byte valido potrebbe essere scambiato per la fine del file, oppure la fine del file non verrebbe mai riconosciuta.

3.4 Lettura e scrittura per righe

PROTOTIPI (STDIO.H)

```
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, ...);
```

La `fgets` legge una riga dallo stream e la memorizza come stringa nel buffer `buf` di `n` byte: copia al massimo `n` meno 1 caratteri, compreso il ritorno a capo finale, e aggiunge il terminatore. Restituisce `buf`, oppure `NULL` alla fine del file o in caso di errore. Se la riga è più lunga del buffer viene letta in più chiamate. La `fputs` scrive una stringa sullo stream. La `fprintf` e la `fscanf` sono le versioni della `printf` e della `scanf` che lavorano su uno stream qualsiasi.

NUMERA.C

```
/* numera.c: stampa un file di testo numerandone le righe (stream e fgets) */
#include <stdio.h>
#include <stdlib.h>

#define DIM_RIGA 1024

int main(int argc, char *argv[]) {
    FILE *f;
    char riga[DIM_RIGA];
    int n = 0;

    if (argc != 2) {
        fprintf(stderr, "uso: %s <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* apertura dello stream in sola lettura */
    if ((f = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
}
```

NUMERA.C (CONTINUA)

```
/* fgets legge una riga alla volta (ritorno a capo compreso) */
while (fgets(riga, DIM_RIGA, f) != NULL) {
    n++;
    printf("%4d  %s", n, riga);
}

fclose(f);
exit(EXIT_SUCCESS);
}
```

Per eliminare il ritorno a capo letto da fgets si può usare `riga[strcspn(riga, "\n")] = '\0'`. Per dividere una riga in campi separati da un carattere (ad esempio un file con righe del tipo `data|livello|messaggio`) si usa `strtok`, oppure, nei programmi con più thread, la sua versione rientrante `strtok_r`, che non usa uno stato interno condiviso.

3.5 Lettura e scrittura per blocchi

PROTOTIPI (STDIO.H)

```
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);
```

La `fread` e la `fwrite` trasferiscono `nobj` record di `size` byte ciascuno fra lo stream e la memoria puntata da `ptr`, e restituiscono il numero di record effettivamente trasferiti, che può essere minore di `nobj` alla fine del file o in caso di errore. Sono adatte ai **file binari**, ad esempio per leggere o scrivere vettori di numeri o strutture.

La tabella confronta i tempi necessari per copiare lo stesso file con i diversi metodi.

METODO	CPU UTENTE (S)	CPU KERNEL (S)	TEMPO TOTALE (S)
read e write con buffer ottimale	0,05	0,29	3,18
fgets e fputs	2,27	0,30	3,49
fgetc e fputc	8,16	0,40	10,18
read e write un byte alla volta	134,61	249,94	394,95

Gli stream con un carattere alla volta sono decine di volte più veloci delle chiamate di sistema con un byte alla volta, ma restano più lenti di `read` e `write` con un buffer ben dimensionato, perché aggiungono il costo delle chiamate di funzione e delle copie fra buffer.

3.6 Il posizionamento sugli stream

Le funzioni `fseek` e `fseeko` spostano la posizione corrente dello stream, con gli stessi parametri della `lseek`; `ftell` e `ftello` la restituiscono; `rewind` riporta la posizione all'inizio. Le varianti con la `o` finale usano il tipo `off_t` invece di `long` e sono da preferire, perché `off_t` rappresenta posizioni oltre i 2 GiB anche sui sistemi in cui `long` è di 32 bit.

Il file system: attributi, directory e link

Come si leggono gli attributi di un file, come si creano e si esplorano le directory e come si gestiscono i link fisici e simbolici.

4.1 Gli attributi di un file

PROTOTIPI (SYS/STAT.H)

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

Le tre funzioni riempiono una struttura `struct stat` con gli attributi di un oggetto del file system, indicato con il percorso (`stat`, `lstat`) o con un descrittore aperto (`fstat`). La differenza fra `stat` e `lstat` riguarda i **link simbolici**: `stat` li attraversa e restituisce gli attributi del file a cui puntano, `lstat` restituisce quelli del link stesso. I campi principali sono:

CAMPO	CONTENUTO
<code>st_mode</code>	Tipo di oggetto e permessi
<code>st_ino</code>	Numero dell'inode
<code>st_nlink</code>	Numero di link fisici all'inode
<code>st_uid, st_gid</code>	Identificativi dell'utente e del gruppo proprietari
<code>st_size</code>	Dimensione in byte
<code>st_atime</code>	Ultimo accesso al contenuto
<code>st_mtime</code>	Ultima modifica del contenuto
<code>st_ctime</code>	Ultimo cambiamento dello stato dell'inode (attributi o contenuto)

4.2 Il tipo di file e i permessi

Il campo `st_mode` contiene insieme il tipo dell'oggetto e i permessi. I permessi si isolano con `st_mode & 0777`; il tipo si verifica con alcune macro:

MACRO	VERA SE L'OGGETTO È
<code>S_ISREG(m)</code>	Un file regolare
<code>S_ISDIR(m)</code>	Una directory
<code>S_ISLNK(m)</code>	Un link simbolico (ha senso solo con <code>lstat</code>)
<code>S_ISCHR(m)</code>	Un dispositivo speciale a caratteri
<code>S_ISBLK(m)</code>	Un dispositivo speciale a blocchi

I tempi sono di tipo `time_t`, un numero di secondi: si convertono nell'ora locale con `localtime` e in una stringa leggibile con `asctime`, ad esempio `asctime(localtime(&info.st_mtime))`. Le funzioni `getpwuid` e `getgrgid` ricavano dai campi `st_uid` e `st_gid` il nome dell'utente e del gruppo.

4.3 Creare e attraversare le directory

PROTOTIPI (SYS/STAT.H, UNISTD.H)

```
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);
```

La `mkdir` crea una directory con i permessi `mode` (ai quali si applica la maschera di creazione) e restituisce 0, oppure meno 1; se la directory esiste già fallisce con l'errore `EEXIST`, e questo permette di crearla «solo se non esiste» controllando il codice di errore. La `rmdir` cancella una directory, che deve essere **vuota**. La `chdir` cambia la **directory di lavoro** del processo, rispetto alla quale si risolvono i percorsi relativi, e la `getcwd` la copia nel buffer indicato.

4.4 Leggere il contenuto di una directory

PROTOTIPI (DIRENT.H)

```
DIR *opendir(const char *pathname);
struct dirent *readdir(DIR *dp);
int closedir(DIR *dp);
void rewinddir(DIR *dp);
```

La `opendir` apre uno **stream di directory** e restituisce un puntatore di tipo `DIR *`, oppure `NULL`. Ogni chiamata a `readdir` restituisce la voce successiva, una `struct dirent` che contiene il numero di inode `d_ino` e il **nome** `d_name`; alla fine delle voci restituisce `NULL`. La `rewinddir` riporta la lettura all'inizio (con `telldir` e `seekdir` si può memorizzare e ripristinare una posizione), e la `closedir` chiude lo stream.

DA RICORDARE

Quali sono i due errori più comuni quando si scorre una directory?

Il primo è usare `d_name` come se fosse un percorso: contiene **solo il nome** del file, quindi per usarlo con `stat` o `open` bisogna costruire il percorso completo concatenando il nome della directory, una barra e il nome, ad esempio con `snprintf`. Il secondo è dimenticare che fra le voci compaiono sempre anche `.` (la directory stessa) e `..` (la directory padre), che vanno escluse quando si attraversano le directory in modo ricorsivo.

Il programma seguente elenca i file regolari contenuti direttamente in una directory, con la loro dimensione e il totale: è lo schema su cui si basa la scansione delle directory nei programmi concorrenti dei capitoli finali.

ELENCA.C

```
/* elenca.c: elenca i file regolari di una directory con la loro dimensione */
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

#define MAX_PERCORSO 1024
```

ELENCA.C (CONTINUA)

```
int main(int argc, char *argv[]) {
    DIR *dir;
    struct dirent *voce;
    struct stat info;
    char percorso[MAX_PERCORSO];
    long long totale = 0;

    if (argc != 2) {
        fprintf(stderr, "uso: %s <directory>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((dir = opendir(argv[1])) == NULL) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }

    /* readdir restituisce NULL quando le voci sono finite */
    while ((voce = readdir(dir)) != NULL) {
        /* d_name contiene solo il nome: si costruisce il percorso completo */
        snprintf(percorso, MAX_PERCORSO, "%s/%s", argv[1], voce->d_name);

        /* lstat non segue i link simbolici */
        if (lstat(percorso, &info) < 0) {
            perror(percorso);
            continue;
        }
        if (S_ISREG(info.st_mode)) { /* solo i file regolari */
            printf("%10lld %s\n", (long long)info.st_size, voce->d_name);
            totale += info.st_size;
        }
    }

    closedir(dir);
    printf("totale: %lld byte\n", totale);
    exit(EXIT_SUCCESS);
}
```

4.5 Link fisici e simbolici

PROTOTIPI (UNISTD.H, STDIO.H)

```
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 bufsz);
```

La `link` crea un nuovo **link fisico** (*hard link*) a un file esistente, cioè una nuova voce di directory che punta allo stesso inode, e la `unlink` ne rimuove uno: il file viene davvero cancellato quando il numero di link scende a zero e nessun processo lo tiene aperto. La `remove` usa `unlink` sui file e `rmdir` sulle directory. La `rename` rinomina o sposta file e directory all'interno dello stesso file system. La `symlink` crea un **link simbolico**, un file speciale che contiene il percorso dell'oggetto a cui punta, e la `readlink` legge quel percorso; attenzione, la `readlink` non aggiunge il terminatore di stringa.

4.6 Altre operazioni sui file

La `truncate` e la `ftruncate` impostano la dimensione di un file, indicato con il percorso o con un descrittore: se la nuova dimensione è minore il file viene accorciato, se è maggiore viene allungato con un buco che si legge come zeri. La `chmod` cambia i permessi di un oggetto, la `chown` l'utente e il gruppo proprietari; su Linux, come su molti altri sistemi, solo l'amministratore può cambiare il proprietario di un file.

DA RICORDARE

Come si visita una directory in modo ricorsivo?

Con una funzione che apre la directory con `opendir`, ne scorre le voci con `readdir` e, per ogni voce, costruisce il percorso completo e ne legge gli attributi con `lstat`: se la voce è un file regolare la elabora, se è una directory diversa da `.` e `..` richiama se stessa sul nuovo percorso. Usare `lstat` invece di `stat` evita di seguire i link simbolici, che potrebbero creare cicli infiniti. Ogni livello di ricorsione tiene aperto uno stream di directory, che va chiuso con `closedir` prima di ritornare.

La mappatura dei file in memoria

Come si accede al contenuto di un file come se fosse un vettore in memoria, con `mmap`, `msync` e `munmap`.

5.1 L'idea della mappatura

Invece di leggere un file con `read` copiandone il contenuto in un buffer, si può chiedere al sistema operativo di **mappare** una porzione del file nello spazio di indirizzamento del processo. Il file diventa accessibile attraverso un puntatore, come un normale vettore: leggere un elemento significa leggere il byte corrispondente del file. Il meccanismo sfrutta la **memoria virtuale**: le pagine della zona mappata vengono caricate dal file al primo accesso, con un page fault, e senza copie intermedie fra kernel e buffer dell'utente.

5.2 La chiamata `mmap`

PROTOTIPO (SYS/MMAN.H)

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

La `mmap` mappa `len` byte del file aperto con il descrittore `fd`, a partire dalla posizione `off` (che deve essere un multiplo della dimensione della pagina, quindi in pratica quasi sempre 0). Con `addr` uguale a `NULL` è il sistema operativo a scegliere l'indirizzo, che viene restituito; in caso di errore la funzione restituisce la costante `MAP_FAILED` (non `NULL`). Una lunghezza nulla non è ammessa, quindi i file vuoti vanno trattati a parte.

