Scheda di Lavoro – Compito

Esercizio 1 – safe_copy avanzato

Obiettivo: realizzare un programma C chiamato safe_copy che gestisce la copia sicura di file, usando perror(), struct, stdout/stderr e parametri da CLI.

Nota (Windows): anche se stdout e stderr appaiono entrambi nella console, sono due flussi distinti. Per verificarne la separazione, esegui il programma da cmd o PowerShell usando la redirezione:

```
safe_copy sorgente.txt destinazione.txt 2>err.log
```

Controlla:

- shell → contiene il resoconto "normale" (stdout)
- err.log → contiene solo i messaggi di errore (stderr) generati dal programma

Consegne:

1. Il programma deve essere eseguito come:
   `./safe_copy sorgente.txt destinazione.txt`
2. Definisci una struct CopyStats che contenga:
   - numero di byte copiati
   - numero di errori di lettura
   - numero di errori di scrittura
3. Apri il file sorgente in sola lettura e il file destinazione in scrittura.[1]
4. Se fopen() fallisce su uno dei due file, usa perror() e termina.
5. Copia il contenuto a blocchi di 256 byte.
6. Se fread() fallisce (fread < size richiesto e non per EOF), usa perror() e incrementa errori di lettura.
7. Se fwrite() fallisce, usa perror() e incrementa errori di scrittura.
8. Al termine:
   - stampa su stdout un resoconto della struct CopyStats
   - stampa su stderr eventuali anomalie non critiche
9. **Modifica i permessi del file sorgente per generare volutamente un errore e annotalo.**

Verifica:

• Riesci a distinguere cosa finisce su stdout e cosa su stderr?
• perror mostra messaggi diversi a seconda del tipo di errore?
• La struct viene popolata correttamente?

---

[1] Nota: i file vanno aperti in modalità **binaria**!

Esercizio 2 – Apertura ripetuta di file

Obiettivo: esplorare errori reali di sistema aprendo ripetutamente un file finché il sistema non rifiuta ulteriori aperture.

Consegne:

1. Scrivi un programma C chiamato openloop.
2. Crea una opportuna struttura dati dinamica che possa archiviare opportunamente, **per ogni apertura file**:
   - fp (File pointer)
   - indice progressivo di apertura del file
   - handle kernel del file [2]

   e una struct **OpenInfo** contenente:
   - numero totale di aperture riuscite
   - numero di aperture fallite
   - puntatore alla precedente lista
   - dimensione della precedente lista
3. In un ciclo while (true):
   - prova a fare fopen("test.txt", "r")
   - se ha successo, aggiungi il FILE* a un array dinamico
   - se fallisce:
     - usa perror("fopen")
     - incrementa il contatore di errori
     - esci dal ciclo
4. Chiudi *tutti* i file rimasti aperti.
5. Stampa un report finale su stdout.
6. Stampa su stderr eventuali condizioni anomale (es. impossibile chiudere un file).

Verifica:

- Quale errore di sistema viene segnalato da perror quando finiscono i file descriptor?
- Il programma rilascia correttamente tutte le risorse?
- Riesci a far provocare altri tipi di errore al SO?
- (Solo per Linux/MAC) Cosa cambia se riduci le risorse del processo con ulimit -n?

---

[2] **Ottenere l'handle del file:** FILE* è lo stream della libreria C. Se serve accedere al livello del sistema operativo, è possibile ottenere il file descriptor/handle con funzioni non portabili:

- **POSIX (Linux/macOS):** `int fd = fileno(fp);`
- **Windows (MSVC):**

```
int fd = _fileno(fp);              // file descriptor C
intptr_t h = _get_osfhandle(fd);   // HANDLE nativo Windows
```