Programmazione Avanzata e parallela

Lezione 9 e 11

Luca Manzoni

Memoria di massa

E costo delle operazioni su di essa

- Accesso ai files in C
  - Apertura di file
  - Lettura e scrittura con fscanf e fprintf
  - Lettura e scrittura con fread e fwrite
- Modello della memoria di massa
- Prestazioni come numero di operazioni di I/O

Files in C

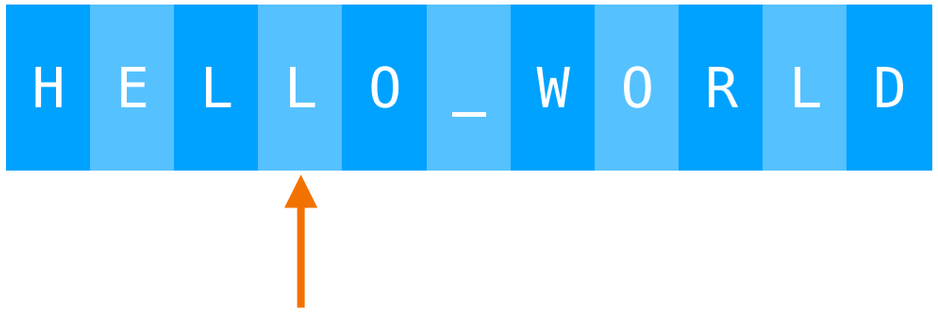
Modalità di accesso

- Quando vogliamo leggere un file il suo contenuto non viene direttamente caricato tutto in memoria
- Chiediamo al sistema operativo di fornirci una struttura che useremo per l'accesso
- È come avere una struttura che ci dice che file abbiamo aperto, il modo di accesso (e.g., per leggere o per scrivere) e il punto in cui siamo arrivati in lettura/scrittura del file
- Questo ci permette di accedere al contenuto del file come fosse un flusso (o stream) di caratteri/byte

Files in C

File come stream

Contenuto del file

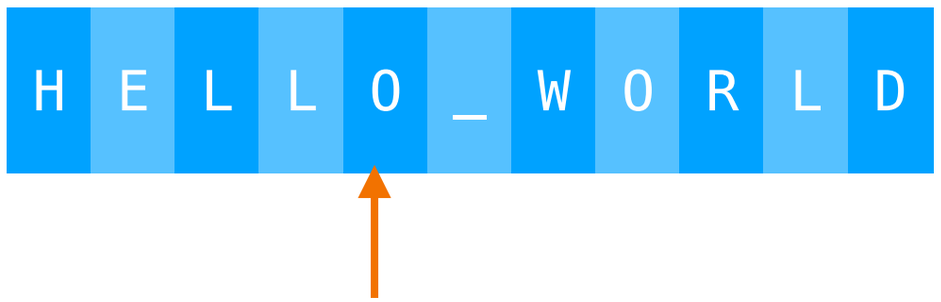


- Le operazioni di lettura e scrittura si riferiscono alla posizione corrente nel file

- Quando facciamo la prossima operazione di lettura leggeremo “O”…

- …e avanzeremo la posizione corrente a quella successiva

Contenuto del file



Valore letto: `O`

Files in C

Apertura dei file

Puntatore a FILE (non serve sapere come sia implementato)

Percorso del file da aprire.
E.g., “test.txt”

```
FILE * fopen(char * path, char * mode)
```

Modalità di accesso al file:

- “**r**” - lettura
- “**w**” - scrittura
- “**a**” - append (scrittura aggiungendo a fine file)
- “**r+**” o “**w+**” - lettura e scrittura
- “**rb**”, “**wb**”, “**ab**”, “**r+b**”, “**w+b**” - come prima ma considerando il file come binario e non di testo

Files in C

Chiudere un file

- Quando abbiamo finito di accedere ad un file è necessario “chiuderlo”
- Serve a liberare le risorse associate al file (la struttura puntata dal FILE *)…
- …e garantire che tutti i buffer associati (vedremo dopo cosa sono) siano svuotati
- Questo si fa con la funzione `fclose(FILE * fp)`

Files in C

Come leggere/scrivere

- Le funzioni usuali di lettura e scrittura (printf e scanf) hanno una versione che prende come argomento aggiuntivo che indica da dove leggere e scrivere:

- **fprintf(FILE * fp, char * format, …)**
  funzione come printf e, infatti, possiamo passare stdout come primo parametro a fprintf per ottenere lo stesso comportamento

- **fscanf(FILE * fp, char * format, …)**
  funziona come scanf, ritorna il numero di elementi letti oppure EOF se è terminato il contenuto del file