PARAMETRO	VALORI
<code>prot</code>	Combinazione di <code>PROT_READ</code> , <code>PROT_WRITE</code> e <code>PROT_EXEC</code> ; deve essere compatibile con il modo di apertura del file
<code>MAP_SHARED</code>	Le scritture nella zona mappata modificano il file e sono visibili agli altri processi che lo mappano
<code>MAP_PRIVATE</code>	Le scritture restano private al processo (copy on write) e non modificano il file

Dopo la `mmap` il descrittore può essere chiuso: la mappatura resta valida. La dimensione da mappare si ricava di solito con `fstat`.

5.3 La sincronizzazione e la rimozione della mappatura

PROTOTIPI (SYS/MMAN.H)

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

Con una mappatura condivisa le modifiche fatte in memoria raggiungono il file attraverso la cache del sistema, che le scrive sul disco quando vuole. La `msync` forza la scrittura su disco di una zona mappata, in modo sincrono (`MS_SYNC`, la chiamata ritorna a scrittura avvenuta) o asincrono (`MS_ASYNC`). La `munmap` rimuove la mappatura; alla terminazione del processo tutte le mappature vengono rimosse automaticamente.

5.4 Un esempio: contare un carattere

CONTA_MMAP.C

```
/* conta_mmap.c: conta le occorrenze di un carattere in un file usando mmap */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd;
    struct stat info;
    char *mappa;
    long conteggio = 0;

    if (argc != 3) {
        fprintf(stderr, "uso: %s <carattere> <file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((fd = open(argv[2], O_RDONLY)) < 0) {
        perror(argv[2]);
        exit(EXIT_FAILURE);
    }

    /* la dimensione del file serve per sapere quanto mappare */
    if (fstat(fd, &info) < 0) {
        perror("fstat");
        exit(EXIT_FAILURE);
    }
    if (info.st_size == 0) {          /* mmap non accetta una lunghezza nulla */
        printf("il file è vuoto\n");
        exit(EXIT_SUCCESS);
    }

    /* mappatura in sola lettura dell'intero file */
    mappa = mmap(NULL, info.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (mappa == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fd);                      /* la mappatura resta valida */

    /* il contenuto si legge come un normale vettore di caratteri */
    for (off_t i = 0; i < info.st_size; i++)
        if (mappa[i] == argv[1][0])
            conteggio++;

    munmap(mappa, info.st_size);
    printf("'%'c' compare %ld volte\n", argv[1][0], conteggio);
    exit(EXIT_SUCCESS);
}
```

5.5 Leggere record binari con mmap

La mappatura è particolarmente comoda per i **file binari** formati da record di dimensione fissa. Se un file contiene vettori di M byte senza segno, si mappa il file come `unsigned char *` e il record numero k occupa i byte da $k * M$ a $k * M + M$ meno 1; il numero dei record è la dimensione del file divisa per M . Se i record sono strutture o interi di più byte, si può convertire il puntatore restituito da `mmap` in un puntatore al tipo corretto, ad esempio `int *`, e scorrere il file come un vettore di quel tipo, a condizione che il file sia stato scritto con la stessa rappresentazione (dimensione degli interi e ordine dei byte) della macchina che lo legge. Il capitolo 12 usa questa tecnica in un programma completo.

DA RICORDARE

Quando conviene la mappatura in memoria rispetto alla lettura con `read`?

La mappatura evita la copia dei dati dalla cache del kernel al buffer del processo e permette di accedere al file **in qualunque ordine** con la sola aritmetica dei puntatori, senza `lseek`: conviene quindi per file letti più volte, ad accesso diretto o da elaborare come vettori. Per una semplice lettura sequenziale, una sola volta dall'inizio alla fine, `read` con un buffer ben dimensionato ha prestazioni simili ed è più semplice; la mappatura inoltre non si applica a pipe, socket e terminali, e richiede attenzione con i file vuoti e con i file che cambiano dimensione mentre sono mappati.

I processi

Come si creano i processi, come il padre attende i figli, come si esegue un altro programma e come si inviano i segnali.

6.1 Identificativi e creazione

PROTOTIPI (UNISTD.H)

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

Ogni processo è identificato da un **PID**; la `getpid` restituisce il PID del processo chiamante e la `getppid` quello del padre. La `fork` **duplica** il processo chiamante: il figlio è una copia del padre, con una copia dello spazio di indirizzamento (realizzata con il copy on write), e riprende l'esecuzione dallo stesso punto, cioè dal ritorno della `fork`. La `fork` è una funzione che **ritorna due volte**, con valori diversi: nel **padre** restituisce il PID del figlio appena creato (oppure meno 1 se la creazione fallisce), nel **figlio** restituisce 0. È questo valore a permettere ai due processi di prendere strade diverse.

SCHEMA TIPICO

```
pid_t pid = fork();
if (pid < 0)
    errore("fork");
if (pid == 0) {
    /* codice eseguito solo dal figlio */
    exit(EXIT_SUCCESS);          /* il figlio termina qui */
}
/* codice eseguito solo dal padre */
```

Se il padre termina prima del figlio, il figlio rimasto **orfano** viene adottato da un altro processo del sistema (tradizionalmente `init`, con PID 1), che ne raccoglierà la terminazione.

6.2 Fork, file aperti e buffer degli stream

Il figlio eredita una copia della tabella locale dei file aperti del padre, le cui voci puntano **alle stesse voci della tabella globale**: padre e figlio condividono quindi i flag di stato e la **posizione corrente** di ogni file aperto prima della `fork`. Se entrambi scrivono sullo stesso file, le scritture si accodano invece di sovrasciversi.

Il figlio eredita anche una copia dei **buffer degli stream** in memoria utente. Se il padre ha stampato con `printf` un testo ancora nel buffer (ad esempio perché non termina con un ritorno a capo, o perché lo standard output è rediretto su un file e quindi completamente bufferizzato), quel testo si trova in entrambi i buffer e verrà **stampato due volte**. Per evitarlo si chiama `fflush(stdout)` prima della `fork`.

6.3 Attendere i figli

PROTOTIPI (SYS/WAIT.H)

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

La `wait` blocca il padre fino alla terminazione di un figlio qualsiasi, la `waitpid` fino alla terminazione del figlio con il PID indicato (con `options` a 0). Entrambe restituiscono il PID del figlio terminato, oppure meno 1 (ad esempio se non ci sono figli da attendere). Se `statloc` non è NULL, nell'intero puntato viene scritto lo **stato di terminazione**, da esaminare con alcune macro: `WIFEXITED(stato)` è vera se il figlio è terminato normalmente, e in quel caso `WEXITSTATUS(stato)` ne restituisce lo stato di uscita; `WIFSIGNALED(stato)` è vera se il figlio è stato terminato da un segnale.

DEFINIZIONE

Un processo che termina resta nello stato di **zombie** (*defunct*) finché il padre non ne raccoglie lo stato di terminazione con `wait` o `waitpid`: il kernel conserva le sue informazioni minime proprio perché il padre possa chiederle. Un padre che crea molti figli senza mai attenderli accumula processi zombie.

6.4 Eseguire un programma

PROTOTIPI (UNISTD.H)

```
int execl(const char *pathname, const char *arg0, ... /* (char *)0 */);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */);
int execvp(const char *filename, char *const argv[]);
```

Le funzioni della famiglia `exec` **sostituiscono il programma** in esecuzione nel processo chiamante con un nuovo programma: il processo resta lo stesso (stesso PID, stessi file aperti), ma codice, dati, heap e stack vengono rimpiazzati. Per questo, se hanno successo, **non ritornano**; se ritornano, si è verificato un errore. Le varianti si distinguono per due lettere:

LETTERA	SIGNIFICATO
l	Gli argomenti del nuovo programma sono passati come lista di parametri, terminata dalla sentinella <code>(char *)0</code>
v	Gli argomenti sono passati come vettore di stringhe, il cui ultimo elemento è NULL
p	Se il nome non contiene una barra, l'eseguibile viene cercato nelle directory elencate nella variabile d'ambiente PATH

Per convenzione il primo argomento passato al nuovo programma è il suo stesso nome. La combinazione di `fork`, `exec` e `waitpid` è il modo in cui la shell esegue i comandi: crea un figlio, il figlio esegue il comando con `exec` e il padre attende la sua terminazione.

ESEGUI.C

```
/* esegui.c: esegue un comando in un processo figlio e ne riporta l'exit code */
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
```

ESEGUI.C (CONTINUA)

```
int main(int argc, char *argv[]) {
    pid_t pid;
    int stato;

    if (argc < 2) {
        fprintf(stderr, "uso: %s <comando> [argomenti]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((pid = fork()) < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        /* figlio: argv + 1 è il vettore {comando, argomenti..., NULL} */
        execvp(argv[1], argv + 1);
        perror("execvp");          /* si arriva qui solo se exec fallisce */
        exit(EXIT_FAILURE);
    }

    /* padre: attende proprio quel figlio */
    if (waitpid(pid, &stato, 0) < 0) {
        perror("waitpid");
        exit(EXIT_FAILURE);
    }
    if (WIFEXITED(stato))
        printf("il figlio %d è terminato con exit code %d\n", (int)pid,
              WEXITSTATUS(stato));
    exit(EXIT_SUCCESS);
}
```

6.5 Script interpretati e la funzione system

Sui sistemi UNIX anche un file di testo può essere eseguito, se ha il permesso di esecuzione e se la sua prima riga indica l'interprete con la convenzione `#!`, ad esempio `#!/usr/bin/bash` o `#!/usr/bin/python3`: è il kernel stesso a riconoscere la riga e ad avviare l'interprete passandogli il file. Per eseguire un comando completo in un sottoprocesso si può usare anche `system("comando argomenti")`, che con `fork`, `exec` e `waitpid` avvia una shell a cui passa la stringa, e restituisce lo stato di terminazione della shell.

6.6 I segnali

I **segnali** sono notifiche asincrone con cui il sistema informa un processo di un evento. Per ogni segnale il processo può lasciare la reazione predefinita, **ignorarlo** oppure eseguire una **funzione di gestione**; due segnali, `SIGKILL` e `SIGSTOP`, non si possono né ignorare né gestire.

SEGNALE	EVENTO	REAZIONE PREDEFINITA
SIGHUP	Perdita del terminale	Terminazione, gestibile
SIGINT	Interruzione dal terminale (Ctrl+C)	Terminazione, gestibile
SIGTERM	Richiesta di terminazione	Terminazione, gestibile
SIGKILL	Terminazione forzata	Terminazione, inevitabile
SIGSEGV, SIGILL	Accesso non valido alla memoria, istruzione illegale	Terminazione
SIGCHLD	Terminazione di un figlio	Ignorato

PROTOTIPI (SIGNAL.H)

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

Nonostante il nome, la `kill` invia un segnale **qualsiasi** al processo indicato; il processo destinatario deve appartenere allo stesso utente, a meno di avere i privilegi di amministratore. La `raise` invia un segnale al processo chiamante.

I thread POSIX

La libreria dei POSIX thread: creare i thread, passare loro i dati, attenderne la terminazione e riconoscere le corse critiche.

7.1 Thread e processi

I **POSIX thread** (o *pthread*) sono l'interfaccia standard per creare e gestire i thread sui sistemi compatibili con POSIX. Rispetto ai processi creati con `fork`, i thread di un processo condividono lo spazio di indirizzamento (codice, variabili globali, heap), i file aperti e la directory di lavoro; ognuno ha però il proprio **stack**, i propri registri e il proprio identificativo. Crearli e alternarli costa molto meno che con i processi, e scambiarsi dati è immediato, perché basta un puntatore a una struttura comune: proprio per questo l'accesso ai dati condivisi va **sincronizzato**.

