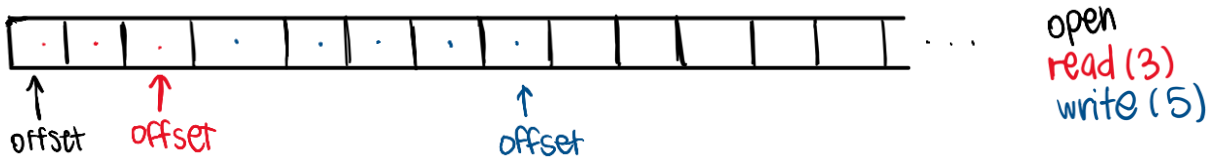
---------- seti 21/10 ----------

Per il sistema operativo un file è semplicemente una sequenza di byte, il sistema operativo non cerca in nessun modo di interpretare quei byte

se queste sequenze di byte rispettano un qualche formato di file -> esempio una jpeg, per essere un'immagine jpeg deve esserci un certo header, l'immagine dev'essere codificata in un certo modo ecc. -> se questa sequenza di byte rispetta quel formato, poi noi possiamo utilizzare un qualsiasi programma che legge quel formato e visualizzare l'immagine
spesso si usa un'estensione standard per etichettare il formato del file (esempio "nomefile".jpeg)

quando apriamo un file, il sistema operativo associa al file aperto un offset (puntatore)
ad esempio se abbiamo un file fatto di tantissimi byte, non appena facciamo la open, il suo offset sarà 0 -> non ho ancora letto né scritto niente
se gli dico "voglio leggere 3 byte", la read andrà a leggere i primi 3 byte del file, restituirà il contenuto dei primi 3 byte e sposterà l'offset di 3 posizioni
se facciamo un'altra read o una write nello stesso file descriptor, questa azione inizierà dall'offset, per poi spostarlo in avanti



se volessi leggere gli ultimi 10 byte del file, posso spostare l'offset e metterla in fondo al file, poi fare una read di 10

il fatto di spostare l'offset ha un costo costante, indipendente dalla lunghezza del file
in unix quasi tutto è un file; alcuni di questi file non sono file regolari, e quindi non si può spostare l'offset
in generale possiamo usare la system call **lseek** per spostare l'offset, non sempre funziona (in base al file)
la funzione lseek prende un file descriptor, un offset e una costante che gli dice rispetto a cosa indichiamo la nuova posizione (esempio: se scrivo 46, l'azione parte dal byte 46)

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

whence può essere: seek_set, seek_cur o seek_end

questo cursore può essere spostato anche "oltre" la fine del file -> se cerco di leggere fallisce, ma se ci scrivo funziona e quindi creo un file "con dei buchi" -> tutti i byte che ho saltato sono degli zero che non occupano spazio sul disco

i metadati di un file sono delle informazioni riguardo al file, ad esempio come si chiama, quanto è lungo, quando è stato modificato l'ultima volta, quando l'ho creato, ecc.
quasi tutti i metadati in un file system alla unix vengono memorizzati in una struttura dati che si chiama emode
ci sono delle system call (stat, ecc.) che ti permettono di leggere i metadati di un file, in particolare vanno ad inizializzare una struttura chiamata struct stat che dà le informazioni sul file

```
struct stat {
        dev_t     st_dev;         // major (12 bits) + minor (20 bits)
        ino_t     st_ino;         // Inode number
        mode_t    st_mode;        // File type and mode
        nlink_t   st_nlink;       // Number of hard links
        uid_t     st_uid;         // User ID of owner
        gid_t     st_gid;         // Group ID of owner
        dev_t     st_rdev;        // Device ID (if special file)
        off_t     st_size;        // Total size, in bytes
        blksize_t st_blksize;     // Block size for filesystem I/O
        blkcnt_t  st_blocks;      // Number of 512B blocks allocated
        struct timespec st_atim;  // Time of last access
        struct timespec st_mtim;  // Time of last modification
        struct timespec st_ctim;  // Time of last status change
}
```

esistono file system diversi, dove file system è il modo in cui vengono memorizzati i dati sul disco; ogni file system ha diversi limiti, come la lunghezza massima del nome del file, ecc.

- esistono delle costanti posix che si chiamano _posix_"qualcosa", che nonostante abbiano "**max**" nel nome, corrispondono alla **misura minima** che deve essere garantita
  esempio. _posix_name_max 14 dice che la lunghezza minima del nome è 14
- quindi se voglio sapere la lunghezza massima del nome di un file devo chiedere al sistema, tramite **getconf**

quando abbiamo un sorgente in c o c++ per poterlo eseguire devo compilarlo, perché i processori non sanno cos'è c, c++ perché non è codice macchina; quindi il compilatore prende il codice ad alto livello e lo traduce nel codice macchina della piattaforma target.
il linker è un **linker statico**, quando il linker (compilatore) mette insieme i pezzi, va a creare un file eseguibile che contiene tutto il codice necessario per l'esecuzione, quindi **codice + libreria**

ha alcuni svantaggi: uno svantaggio è che la maggior parte dei file dei miei programmi usa la libreria c, quindi quasi ogni eseguibile ha una sua copia delle funzioni della libreria, quindi occupa spazio su disco e ram
l'approccio più moderno usa quindi un **linking dinamico** -> le librerie non vengono copiate nell'eseguibile, ma vengono scritti dei metadati che dicono quali sono le dipendenze di quell'eseguibile

- i file eseguibili sono quindi più leggeri
- quando lancio l'eseguibile gli viene mappata la libreria c
- facilita l'aggiornamento
- l'unico svantaggio è che è problematico portare l'eseguibile da un sistema all'altro -> se uso una libreria che non è installata di default, non funzionerà l'eseguibile