7.2 Identificativi e compilazione

Un thread è identificato da un valore di tipo `pthread_t`, univoco solo all'interno del processo; la sua rappresentazione dipende dalla piattaforma (un intero, un puntatore o una struttura), quindi due identificativi non si confrontano con `==` ma con `pthread_equal`. La `pthread_self` restituisce l'identificativo del thread chiamante. Oltre a includere `pthread.h`, i programmi con i thread vanno compilati con l'opzione `-pthread`.

PROTOTIPI (PTHREAD.H)

```
pthread_t pthread_self(void);
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

7.3 Creare un thread

PROTOTIPO (PTHREAD.H)

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

La `pthread_create` crea un nuovo thread che esegue la funzione `thread_func` con l'argomento `thread_arg`; l'identificativo del nuovo thread viene scritto in `*tidp`. Gli attributi `attr` permettono di personalizzare il thread e di norma valgono NULL. La funzione del thread ha sempre lo stesso prototipo: riceve un `void *` e restituisce un `void *`. La `pthread_create`, come tutte le funzioni dei pthread, restituisce **0 in caso di successo** e un **codice di errore positivo** altrimenti, senza usare `errno`: il messaggio si ottiene quindi con `strerror(err)` e non con `perror`.

7.4 Terminare e attendere un thread

PROTOTIPI (PTHREAD.H)

```
void pthread_exit(void *rval_ptr);
int pthread_join(pthread_t thread, void **rval_ptr);
```

Un thread termina quando la sua funzione ritorna o quando chiama `pthread_exit`, che equivale a un `return` con lo stesso valore. Se un thread qualsiasi chiama `exit`, termina **l'intero processo**, con tutti i suoi thread; anche il ritorno dal `main` chiama `exit`, quindi il thread principale non deve terminare prima che gli altri abbiano finito. Il

processo resta invece in vita finché c'è almeno un thread attivo, se il thread principale termina con `pthread_exit`.

La `pthread_join` blocca il chiamante fino alla terminazione del thread indicato, in modo simile alla `waitpid` per i processi; se `rval_ptr` non è `NULL`, vi deposita il valore restituito dal thread. Per questo è importante conservare gli identificativi ottenuti da `pthread_create`.

7.5 Passare i dati ai thread

La funzione di un thread riceve un solo argomento, un puntatore generico. Per passare più dati si raccolgono in una **struttura**, se ne passa l'indirizzo e la funzione lo converte nel tipo corretto. Una buona organizzazione, usata in tutti i programmi di questa dispensa, prevede per ogni thread una struttura con i **dati privati** del thread (identificativo, numero d'ordine, compito assegnato) e un **puntatore a una struttura comune** che contiene i dati condivisi e gli strumenti di sincronizzazione. Le strutture vengono create nel `main` e nessun dato è dichiarato come variabile globale.

DA RICORDARE

Perché non si passa al thread l'indirizzo del contatore del ciclo che lo crea?

Perché tutti i thread riceverebbero **lo stesso indirizzo**, e il valore che vi leggono dipende da quando lo leggono: nel frattempo il ciclo può averlo già incrementato, e più thread potrebbero vedere lo stesso numero. Ogni thread deve ricevere l'indirizzo di dati **propri**, ad esempio l'elemento `i` di un vettore di strutture, che resti valido per tutta la durata del thread: non va quindi passato l'indirizzo di una variabile locale di una funzione che termina prima del thread.

Il programma seguente divide la somma di un vettore fra più thread: ogni thread riceve nella propria struttura la porzione da elaborare e vi lascia il risultato, che il `main` raccoglie dopo la `pthread_join`.

SOMMA_THREAD.C

```
/* somma_thread.c: N thread sommano ciascuno una parte di un vettore */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N_THREAD 4
#define N_ELEMENTI 1000

/* dati privati di ogni thread, passati come unico argomento */
typedef struct {
    pthread_t tid;
    int numero;          /* numero del thread (1, 2, ...) */
    int *vettore;        /* vettore condiviso (solo in lettura) */
    int inizio, fine;    /* porzione assegnata: [inizio, fine) */
    long somma;          /* risultato parziale */
} dati_thread;

void *somma_parziale(void *arg) {
    dati_thread *d = arg;
    d->somma = 0;
    for (int i = d->inizio; i < d->fine; i++)
        d->somma += d->vettore[i];
    printf("[T%d] somma parziale: %ld\n", d->numero, d->somma);
    return NULL;
}
```

SOMMA_THREAD.C (CONTINUA)

```
int main(void) {
    int vettore[N_ELEMENTI];
    dati_thread dati[N_THREAD];
    long totale = 0;
    int err;

    for (int i = 0; i < N_ELEMENTI; i++)
        vettore[i] = i + 1;

    /* creazione dei thread, ognuno con la propria porzione */
    for (int i = 0; i < N_THREAD; i++) {
        dati[i].numero = i + 1;
        dati[i].vettore = vettore;
        dati[i].inizio = i * N_ELEMENTI / N_THREAD;
        dati[i].fine = (i + 1) * N_ELEMENTI / N_THREAD;
        if ((err = pthread_create(&dati[i].tid, NULL, somma_parziale, &dati[i])) != 0) {
            fprintf(stderr, "pthread_create: %s\n", strerror(err));
            exit(EXIT_FAILURE);
        }
    }

    /* attesa della fine dei thread e raccolta dei risultati */
    for (int i = 0; i < N_THREAD; i++) {
        pthread_join(dati[i].tid, NULL);
        totale += dati[i].somma;
    }
    printf("[MAIN] somma totale: %ld\n", totale);
    exit(EXIT_SUCCESS);
}
```

7.6 Dati condivisi e corse critiche

Quando più thread modificano lo stesso dato senza sincronizzazione si ha una **corsa critica**. Anche un'istruzione semplice come `contatore++` non è atomica: si traduce nella lettura del valore in un registro, nell'incremento e nella scrittura in memoria, e due thread che la eseguono contemporaneamente possono leggere lo stesso valore e perdere un incremento. Il risultato cambia da un'esecuzione all'altra, e l'errore può non manifestarsi per molte esecuzioni consecutive: per questo le corse critiche vanno **evitate per costruzione**, proteggendo ogni accesso ai dati condivisi con gli strumenti dei prossimi capitoli, e non cercate con le prove.

Mutex e semafori

I due strumenti di sincronizzazione più usati: il mutex per la mutua esclusione e il semaforo numerico per contare risorse ed eventi.

8.1 I mutex

PROTOTIPI (PTHREAD.H)

```
int pthread_mutex_init(pthread_mutex_t *mutex, const 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);
```

Un **mutex** realizza la mutua esclusione: un thread lo acquisisce con `pthread_mutex_lock` prima di entrare nella sezione critica e lo rilascia con `pthread_mutex_unlock` all'uscita; se il mutex è già posseduto da un altro thread, la lock blocca il chiamante finché non viene rilasciato. La `pthread_mutex_trylock` non si blocca mai: se il mutex è occupato restituisce subito `EBUSY`. Tutti i thread devono usare **lo stesso** oggetto `pthread_mutex_t`, che si trova quindi nella struttura condivisa.

Un mutex si inizializza con `pthread_mutex_init` (con attributi `NULL`) e si distrugge con `pthread_mutex_destroy` quando non serve più; un mutex dichiarato staticamente si può inizializzare anche con la costante `PTHREAD_MUTEX_INITIALIZER`. Il mutex ha un **proprietario**: deve essere rilasciato dallo stesso thread che lo ha acquisito.

CONTATORE.C

```
/* contatore.c: più thread incrementano un contatore condiviso protetto da mutex */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define N_THREAD 4
#define INCREMENTI 1000000

/* dati condivisi: il contatore e il mutex che lo protegge */
typedef struct {
    long contatore;
    pthread_mutex_t mutex;
} condivisi;

typedef struct {
    pthread_t tid;
    condivisi *c;
} dati_thread;
```

CONTATORE.C (CONTINUA)

```
void *incrementa(void *arg) {
    dati_thread *d = arg;
    for (int i = 0; i < INCREMENTI; i++) {
        pthread_mutex_lock(&d->c->mutex);    /* ingresso in sezione critica */
        d->c->contatore++;
        pthread_mutex_unlock(&d->c->mutex);    /* uscita dalla sezione critica */
    }
    return NULL;
}

int main(void) {
    condivisi c;
    dati_thread dati[N_THREAD];

    c.contatore = 0;
    pthread_mutex_init(&c.mutex, NULL);

    for (int i = 0; i < N_THREAD; i++) {
        dati[i].c = &c;
        pthread_create(&dati[i].tid, NULL, incrementa, &dati[i]);
    }
    for (int i = 0; i < N_THREAD; i++)
        pthread_join(dati[i].tid, NULL);

    printf("valore finale: %ld (atteso %ld)\n", c.contatore,
           (long)N_THREAD * INCREMENTI);
    pthread_mutex_destroy(&c.mutex);
    exit(EXIT_SUCCESS);
}
```

Senza le due chiamate al mutex il valore finale sarebbe quasi sempre inferiore a quello atteso. La sezione critica va tenuta **più breve possibile**: dentro vanno solo gli accessi ai dati condivisi, mentre le elaborazioni lunghe, le stampe e l'I/O sui file vanno fatte fuori, su copie locali dei dati.

8.2 I semafori numerici POSIX

PROTOTIPI (SEMAPHORE.H)

```
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);
```

Un **semaforo** contiene un valore intero non negativo. La `sem_wait` (l'operazione down) decrementa il valore se è positivo, altrimenti blocca il chiamante finché non diventa positivo; la `sem_post` (l'operazione up) incrementa il valore e, se ci sono thread bloccati, ne sblocca uno; la `sem_trywait` prova a decrementare senza bloccarsi e, se il valore è 0, fallisce con l'errore `EAGAIN`. La `sem_init` imposta il **valore iniziale**; il secondo argomento vale **0** per un semaforo usato fra thread dello stesso processo e un valore diverso da 0 per un semaforo condiviso fra processi diversi attraverso una zona di memoria condivisa.

DEFINIZIONE

A differenza delle funzioni dei pthread, le funzioni dei semafori seguono la convenzione delle chiamate di sistema: restituiscono 0 in caso di successo e **meno 1** in caso di errore, con il codice in `errno`. Gli errori vanno quindi riportati con `perror`.

A differenza del mutex, un semaforo **non ha proprietario**: la `sem_post` può essere eseguita da un thread diverso da quello che ha eseguito la `sem_wait`. È ciò che lo rende adatto non solo alla mutua esclusione (un semaforo con valore iniziale 1), ma anche a **contare** risorse disponibili o eventi avvenuti e a far attendere un thread finché un altro non ha prodotto qualcosa.

8.3 Il produttore e consumatore con i semafori

Lo schema più importante della programmazione concorrente è il **buffer limitato**: alcuni thread inseriscono elementi in una coda di capienza fissa e altri li estrarono. Servono tre strumenti: un semaforo **vuoti**, con valore iniziale pari alla capienza, che conta i posti liberi e blocca chi inserisce quando la coda è piena; un semaforo **pieni**, con valore iniziale 0, che conta gli elementi presenti e blocca chi estrae quando la coda è vuota; un **mutex** che protegge la coda durante l'inserimento e l'estrazione. La coda si realizza con un vettore circolare e due indici, in per il prossimo inserimento e out per la prossima estrazione, che avanzano modulo la capienza.

PRODCONS_SEM.C

```
/* prodcons_sem.c: produttore e consumatore con coda limitata, semafori e mutex */
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define CAPIENZA 5
#define N_NUMERI 20

/* coda circolare condivisa con gli strumenti di sincronizzazione */
typedef struct {
    int dati[CAPIENZA];
    int in, out;           /* prossima posizione di inserimento e di estrazione */
    sem_t vuoti;           /* posti liberi: parte da CAPIENZA */
    sem_t pieni;           /* elementi presenti: parte da 0 */
    pthread_mutex_t mutex; /* accesso esclusivo a dati, in, out */
} coda;

void inserisci(coda *q, int valore) {
    sem_wait(&q->vuoti);           /* attende un posto libero */
    pthread_mutex_lock(&q->mutex);
    q->dati[q->in] = valore;
    q->in = (q->in + 1) % CAPIENZA;
    pthread_mutex_unlock(&q->mutex);
    sem_post(&q->pieni);           /* segnala un nuovo elemento */
}

int estrai(coda *q) {
    int valore;
    sem_wait(&q->pieni);           /* attende un elemento */
    pthread_mutex_lock(&q->mutex);
    valore = q->dati[q->out];
    q->out = (q->out + 1) % CAPIENZA;
    pthread_mutex_unlock(&q->mutex);
    sem_post(&q->vuoti);           /* segnala un posto liberato */
}
```

PRODCONS_SEM.C (CONTINUA)

```
    return valore;
}

void *produttore(void *arg) {
    coda *q = arg;
    for (int i = 1; i <= N_NUMERI; i++) {
        inserisci(q, i * i);
        printf("[P] inserito %d\n", i * i);
    }
    inserisci(q, 0);          /* valore sentinella: fine dei lavori */
    return NULL;
}

void *consumatore(void *arg) {
    coda *q = arg;
    long somma = 0;
    int valore;
    while ((valore = estrai(q)) != 0) { /* si ferma alla sentinella */
        printf("[C] estratto %d\n", valore);
        somma += valore;
    }
    printf("[C] somma dei valori: %ld\n", somma);
    return NULL;
}

int main(void) {
    coda q;
    pthread_t tp, tc;

    q.in = q.out = 0;
    /* secondo argomento 0: semaforo usato fra thread dello stesso processo */
    sem_init(&q.vuoti, 0, CAPIENZA);
    sem_init(&q.pieni, 0, 0);
    pthread_mutex_init(&q.mutex, NULL);

    pthread_create(&tp, NULL, produttore, &q);
    pthread_create(&tc, NULL, consumatore, &q);
    pthread_join(tp, NULL);
    pthread_join(tc, NULL);

    sem_destroy(&q.vuoti);
    sem_destroy(&q.pieni);
    pthread_mutex_destroy(&q.mutex);
    exit(EXIT_SUCCESS);
}
```

Il produttore segnala la fine dei lavori inserendo un **valore sentinella** (qui 0, che non può essere un dato valido), e il consumatore termina quando lo estrae. Con più consumatori servono tante sentinelle quanti sono i consumatori; il capitolo 10 descrive questa e le altre tecniche di terminazione.

DA RICORDARE

Perché nell'inserimento la `sem_wait` su vuoti va eseguita prima di acquisire il mutex?

Perché se un produttore acquisisse il mutex e poi si bloccasse sulla `sem_wait` a coda piena, terrebbe il mutex mentre attende: nessun consumatore potrebbe più acquisirlo per estrarre un elemento e liberare un posto, e i thread resterebbero bloccati per sempre in uno **stallo**. Il mutex va preso solo quando si è sicuri di poter procedere, cioè dopo la `sem_wait`, e va rilasciato prima della `sem_post`.

8.4 Stampare senza mescolare l'output

Ogni chiamata a `printf` è di per sé sicura con i thread, ma una riga composta da più chiamate (ad esempio un ciclo che stampa gli elementi di un vettore) può essere interrotta dalle stampe di altri thread. Le funzioni `flockfile(stdout)` e `funlockfile(stdout)` acquisiscono e rilasciano il lock interno dello stream: le stampe eseguite fra le due compaiono di seguito, senza intrusioni.

UNA RIGA COMPOSTA DA PIÙ STAMPE

```
flockfile(stdout);                /* da qui nessun altro thread stampa su stdout */
printf("[T%d] valori:", numero);
for (int j = 0; j < n; j++)
    printf(" %d", v[j]);
printf("\n");
funlockfile(stdout);              /* le altre stampe possono riprendere */
```

Il lock dello stream è **ricorsivo**: lo stesso thread può chiamare `printf`, che lo acquisisce a sua volta, senza bloccarsi. Fra le due chiamate non vanno però eseguite operazioni lunghe o attese su altri strumenti di sincronizzazione, perché tutti gli altri thread che vogliono stampare restano fermi.

Variabili condizione, lock lettori e scrittori, barriere

Gli strumenti per realizzare i monitor, per distinguere lettori e scrittori e per sincronizzare gruppi di thread a fasi.

9.1 Le variabili condizione

PROTOTIPI (PTHREAD.H)

```
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);
```

Una **variabile condizione** permette a un thread di attendere che una condizione sui dati condivisi diventi vera, e si usa **sempre insieme a un mutex**, come nei monitor. Il thread acquisisce il mutex, controlla la condizione e, se non è soddisfatta, chiama `pthread_cond_wait`: la chiamata **rilascia il mutex** e sospende il thread in un'unica operazione atomica, e al risveglio **riacquisisce il mutex** prima di ritornare. Chi modifica i dati in modo da rendere vera la condizione chiama `pthread_cond_signal`, che risveglia almeno uno dei thread in attesa, oppure `pthread_cond_broadcast`, che li risveglia tutti. Una variabile condizione statica si può inizializzare con `PTHREAD_COND_INITIALIZER`.

DEFINIZIONE

La condizione va **sempre ricontrollata al risveglio**, con un ciclo `while` e non con un `if`: fra la `signal` e il momento in cui il thread risvegliato riacquisisce il mutex, un altro thread può aver modificato i dati, e lo standard ammette anche risvegli spuri, senza nessuna `signal`. Lo schema corretto è `while (condizione non vera) pthread_cond_wait(&cond, &mutex);`.

La `broadcast` serve quando un cambiamento può interessare **più thread** o thread che attendono condizioni diverse sulla stessa variabile: il caso tipico è la fine dei lavori, che deve risvegliare tutti i consumatori in attesa perché possano accorgersi che non arriveranno altri dati e terminare.

9.2 Un monitor con più produttori e consumatori

Il programma seguente realizza la stessa coda limitata del capitolo precedente nello stile dei **monitor**: la struttura della coda contiene sia i dati sia il mutex e le due variabili condizione, e le funzioni `inserisci` ed `estrai` sono le sole a toccarla. La coda conta anche i **produttori attivi**: quando l'ultimo termina, risveglia con una `broadcast` i consumatori, che escono quando trovano la coda vuota e nessun produttore attivo.

PRODCONS_COND.C

```
/* prodcons_cond.c: più produttori e consumatori con un monitor
 * (mutex e variabili condizione) */
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define CAPIENZA 5
#define N_PROD 3
#define N_CONS 2
#define N_NUMERI 10          /* numeri prodotti da ogni produttore */

/* il "monitor": dati della coda e strumenti di sincronizzazione insieme */
typedef struct {
    int dati[CAPIENZA];
    int in, out, n;           /* n: elementi presenti */
    int produttori_attivi;   /* quando arriva a 0 non arriveranno altri dati */
    pthread_mutex_t mutex;
    pthread_cond_t non_piena; /* attesa dei produttori */
    pthread_cond_t non_vuota; /* attesa dei consumatori */
} coda;

typedef struct {
    pthread_t tid;
    int numero;
    coda *q;
} dati_thread;

void inserisci(coda *q, int valore) {
    pthread_mutex_lock(&q->mutex);
    while (q->n == CAPIENZA)          /* condizione ricontrollata al risveglio */
        pthread_cond_wait(&q->non_piena, &q->mutex);
    q->dati[q->in] = valore;
    q->in = (q->in + 1) % CAPIENZA;
    q->n++;
    pthread_cond_signal(&q->non_vuota);
    pthread_mutex_unlock(&q->mutex);
}

/* restituisce false quando la coda è vuota e i produttori hanno finito */
bool estrai(coda *q, int *valore) {
    pthread_mutex_lock(&q->mutex);
    while (q->n == 0 && q->produttori_attivi > 0)
        pthread_cond_wait(&q->non_vuota, &q->mutex);
    if (q->n == 0) {                  /* vuota e nessun produttore attivo */
        pthread_mutex_unlock(&q->mutex);
        return false;
    }
    *valore = q->dati[q->out];
    q->out = (q->out + 1) % CAPIENZA;
    q->n--;
    pthread_cond_signal(&q->non_piena);
    pthread_mutex_unlock(&q->mutex);
    return true;
}
```

PRODCONS_COND.C (CONTINUA)

```
void fine_prodotto(coda *q) {
    pthread_mutex_lock(&q->mutex);
    q->produttori_attivi--;
    if (q->produttori_attivi == 0) /* risveglia tutti i consumatori in attesa */
        pthread_cond_broadcast(&q->non_vuota);
    pthread_mutex_unlock(&q->mutex);
}

void *produttore(void *arg) {
    dati_thread *d = arg;
    for (int i = 1; i <= N_NUMERI; i++)
        inserisci(d->q, d->numero * 100 + i);
    fine_prodotto(d->q);
    return NULL;
}

void *consumatore(void *arg) {
    dati_thread *d = arg;
    int valore, quanti = 0;
    while (estrai(d->q, &valore)) {
        printf("[C%d] estratto %d\n", d->numero, valore);
        quanti++;
    }
    printf("[C%d] fine: %d valori estratti\n", d->numero, quanti);
    return NULL;
}

int main(void) {
    coda q;
    dati_thread prod[N_PROD], cons[N_CONS];

    q.in = q.out = q.n = 0;
    q.produttori_attivi = N_PROD;
    pthread_mutex_init(&q.mutex, NULL);
    pthread_cond_init(&q.non_piena, NULL);
    pthread_cond_init(&q.non_vuota, NULL);

    for (int i = 0; i < N_PROD; i++) {
        prod[i].numero = i + 1;
        prod[i].q = &q;
        pthread_create(&prod[i].tid, NULL, produttore, &prod[i]);
    }
    for (int i = 0; i < N_CONS; i++) {
        cons[i].numero = i + 1;
        cons[i].q = &q;
        pthread_create(&cons[i].tid, NULL, consumatore, &cons[i]);
    }
    for (int i = 0; i < N_PROD; i++)
        pthread_join(prod[i].tid, NULL);
    for (int i = 0; i < N_CONS; i++)
        pthread_join(cons[i].tid, NULL);

    pthread_mutex_destroy(&q.mutex);
    pthread_cond_destroy(&q.non_piena);
    pthread_cond_destroy(&q.non_vuota);
    exit(EXIT_SUCCESS);
}
```

9.3 I lock per lettori e scrittori

PROTOTIPI (PTHREAD.H)

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const 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);
```

Un **lock per lettori e scrittori** risolve il problema omonimo: più thread possono possederlo insieme **in lettura** (rdlock), ma un thread che lo acquisisce **in scrittura** (wrlock) lo possiede in modo esclusivo. Conviene rispetto a un mutex quando i dati condivisi vengono letti molto più spesso di quanto vengano modificati, come in una tabella consultata da molti thread e aggiornata di rado. Il rilascio avviene con pthread_rwlock_unlock in entrambi i casi; le varianti try non si bloccano e la costante PTHREAD_RWLOCK_INITIALIZER inizializza un lock statico.

9.4 Le barriere

PROTOTIPI (PTHREAD.H)

```
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);
```

Una **barriera** sincronizza un gruppo di count thread: ogni thread che chiama pthread_barrier_wait si blocca finché non l'hanno chiamata tutti i count thread, poi vengono sbloccati tutti insieme. È lo strumento adatto agli algoritmi **a fasi**, in cui nessun thread deve iniziare la fase successiva prima che tutti abbiano completato quella corrente (ad esempio un ordinamento in cui ogni thread ordina una parte del vettore e poi le parti vengono fuse). Al momento dello sblocco, **un solo** thread riceve il valore speciale PTHREAD_BARRIER_SERIAL_THREAD e gli altri ricevono 0: quel thread può fare da coordinatore per un'operazione da eseguire una volta sola. Dopo lo sblocco la barriera torna utilizzabile per lo stesso numero di thread.

BARRIERA.C

```
/* barriera.c: i thread lavorano in due fasi separate da una barriera */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define N_THREAD 4

typedef struct {
    pthread_t tid;
    int numero;
    pthread_barrier_t *barriera;
} dati_thread;

void *lavora(void *arg) {
    dati_thread *d = arg;
    printf("[T%d] fase 1 completata\n", d->numero);
```

BARRIERA.C (CONTINUA)

```
/* nessuno prosegue finché tutti non hanno finito la fase 1 */
int esito = pthread_barrier_wait(d->barriera);
if (esito == PTHREAD_BARRIER_SERIAL_THREAD)
    printf("[T%d] scelto dalla barriera come coordinatore\n", d->numero);

printf("[T%d] fase 2 completata\n", d->numero);
return NULL;
}

int main(void) {
    pthread_barrier_t barriera;
    dati_thread dati[N_THREAD];

    pthread_barrier_init(&barriera, NULL, N_THREAD); /* soglia: N_THREAD thread */
    for (int i = 0; i < N_THREAD; i++) {
        dati[i].numero = i + 1;
        dati[i].barriera = &barriera;
        pthread_create(&dati[i].tid, NULL, lavora, &dati[i]);
    }
    for (int i = 0; i < N_THREAD; i++)
        pthread_join(dati[i].tid, NULL);
    pthread_barrier_destroy(&barriera);
    exit(EXIT_SUCCESS);
}
```

Progettare un programma concorrente

Un metodo per passare dal testo di un problema a un programma con più thread corretto: ruoli, strutture condivise, buffer, terminazione e verifica.

10.1 Dal problema alla struttura dei thread

I problemi di programmazione concorrente sui sistemi UNIX hanno spesso la stessa forma: alcuni thread **producono** dati (leggendo file o scansionando directory), altri li **elaborano** e il thread principale **raccoglie** i risultati. I thread sono collegati da **buffer condivisi** di capienza limitata, e formano una catena di stadi, una *pipeline*.



Lo schema a stadi: i produttori riempiono un buffer limitato, gli elaboratori lo svuotano e passano i risultati al thread principale.

Il primo passo è individuare con precisione, dal testo del problema, i **ruoli** dei thread e quanti sono, i **buffer** con la loro capienza e il tipo di elemento che contengono, gli strumenti di sincronizzazione richiesti (semafori e mutex, oppure mutex e variabili condizione) e il **formato dell'output**. Conviene scrivere prima le strutture dati, poi le funzioni di inserimento ed estrazione, poi le funzioni dei thread e infine il `main`.

10.2 Le strutture dati condivise

Tutti i dati condivisi stanno in **una struttura comune** creata nel `main`: i buffer con i loro indici, i semafori, i mutex, le variabili condizione e i contatori per la fine dei lavori. Ogni thread riceve una struttura privata che contiene, oltre ai propri dati, un puntatore a quella comune. Se il numero di thread dipende dagli argomenti, il vettore delle strutture private si alloca con `malloc`. In questo modo non serve nessuna variabile globale, e ogni funzione dichiara esplicitamente, attraverso i parametri, i dati su cui lavora.

Gli elementi di un buffer possono essere **valori** (una struttura copiata per intero, come un record con un vettore o un blocco di un file) oppure **puntatori** a dati allocati dinamicamente. I valori sono più semplici da gestire, perché il buffer contiene una copia indipendente; i puntatori sono adatti a dati di dimensione variabile, come le stringhe.

10.3 Il buffer limitato: due realizzazioni

	SEMAFORI E MUTEX	MUTEX E VARIABILI CONDIZIONE
Strumenti	Semafori vuoti (iniziale: capienza) e pieni (iniziale: 0), un mutex	Un mutex, variabili condizione <code>non_piena</code> e <code>non_vuota</code> , il contatore degli elementi
Inserimento	<code>sem_wait(vuoti)</code> , lock, inserimento, unlock, <code>sem_post(pieni)</code>	Lock, while piena <code>wait(non_piena)</code> , inserimento, <code>signal(non_vuota)</code> , unlock
Estrazione	<code>sem_wait(pieni)</code> , lock, estrazione, unlock, <code>sem_post(vuoti)</code>	Lock, while vuota <code>wait(non_vuota)</code> , estrazione, <code>signal(non_piena)</code> , unlock
Fine lavori	Elementi sentinella, uno per ogni consumatore	Contatore dei produttori attivi e broadcast

Le due realizzazioni sono equivalenti. Con i semafori il conteggio dei posti e degli elementi è affidato ai semafori stessi; con le variabili condizione è il codice a controllare esplicitamente lo stato della coda, ed è più facile aggiungere condizioni particolari, come la fine dei lavori. Un buffer da **un solo elemento** (un «record» che passa i risultati al thread principale) è un caso particolare: con i semafori bastano due semafori, uno con valore iniziale 1 (record libero) e uno con valore iniziale 0 (record pieno), e il mutex non serve, perché il semaforo del record libero già garantisce che un solo thread alla volta lo scriva.

10.4 Far terminare i thread spontaneamente

Ogni thread deve accorgersi da solo che il lavoro è finito e terminare ritornando dalla propria funzione, senza essere interrotto dall'esterno. Le tecniche più usate sono tre.

TECNICA	FUNZIONAMENTO
Sentinella	Un elemento speciale (un puntatore NULL, un valore non valido, un record con un flag di fine) indica che non arriveranno altri dati; il consumatore che lo estrae termina. Serve una sentinella per ogni consumatore
Contatore dei produttori	Un contatore condiviso, protetto da mutex, parte dal numero dei produttori e viene decrementato da ognuno quando finisce: il produttore che lo porta a 0 sa di essere l'ultimo e inserisce le sentinelle
Flag con variabile condizione	Con i monitor, il consumatore attende finché la coda è vuota e ci sono produttori attivi; quando l'ultimo produttore termina esegue una broadcast, e i consumatori trovano la coda vuota senza produttori ed escono

La catena si propaga da uno stadio all'altro: l'ultimo produttore avvisa gli elaboratori, l'ultimo elaboratore avvisa il thread principale, e il main, dopo aver ricevuto il segnale di fine, esegue la `pthread_join` su tutti i thread e libera le risorse.

DA RICORDARE

Perché la sentinella non può essere inserita da ogni produttore quando finisce?

Perché i consumatori terminerebbero alla prima sentinella estratta, mentre gli altri produttori potrebbero avere ancora dati da inserire: quei dati resterebbero nel buffer senza nessuno che li elabori, e i produttori potrebbero bloccarsi per sempre a buffer pieno. Le sentinelle vanno inserite solo quando **tutti** i produttori hanno finito, ed è il contatore condiviso a dire chi è l'ultimo.

10.5 Stringhe e memoria

Quando nel buffer si inseriscono **puntatori** a stringhe, ogni stringa deve essere una copia allocata dinamicamente (ad esempio con `strdup`) e non l'indirizzo di una variabile locale del produttore, che verrebbe sovrascritta alla successiva iterazione. Chi estrae la stringa ne diventa responsabile e la libera con `free` quando non serve più. Quando invece gli elementi sono **strutture con vettori di caratteri** (ad esempio un percorso di al massimo 1024 caratteri), la stringa si copia nel record con `strncpy` o `snprintf`, controllando che entri nello spazio disponibile.

10.6 Controllo degli errori e verifica

Ogni chiamata che può fallire va controllata: le chiamate di sistema e le funzioni dei semafori con `perror`, le funzioni dei pthread con `strerror` sul codice restituito. Il programma va compilato con `-Wall` e senza avvisi. Per verificarlo conviene preparare casi di prova di cui si conosce il risultato (directory con pochi file di dimensione nota, file binari generati apposta) e ripetere l'esecuzione molte volte, perché un errore di sincronizzazione può manifestarsi raramente. Lo strumento ThreadSanitizer, attivato compilando con `-fsanitize=thread`, individua durante l'esecuzione gli accessi ai dati condivisi non protetti.

10.7 Una lista di controllo

1. Tutti i dati condivisi stanno in una struttura creata nel `main`, e non ci sono variabili globali.
2. Ogni accesso a un dato condiviso avviene con il mutex che lo protegge acquisito, e le sezioni critiche non contengono attese inutili o I/O lungo.
3. Le `sem_wait` su vuoti e pieni precedono l'acquisizione del mutex; le `pthread_cond_wait` sono dentro un ciclo `while`.
4. Ogni thread riceve l'indirizzo di dati propri e validi per tutta la sua durata.
5. La fine dei lavori raggiunge tutti i thread di ogni stadio, e il `main` attende tutti i thread con `pthread_join`.
6. La memoria allocata dinamicamente viene liberata, e semafori, mutex e variabili condizione vengono distrutti alla fine.
7. L'output rispetta il formato richiesto, e le righe composte da più stampe sono protette con `flockfile`.

Esempio svolto: dimensione dei file in più directory

Un programma con thread di scansione, un thread di elaborazione e il thread principale, coordinati con semafori e mutex.

11.1 Il problema

Si vuole scrivere un programma dimensioni, da invocare come dimensioni <dir 1> ... <dir n>, che calcoli la dimensione totale in byte dei file regolari contenuti direttamente nelle directory indicate, senza scendere nelle sottodirectory. Il programma crea $n + 1$ thread. Gli n thread **DIR** scansionano in parallelo una directory ciascuno e, per ogni file regolare trovato, inseriscono il suo percorso in un **buffer condiviso di 10 percorsi**. Un thread **STAT** estrae i percorsi uno alla volta, determina la dimensione del file e passa la coppia (percorso, dimensione) al thread principale **MAIN** attraverso un **record** condiviso; il MAIN mantiene il totale. I thread si coordinano con **mutex e semafori** POSIX, devono terminare spontaneamente e non si usano variabili globali. L'output ha questa forma:

ESEMPIO DI OUTPUT

```
[DIR1] scansione della cartella '/usr/include'...
[DIR1] trovato il file 'stdio.h' in '/usr/include'
[STAT] il file '/usr/include/stdio.h' ha dimensione 31641 byte
[MAIN] con il file '/usr/include/stdio.h' il totale parziale è di 31641 byte
...
[MAIN] il totale finale è di 2828698 byte
```

11.2 La struttura del programma



I thread e le strutture condivise del programma dimensioni.

Gli strumenti di sincronizzazione sono due gruppi. Per il **buffer dei percorsi**, che ha più produttori (i thread DIR) e un consumatore (STAT), si usa lo schema classico: semafori vuoti e pieni e un mutex. Per il **record** fra STAT e MAIN, che contiene un solo elemento, bastano due semafori: `rec_vuoto` con valore iniziale 1 e `rec_pieno` con valore iniziale 0.

La fine dei lavori segue la catena descritta nel capitolo 10. Ogni thread DIR, quando ha finito la propria directory, decrementa il contatore `dir_attivi` sotto mutex; quello che lo porta a 0 inserisce nel buffer la **sentinella** NULL. STAT, quando estrae la sentinella, scrive nel record un segnale di fine (il campo `fine` a 1), e MAIN, quando lo riceve, esce dal ciclo, attende tutti i thread e stampa il totale.

11.3 Le strutture dati e le funzioni del buffer

DIMENSIONI.C, RIGHE DA 1 A 64

```
/* dimensioni.c: dimensione totale dei file regolari contenuti in più directory
 * uso: dimensioni <dir 1> <dir 2> ... <dir n>
 * thread: n thread DIR (scansione), un thread STAT, il thread principale MAIN */
#define _POSIX_C_SOURCE 200809L
#include <dirent.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#define CAPIENZA 10          /* capienza del buffer dei percorsi */
#define MAX_PERCORSO 1024

/* dati condivisi fra tutti i thread (allocati nel main, nessuna variabile globale) */
typedef struct {
    /* buffer circolare dei percorsi: DIR inseriscono, STAT estrae */
    char *percorsi[CAPIENZA];
    int in, out;
    sem_t vuoti, pieni;
    pthread_mutex_t mutex;
    int dir_attivi;          /* thread DIR non ancora terminati (protetto da mutex) */

    /* record singolo per passare i risultati da STAT a MAIN */
    char percorso[MAX_PERCORSO];
    long long dimensione;
    int fine;                /* 1: STAT ha finito e il record non contiene dati */
    sem_t rec_vuoto, rec_pieno;
} condivisi;

/* dati privati di un thread DIR */
typedef struct {
    pthread_t tid;
    int numero;
    char *cartella;
    condivisi *c;
} dati_dir;

void errore(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

/* inserimento nel buffer dei percorsi (NULL indica la fine dei lavori) */
void inserisci(condivisi *c, char *percorso) {
    sem_wait(&c->vuoti);
    pthread_mutex_lock(&c->mutex);
    c->percorsi[c->in] = percorso;
    c->in = (c->in + 1) % CAPIENZA;
    pthread_mutex_unlock(&c->mutex);
    sem_post(&c->pieni);
}
```

DIMENSIONI.C, RIGHE DA 1 A 64 (CONTINUA)

```
char *estrai(condivisi *c) {
    char *percorso;
    sem_wait(&c->pieni);
    pthread_mutex_lock(&c->mutex);
    percorso = c->percorsi[c->out];
    c->out = (c->out + 1) % CAPIENZA;
    pthread_mutex_unlock(&c->mutex);
    sem_post(&c->vuoti);
    return percorso;
}
```

Nel buffer si inseriscono **puntatori**: il thread DIR costruisce il percorso in una variabile locale e ne inserisce una copia allocata con `strdup`, che STAT libererà dopo l'uso. Le funzioni `inserisci` ed `estrai` sono le sole a toccare il buffer, e acquisiscono il mutex solo dopo la `sem_wait`.

11.4 I thread DIR e STAT

DIMENSIONI.C, RIGHE DA 66 A 125

```
void *thread_dir(void *arg) {
    dati_dir *d = arg;
    DIR *dir;
    struct dirent *voce;
    struct stat info;
    char percorso[MAX_PERCORSO];
    int ultimo;

    printf("[DIR%d] scansione della cartella '%s'...\n", d->numero, d->cartella);
    if ((dir = opendir(d->cartella)) == NULL)
        errore(d->cartella);

    while ((voce = readdir(dir)) != NULL) {
        snprintf(percorso, MAX_PERCORSO, "%s/%s", d->cartella, voce->d_name);
        if (lstat(percorso, &info) < 0)
            errore(percorso);
        if (S_ISREG(info.st_mode)) {
            printf("[DIR%d] trovato il file '%s' in '%s'\n", d->numero,
                voce->d_name, d->cartella);
            /* copia dinamica: la variabile locale verrà riusata */
            inserisci(d->c, strdup(percorso));
        }
    }
    closedir(dir);

    /* l'ultimo thread DIR che termina inserisce la sentinella NULL */
    pthread_mutex_lock(&d->c->mutex);
    d->c->dir_attivi--;
    ultimo = (d->c->dir_attivi == 0);
    pthread_mutex_unlock(&d->c->mutex);
    if (ultimo)
        inserisci(d->c, NULL);
    return NULL;
}
```

DIMENSIONI.C, RIGHE DA 66 A 125 (CONTINUA)

```
void *thread_stat(void *arg) {
    condivisi *c = arg;
    char *percorso;
    struct stat info;

    while ((percorso = estrai(c)) != NULL) {
        if (stat(percorso, &info) < 0)
            errore(percorso);
        printf("[STAT] il file '%s' ha dimensione %lld byte\n", percorso,
            (long long)info.st_size);

        /* passaggio del risultato a MAIN attraverso il record singolo */
        sem_wait(&c->rec_vuoto);
        strcpy(c->percorso, percorso);
        c->dimensione = info.st_size;
        sem_post(&c->rec_pieno);
        free(percorso);
    }

    /* fine dei lavori: record speciale per MAIN */
    sem_wait(&c->rec_vuoto);
    c->fine = 1;
    sem_post(&c->rec_pieno);
    return NULL;
}
```

Il thread DIR usa `lstat` per riconoscere i file regolari senza seguire i link simbolici, poi inserisce il percorso; dopo l'ultima voce verifica, sotto mutex, se è l'ultimo thread DIR ancora attivo. Il thread STAT esegue un ciclo che termina quando `estrai` restituisce la sentinella; per ogni percorso ottiene la dimensione con `stat` e la consegna a MAIN attendendo prima che il record sia libero. La `free` avviene dopo la copia del percorso nel record.

11.5 Il thread principale

DIMENSIONI.C, RIGHE DA 127 A 184

```
int main(int argc, char *argv[]) {
    condivisi c;
    dati_dir *dirs;
    pthread_t tid_stat;
    int n = argc - 1;
    long long totale = 0;

    if (n < 1) {
        fprintf(stderr, "uso: %s <dir 1> ... <dir n>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* inizializzazione dei dati condivisi */
    c.in = c.out = 0;
    c.dir_attivi = n;
    c.fine = 0;
    sem_init(&c.vuoti, 0, CAPIENZA);
    sem_init(&c.pieni, 0, 0);
    sem_init(&c.rec_vuoto, 0, 1);          /* il record è inizialmente libero */
    sem_init(&c.rec_pieno, 0, 0);
    pthread_mutex_init(&c.mutex, NULL);
}
```

DIMENSIONI.C, RIGHE DA 127 A 184 (CONTINUA)

```
/* creazione dei thread */
if ((dirs = malloc(n * sizeof(dati_dir))) == NULL)
    errore("malloc");
for (int i = 0; i < n; i++) {
    dirs[i].numero = i + 1;
    dirs[i].cartella = argv[i + 1];
    dirs[i].c = &c;
    pthread_create(&dirs[i].tid, NULL, thread_dir, &dirs[i]);
}
pthread_create(&tid_stat, NULL, thread_stat, &c);

/* MAIN: raccolta dei risultati fino al record di fine */
while (1) {
    sem_wait(&c.rec_pieno);
    if (c.fine)
        break;
    totale += c.dimensione;
    printf("[MAIN] con il file '%s' il totale parziale è di %lld byte\n",
           c.percorso, totale);
    sem_post(&c.rec_vuoto);
}

for (int i = 0; i < n; i++)
    pthread_join(dirs[i].tid, NULL);
pthread_join(tid_stat, NULL);
printf("[MAIN] il totale finale è di %lld byte\n", totale);

/* liberazione delle risorse */
free(dirs);
sem_destroy(&c.vuoti);
sem_destroy(&c.pieni);
sem_destroy(&c.rec_vuoto);
sem_destroy(&c.rec_pieno);
pthread_mutex_destroy(&c.mutex);
exit(EXIT_SUCCESS);
}
```

Il MAIN inizializza la struttura condivisa, crea i thread passando a ognuno l'indirizzo della propria struttura privata e poi si comporta da consumatore del record: legge il risultato dopo `sem_wait(&c.rec_pieno)` e libera il record con `sem_post(&c.rec_vuoto)`. Quando trova il segnale di fine esce dal ciclo senza liberare il record, perché nessuno lo scriverà più, e attende tutti i thread.

DA RICORDARE

Perché la sentinella viene inserita dall'ultimo thread DIR e non dal main dopo la `pthread_join` dei thread DIR?

Le due soluzioni sono entrambe corrette dal punto di vista della terminazione, ma la seconda richiederebbe al main di attendere i thread DIR **mentre** dovrebbe già consumare i risultati dal record: il main resterebbe bloccato nella `pthread_join`, STAT si bloccherebbe sul record pieno, il buffer si riempirebbe e i thread DIR resterebbero bloccati in inserisci: con più file di quanti ne contengano il buffer e il record, il programma si bloccherebbe in uno stallo. Con il contatore condiviso la fine dei lavori si propaga lungo la catena senza che nessuno debba attendere fuori posto.

11.6 La verifica

Il programma si compila con `gcc -Wall -pthread -o dimensioni dimensioni.c`. Per verificarne il risultato si confronta il totale con quello ottenuto in altro modo, ad esempio con un breve script che somma le dimensioni dei file regolari delle stesse directory, e si ripete l'esecuzione più volte: l'ordine delle righe cambia da un'esecuzione all'altra, ma il totale finale deve essere sempre lo stesso. Il totale di riferimento si può ottenere dalla shell con `find`, che con le opzioni indicate elenca le dimensioni dei soli file regolari contenuti direttamente nelle directory, senza seguire i link simbolici; un ciclo ripete poi il programma e conta le diverse righe finali ottenute, che devono essere una sola:

VERIFICA DALLA SHELL

```
# totale atteso, senza ricorsione
find /usr/include /etc -maxdepth 1 -type f -printf '%s\n' | awk '{ s += $1 } END { print s }'

# venti esecuzioni: deve comparire una sola riga, ripetuta 20 volte
for i in $(seq 20); do ./dimensioni /usr/include /etc | tail -n 1; done | sort | uniq -c
```

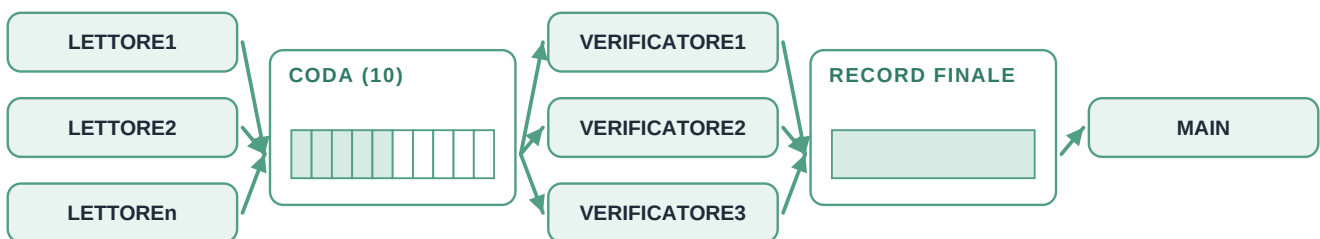
Esempio svolto: vettori in file binari con mmap

Un programma con più lettori che usano la mappatura dei file, più verificatori concorrenti e il thread principale, coordinati con semafori e mutex.

12.1 Il problema

Si vuole scrivere un programma `vettori`, da invocare come `vettori <file 1> ... <file n>`. Ogni file è binario e contiene una sequenza di vettori di **M = 12 elementi**, ognuno rappresentato da un byte senza segno. Un vettore si dice **equisomma** se la somma degli elementi nelle posizioni pari è uguale a quella degli elementi nelle posizioni dispari; ad esempio il vettore 30 20 10 15 5 25 40 10 20 5 25 55 è equisomma, perché entrambe le somme valgono 130.

Il programma crea $n + 3$ thread: n thread **LETTORI**, ognuno dei quali legge uno dei file usando **obbligatoriamente** la mappatura in memoria e inserisce ogni vettore in una **coda intermedia di 10 record**; **3 thread VERIFICATORI**, che in parallelo estraggono i vettori dalla coda e verificano la proprietà; il thread principale **MAIN**, che riceve i vettori equisomma attraverso un **record finale** da un solo vettore e li stampa. I thread si coordinano con semafori e mutex, terminano spontaneamente e non si usano variabili globali.



I thread e le strutture condivise del programma vettori.

12.2 Le strutture dati

VETTORI.C, RIGHE DA 1 A 70

```
/* vettori.c: individua i vettori "equisomma" contenuti in file binari
 * uso: vettori <file 1> ... <file n>
 * ogni file è una sequenza di record di M = 12 byte senza segno; un vettore è
 * equisomma se la somma delle posizioni pari è uguale a quella delle dispari
 * thread: n LETTORI (con mmap), V VERIFICATORI, il thread principale MAIN */
#define _POSIX_C_SOURCE 200809L
#include <fcntl.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
```

VETTORI.C, RIGHE DA 1 A 70 (CONTINUA)

```
#include <unistd.h>

#define M 12                /* elementi di un vettore */
#define CAPIENZA 10         /* capienza della coda intermedia */
#define N_VERIFICATORI 3

/* elemento della coda: un vettore oppure un segnale di fine lavori */
typedef struct {
    unsigned char v[M];
    int numero_file;        /* da quale file proviene */
    int fine;               /* 1: nessun dato, il verificatore deve terminare */
} record;

typedef struct {
    /* coda intermedia: LETTORI inseriscono, VERIFICATORI estraggono */
    record coda[CAPIENZA];
    int in, out;
    sem_t vuoti, pieni;
    pthread_mutex_t mutex;
    int lettori_attivi, verificatori_attivi; /* protetti da mutex */

    /* record finale: un solo vettore alla volta verso MAIN */
    record finale;
    sem_t fin_vuoto, fin_pieno;
} condivisi;

typedef struct {
    pthread_t tid;
    int numero;
    char *nome_file;
    condivisi *c;
} dati_thread;

void errore(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

void inserisci(condivisi *c, record r) {
    sem_wait(&c->vuoti);
    pthread_mutex_lock(&c->mutex);
    c->coda[c->in] = r;
    c->in = (c->in + 1) % CAPIENZA;
    pthread_mutex_unlock(&c->mutex);
    sem_post(&c->pieni);
}

record estrai(condivisi *c) {
    record r;
    sem_wait(&c->pieni);
    pthread_mutex_lock(&c->mutex);
    r = c->coda[c->out];
    c->out = (c->out + 1) % CAPIENZA;
    pthread_mutex_unlock(&c->mutex);
    sem_post(&c->vuoti);
    return r;
}
```

Gli elementi della coda sono **valori**: ogni record contiene una copia del vettore, quindi il lettore può riusare la propria variabile locale subito dopo l'inserimento. Il campo `fine` permette di usare lo stesso tipo anche per il segnale di fine lavori. Il record finale, scritto da più verificatori e letto da MAIN, è protetto dai due semafori `fin_vuoto` (valore iniziale 1) e `fin_pieno` (valore iniziale 0): un verificatore che trova un vettore equisomma esegue `sem_wait` su `fin_vuoto`, e poiché il semaforo vale al massimo 1, solo uno alla volta può scrivere nel record.

12.3 I lettori: la mappatura del file

VETTORI.C, RIGHE DA 72 A 113

```
void *lettore(void *arg) {
    dati_thread *d = arg;
    int fd, ultimo;
    struct stat info;
    unsigned char *mappa;
    record r;

    if ((fd = open(d->nome_file, O_RDONLY)) < 0)
        errore(d->nome_file);
    if (fstat(fd, &info) < 0)
        errore("fstat");
    long n_vettori = info.st_size / M;
    printf("[LETTORE%d] file '%s' con %ld vettori\n", d->numero, d->nome_file,
        n_vettori);

    if (n_vettori > 0) {
        mappa = mmap(NULL, info.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
        if (mappa == MAP_FAILED)
            errore("mmap");
        for (long k = 0; k < n_vettori; k++) {
            for (int j = 0; j < M; j++) /* copia del k esimo record */
                r.v[j] = mappa[k * M + j];
            r.numero_file = d->numero;
            r.fine = 0;
            inserisci(d->c, r);
        }
        munmap(mappa, info.st_size);
    }
    close(fd);

    /* l'ultimo lettore inserisce un segnale di fine per ogni verificatore */
    pthread_mutex_lock(&d->c->mutex);
    d->c->lettori_attivi--;
    ultimo = (d->c->lettori_attivi == 0);
    pthread_mutex_unlock(&d->c->mutex);
    if (ultimo) {
        r.fine = 1;
        for (int i = 0; i < N_VERIFICATORI; i++)
            inserisci(d->c, r);
    }
    return NULL;
}
```

Il lettore ricava la dimensione del file con `fstat` e calcola il numero di vettori dividendo per `M`; un file vuoto non viene mappato, perché la `mmap` non accetta lunghezza nulla. Il record numero `k` occupa i byte da `k * M` in poi della zona mappata. Alla fine il lettore decrementa il contatore `lettori_attivi`: l'ultimo inserisce nella coda un

segnale di fine per ogni verificatore, perché ogni verificatore termina quando ne estrae uno.

12.4 I verificatori e il thread principale

VETTORI.C, RIGHE DA 115 A 154

```
int equisomma(unsigned char v[M]) {
    int pari = 0, dispari = 0;
    for (int j = 0; j < M; j++) {
        if (j % 2 == 0)
            pari += v[j];
        else
            dispari += v[j];
    }
    return pari == dispari;
}

void *verificatore(void *arg) {
    dati_thread *d = arg;
    record r;
    int ultimo;

    while (1) {
        r = estrai(d->c);
        if (r.fine) /* segnale di fine lavori */
            break;
        if (equisomma(r.v)) {
            /* il semaforo fin_vuoto dà a un solo verificatore alla volta il record */
            sem_wait(&d->c->fin_vuoto);
            d->c->finale = r;
            sem_post(&d->c->fin_pieno);
        }
    }

    /* l'ultimo verificatore avvisa MAIN della fine dei lavori */
    pthread_mutex_lock(&d->c->mutex);
    d->c->verificatori_attivi--;
    ultimo = (d->c->verificatori_attivi == 0);
    pthread_mutex_unlock(&d->c->mutex);
    if (ultimo) {
        sem_wait(&d->c->fin_vuoto);
        d->c->finale.fine = 1;
        sem_post(&d->c->fin_pieno);
    }
    return NULL;
}
```

VETTORI.C, RIGHE DA 156 A 216

```
int main(int argc, char *argv[]) {
    condivisi c;
    int n = argc - 1, trovati = 0;
    dati_thread *lettori, verif[N_VERIFICATORI];
```

VETTORI.C, RIGHE DA 156 A 216 (CONTINUA)

```
if (n < 1) {
    fprintf(stderr, "uso: %s <file 1> ... <file n>\n", argv[0]);
    exit(EXIT_FAILURE);
}

c.in = c.out = 0;
c.lettori_attivi = n;
c.verificatori_attivi = N_VERIFICATORI;
sem_init(&c.vuoti, 0, CAPIENZA);
sem_init(&c.pieni, 0, 0);
sem_init(&c.fin_vuoto, 0, 1);
sem_init(&c.fin_pieno, 0, 0);
pthread_mutex_init(&c.mutex, NULL);

if ((lettori = malloc(n * sizeof(dati_thread))) == NULL)
    errore("malloc");
for (int i = 0; i < n; i++) {
    lettori[i].numero = i + 1;
    lettori[i].nome_file = argv[i + 1];
    lettori[i].c = &c;
    pthread_create(&lettori[i].tid, NULL, lettore, &lettori[i]);
}
for (int i = 0; i < N_VERIFICATORI; i++) {
    verif[i].numero = i + 1;
    verif[i].c = &c;
    pthread_create(&verif[i].tid, NULL, verificatore, &verif[i]);
}

/* MAIN: stampa dei vettori equisomma fino al segnale di fine */
while (1) {
    sem_wait(&c.fin_pieno);
    if (c.fine.fine)
        break;
    trovati++;
    flockfile(stdout);          /* la riga viene stampata senza interruzioni */
    printf("[MAIN] vettore equisomma dal file %d:", c.fine.numero_file);
    for (int j = 0; j < M; j++)
        printf(" %d", c.fine.v[j]);
    printf("\n");
    funlockfile(stdout);
    sem_post(&c.fin_vuoto);
}

for (int i = 0; i < n; i++)
    pthread_join(lettori[i].tid, NULL);
for (int i = 0; i < N_VERIFICATORI; i++)
    pthread_join(verif[i].tid, NULL);
printf("[MAIN] trovati %d vettori equisomma\n", trovati);

free(lettori);
sem_destroy(&c.vuoti);
sem_destroy(&c.pieni);
sem_destroy(&c.fin_vuoto);
sem_destroy(&c.fin_pieno);
pthread_mutex_destroy(&c.mutex);
exit(EXIT_SUCCESS);
```

I verificatori estraggono record finché non trovano il segnale di fine; poi, con un secondo contatore, l'ultimo di loro scrive nel record finale il segnale di fine per MAIN. Il MAIN stampa ogni vettore ricevuto con flockfile e funlockfile, perché la riga è composta da più chiamate a printf e non deve mescolarsi con le stampe degli altri thread.

DA RICORDARE

Perché i segnali di fine per i verificatori sono tanti quanti i verificatori, mentre per MAIN ne basta uno?

Ogni segnale di fine viene **consumato** da chi lo estrae: un verificatore che lo trova termina e non lo rimette in coda. Con un solo segnale terminerebbe un solo verificatore, e gli altri resterebbero bloccati per sempre sulla sem_wait della coda vuota. Il MAIN invece è un solo thread, e riceve il segnale di fine solo dall'ultimo verificatore, quando tutti gli altri hanno già finito di scrivere nel record.

12.5 La verifica

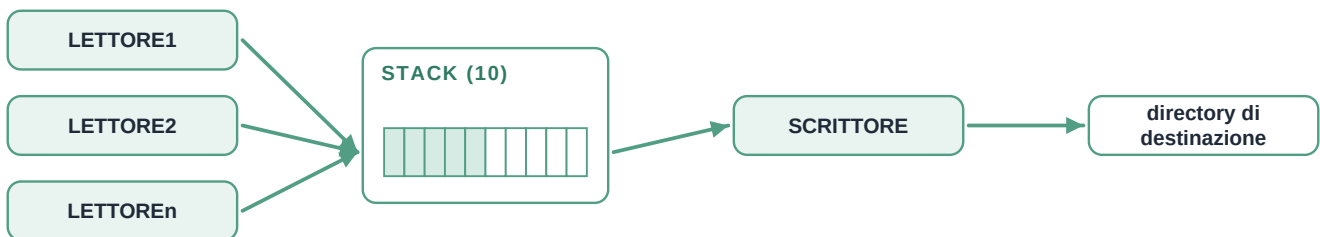
Per provare il programma servono file di cui si conosce il contenuto: si possono generare con un piccolo programma o script che scriva vettori casuali, alcuni costruiti apposta per essere equisomma, e che conti quanti vettori equisomma ha scritto. Il numero di vettori trovati dal programma deve coincidere con quello atteso in tutte le esecuzioni, anche se l'ordine delle stampe cambia, e il programma deve terminare correttamente anche con un file vuoto fra quelli indicati.

Esempio svolto: duplicazione di file a blocchi

Un programma con più lettori e un unico scrittore, coordinati con mutex e variabili condizione attraverso uno stack condiviso di blocchi.

13.1 Il problema

Si vuole scrivere un programma `duplica`, da invocare come `duplica <file 1> ... <file n> <directory>`, che crei nella directory indicata una copia di ciascun file. Il programma crea n thread **LETTORI**, uno per file, e un unico thread **SCRITTORE**. Ogni lettore legge il proprio file a **blocchi di 1 KiB** e inserisce ogni blocco in uno **stack condiviso** con capienza di 10 record; ogni record contiene il blocco, il nome del file, la dimensione totale del file, la posizione (offset) del blocco e il numero di byte validi. Lo scrittore crea la directory di destinazione se non esiste, estrae i record e scrive ogni blocco **nella posizione corretta** del file duplicato, creandolo se non esiste ancora; lo scrittore non accede mai ai file originali. I thread si coordinano con **mutex e variabili condizione**, terminano spontaneamente e il programma deve funzionare anche con i file binari.



I thread e lo stack condiviso del programma duplica.

13.2 Le strutture dati e il monitor

DUPLICA.C, RIGHE DA 1 A 80

```

/* duplica.c: duplica dei file in una directory, a blocchi di 1 KiB
 * uso: duplica <file 1> ... <file n> <directory destinazione>
 * thread: n LETTORI e un SCRITTORE, con mutex e variabili condizione */
#define _POSIX_C_SOURCE 200809L
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

```

DUPLICA.C, RIGHE DA 1 A 80 (CONTINUA)

```
#define DIM_BLOCCO 1024
#define CAPIENZA 10          /* capienza dello stack condiviso */
#define MAX_NOME 256
#define MAX_PERCORSO 1024

/* un blocco di un file, con tutte le informazioni per scriverlo */
typedef struct {
    char buffer[DIM_BLOCCO];
    char nome[MAX_NOME];      /* nome del file (senza directory) */
    off_t dimensione;         /* dimensione totale del file */
    off_t offset;             /* posizione del blocco nel file */
    int lunghezza;            /* byte validi nel buffer */
} record;

/* stack condiviso con mutex e variabili condizione (stile monitor) */
typedef struct {
    record pila[CAPIENZA];
    int n;                    /* record presenti */
    int lettori_attivi;
    pthread_mutex_t mutex;
    pthread_cond_t non_pieno, non_vuoto;
} stack;

typedef struct {
    pthread_t tid;
    int numero;
    char *nome_file;
    stack *s;
} dati_lettore;

typedef struct {
    pthread_t tid;
    char *destinazione;
    stack *s;
} dati_scrittore;

void errore(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

void push(stack *s, record *r) {
    pthread_mutex_lock(&s->mutex);
    while (s->n == CAPIENZA)
        pthread_cond_wait(&s->non_pieno, &s->mutex);
    s->pila[s->n] = *r;
    s->n++;
    pthread_cond_signal(&s->non_vuoto);
    pthread_mutex_unlock(&s->mutex);
}
```

DUPLICA.C, RIGHE DA 1 A 80 (CONTINUA) (CONTINUA)

```
/* restituisce 0 quando lo stack è vuoto e tutti i lettori hanno finito */
int pop(stack *s, record *r) {
    pthread_mutex_lock(&s->mutex);
    while (s->n == 0 && s->lettori_attivi > 0)
        pthread_cond_wait(&s->non_vuoto, &s->mutex);
    if (s->n == 0) {
        pthread_mutex_unlock(&s->mutex);
        return 0;
    }
    s->n--;
    *r = s->pila[s->n];
    pthread_cond_signal(&s->non_pieno);
    pthread_mutex_unlock(&s->mutex);
    return 1;
}
```

Lo stack è un **monitor**: i dati (il vettore dei record e il contatore n) e gli strumenti di sincronizzazione stanno nella stessa struttura, e solo push e pop vi accedono. Uno stack si gestisce con un solo indice, perché si inserisce e si estrae dalla stessa estremità: i blocchi vengono quindi estratti in ordine diverso da quello di lettura, ed è il motivo per cui ogni record porta con sé il proprio offset. La pop restituisce 0 quando lo stack è vuoto e non ci sono più lettori attivi: è il segnale di fine per lo scrittore.

13.3 I lettori

DUPLICA.C, RIGHE DA 82 A 125

```
void *lettore(void *arg) {
    dati_lettore *d = arg;
    int fd;
    struct stat info;
    record r;
    ssize_t letti;

    if ((fd = open(d->nome_file, O_RDONLY)) < 0)
        errore(d->nome_file);
    if (fstat(fd, &info) < 0)
        errore("fstat");
    printf("[LETTORE%d] lettura del file '%s' di %lld byte\n", d->numero,
           d->nome_file, (long long)info.st_size);

    /* basename può modificare la stringa: si lavora su una copia */
    char copia[MAX_PERCORSO];
    strncpy(copia, d->nome_file, MAX_PERCORSO);
    copia[MAX_PERCORSO - 1] = '\0';
    strncpy(r.nome, basename(copia), MAX_NOME);
    r.nome[MAX_NOME - 1] = '\0';
    r.dimensione = info.st_size;
    r.offset = 0;
}
```

DUPLICA.C, RIGHE DA 82 A 125 (CONTINUA)

```

/* un record per ogni blocco; un file vuoto produce un record di 0 byte */
do {
    if ((letti = read(fd, r.buffer, DIM_BLOCCO)) < 0)
        errore("read");
    r.lunghezza = letti;
    if (letti > 0 || r.offset == 0) {
        printf("[LETTORE%d] blocco di offset %lld di %d byte\n", d->numero,
            (long long)r.offset, r.lunghezza);
        push(d->s, &r);
    }
    r.offset += letti;
} while (letti > 0);
close(fd);

/* fine del lettore: lo scrittore potrebbe aspettare solo questo evento */
pthread_mutex_lock(&d->s->mutex);
d->s->lettori_attivi--;
pthread_cond_signal(&d->s->non_vuoto);
pthread_mutex_unlock(&d->s->mutex);
return NULL;
}

```

Il lettore legge con `read` in un ciclo, preparando un record per ogni blocco; l'ultimo blocco può avere meno di 1024 byte, e un file vuoto produce comunque un record di lunghezza 0, perché lo scrittore deve creare anche la sua copia. Il nome del file da creare si ricava con `basename`, che può modificare la stringa ricevuta e viene quindi applicata a una copia. Alla fine il lettore decrementa `lettori_attivi` e risveglia lo scrittore, che potrebbe essere in attesa proprio di questo evento.

13.4 Lo scrittore e il main

DUPLICA.C, RIGHE DA 127 A 162

```

void *scrittore(void *arg) {
    dati_scrittore *d = arg;
    record r;
    char percorso[MAX_PERCORSO];
    int fd;

    /* la directory di destinazione viene creata se non esiste */
    if (mkdir(d->destinazione, 0755) < 0 && errno != EEXIST)
        errore(d->destinazione);

    while (pop(d->s, &r)) {
        snprintf(percorso, MAX_PERCORSO, "%s/%s", d->destinazione, r.nome);

        /* O_EXCL: la open riesce solo se il file non esiste ancora */
        fd = open(percorso, O_WRONLY | O_CREAT | O_EXCL, 0644);
        if (fd >= 0) {
            printf("[SCRITTORE] creazione del file '%s' di %lld byte\n", r.nome,
                (long long)r.dimensione);
            if (ftruncate(fd, r.dimensione) < 0) /* dimensione finale subito */
                errore("ftruncate");
        } else if (errno == EEXIST) {
            if ((fd = open(percorso, O_WRONLY)) < 0)
                errore(percorso);
        } else {

```

DUPLICA.C, RIGHE DA 127 A 162 (CONTINUA)

```

        errore(percorso);
    }

    /* scrittura del blocco nella sua posizione */
    if (pwrite(fd, r.buffer, r.lunghezza, r.offset) != r.lunghezza)
        errore("pwrite");
    printf("[SCRITTORE] scritto il blocco di offset %lld di %d byte su '%s'\n",
        (long long)r.offset, r.lunghezza, r.nome);
    close(fd);
}
return NULL;
}

```

Lo scrittore crea la directory con `mkdir`, ignorando l'errore `EEXIST`. Per ogni record apre il file di destinazione con `O_CREAT | O_EXCL`: se la open riesce il file è appena stato creato, e lo scrittore ne imposta subito la dimensione finale con `ftruncate`; se fallisce con `EEXIST` il file esiste già e viene aperto normalmente. Il blocco viene scritto con `pwrite` alla posizione indicata nel record, senza bisogno di `lseek`.

DUPLICA.C, RIGHE DA 164 A 204

```

int main(int argc, char *argv[]) {
    stack s;
    int n = argc - 2;
    dati_lettore *lettori;
    dati_scrittore scr;

    if (n < 1) {
        fprintf(stderr, "uso: %s <file 1> ... <file n> <directory>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    printf("[MAIN] duplicazione di %d file\n", n);

    s.n = 0;
    s.lettori_attivi = n;
    pthread_mutex_init(&s.mutex, NULL);
    pthread_cond_init(&s.non_pieno, NULL);
    pthread_cond_init(&s.non_vuoto, NULL);

    if ((lettori = malloc(n * sizeof(dati_lettore))) == NULL)
        errore("malloc");
    for (int i = 0; i < n; i++) {
        lettori[i].numero = i + 1;
        lettori[i].nome_file = argv[i + 1];
        lettori[i].s = &s;
        pthread_create(&lettori[i].tid, NULL, lettore, &lettori[i]);
    }
    scr.destinazione = argv[argc - 1];
    scr.s = &s;
    pthread_create(&scr.tid, NULL, scrittore, &scr);

    for (int i = 0; i < n; i++)
        pthread_join(lettori[i].tid, NULL);
    pthread_join(scr.tid, NULL);
    printf("[MAIN] duplicazione completata\n");
}

```

DUPLICA.C, RIGHE DA 164 A 204 (CONTINUA)

```
free(lettori);  
pthread_mutex_destroy(&s.mutex);  
pthread_cond_destroy(&s.non_pieno);  
pthread_cond_destroy(&s.non_vuoto);  
exit(EXIT_SUCCESS);  
}
```

DA RICORDARE

Perché è sufficiente la `pthread_cond_signal` sulla variabile `non_vuoto`, e non serve una `broadcast`?

Perché sulla variabile `non_vuoto` può attendere **un solo thread**, lo scrittore. La `signal` risveglia quindi proprio il thread interessato, sia quando arriva un nuovo record sia quando termina un lettore. Con più consumatori, invece, la fine dei lavori andrebbe comunicata con una `broadcast`, per risvegliarli tutti.

13.5 La verifica

Si esegue il programma su alcuni file di natura diversa (un file di testo, un eseguibile, un file vuoto, un file di dimensione non multipla di 1024) e si confronta ogni copia con l'originale usando il comando `cmp`, che non stampa nulla se i due file sono identici. Ripetendo la prova più volte, dopo aver cancellato la directory di destinazione, le copie devono risultare sempre identiche agli originali.

DA RICORDARE

Perché lo scrittore usa `pwrite` invece di `lseek` seguita da `write`?

Con `pwrite` posizionamento e scrittura avvengono in un'unica chiamata, che non modifica la posizione corrente del descrittore: il codice è più breve e resterebbe corretto anche se più thread scrivessero sullo stesso descrittore, caso in cui la coppia `lseek` e `write` potrebbe intrecciarsi con quella di un altro thread. Poiché lo scrittore apre il file a ogni record, anche `lseek` seguita da `write` funzionerebbe, ma con due chiamate di sistema invece di una.

DA RICORDARE

Che cosa succede se due file in ingresso hanno lo stesso nome in directory diverse?

I loro blocchi finirebbero nello **stesso file** di destinazione, perché lo scrittore individua il file solo dal nome: il risultato sarebbe un miscuglio dei due contenuti. Per evitarlo il programma può controllare all'avvio che i nomi siano tutti diversi, oppure aggiungere al nome di ogni copia il numero del lettore che l'ha prodotta.