Files in C

Come leggere/scrivere

- Più a basso livello abbiamo **getc**() e **putc**() che lavorano a livello di un singolo carattere

- **getc** ha come argomento il file stream da cui leggere e ritorna un intero che è un cast a intero di un unsigned char oppure EOF

- **putc** ha come argomenti un intero (cast a intero di un unsigned char) che è il carattere da scrivere e il file stream su cui scrivere. Ritorna EOF in caso di errore in scrittura

- In entrambi i casi sono interfacce di basso livello (e richiedono una chiamata a funzione per ogni carattere letto o scritto)

Files in C

Come leggere/scrivere

- Per lavorare su blocchi di dati possiamo usare due funzioni (fread e fwrite) in cui passiamo un vettore e quanti caratteri scrivere (del vettore) o leggere (salvandoli nel vettore)

- Anche se putc e getc ci permettono di lavorare su dati binari, queste lavorano un char (solitamente corrispondete a byte) alla volta

- Possiamo invece riempire un intero vettore di int/float/etc con una singola fread…

- …o salvare il contenuto con fwrite

Files in C

fread

Puntatore a dove salvare i dati letti

Dimensione dei singoli elementi

```
int fread(void * ptr, size_t size, size_t nmemb, FILE * fp)
```

Numero di elementi letti (può essere inferiore se non vi è altro da leggere)

Numero di elementi da leggere

Puntatore a FILE da dove leggere

Nota: possiamo usare la funzione *feof(FILE * fp)* per stabilire se il motivo per cui abbiamo letto meno degli elementi richiesti è perché siamo arrivati alla fine del file

Files in C

fwrite

Puntatore da cui
leggere i dati da scrivere

Dimensione dei singoli
elementi

```
int fwrite(void * ptr, size_t size, size_t nmemb, FILE * fp)
```

Numero di elementi scritti
(può essere inferiore se ci
sono stati errori)

Numero di elementi
da scrivere

Puntatore a FILE
in cui scrivere

Files in C

Muoversi all’interno di un file

- Se però volessimo mantenere una struttura per cui non ci serve la contiguità dei dati?

- Esempio: un array ordinato per una ricerca binaria

- Dobbiamo ogni volta leggere tutto fino ad arrivare al punto giusto del file (i.e., accesso sequenziale)?

- No, possiamo posizionarci all’interno di un file usando **fseek**, che sposta la posizione di lettura/scrittura del file in un punto indicato da un offset

- **ftell** ci fornisce la posizione corrente all’interno del file

Files in C

fseek

Puntatore a FILE su cui eseguire lo spostamento

Di quanto spostarsi (può essere un valore negativo) *in bytes*

```
int fseek(FILE * fd, long offset, int whence)
```

0 se lo spostamento
è avvenuto correttamente

Una costante che indica rispetto a cosa sia l'offset:

- SEEK_SET l'inizio del file
- SEEK_CUR la posizione corrente
- SEEK_END la fine del file

OS e file

Cosa fa il sistema operativo

- Solitamente il sistema operativo non scrive immediatamente sulla memoria di massa, ma tiene in un buffer quello che c'è da scrivere, questo viene svuotato (scritto su disco) quando si riempie oppure quando è passato abbastanza tempo

- Per le letture il sistema solitamente non legge un byte alla volta ma un blocco alla volta, con accessi successivi allo stesso blocco che accedono direttamente alla memoria principale (questo caching è gestito dal sistema operativo)

- In linux (e in diversi altri sistemi) questa componente del sistema operativo si chiama *buffer cache* (https://tldp.org/LDP/sag/html/buffer-cache.html)

Modello con memoria esterna

Accedere alla memoria di massa

E costi

- Normalmente contiamo ogni operazione come con costo unitario quando calcoliamo il costo asintotico
- Abbiamo visto che già non è così quando accediamo alla memoria principale (ma l'approssimazione rimane utile)
- Questo è ancora meno vero quando dobbiamo accedere alla memoria di massa
- L'astrazione che usiamo deve essere modificata quando alcune operazioni sono migliaia o centinaia di migliaia di volte più lente

Accedere alla memoria di massa

E costi

100ns (0.0001ms)

Se l’accesso alla cache L1 fosse 1s invece di 2ns…

50 secondi

0.1ms

13 ore e 52 minuti

10ms

57 giorni, 20 ore e 52 minuti

Accedere alla memoria di massa

Modelli di costo

- I dati su cui dobbiamo lavorare hanno dimensione N elementi suddivisi in blocchi di dimensione B elementi

- Assumiamo che ogni blocco di dimensione B sia leggibile in tempo unitario (e se dobbiamo leggere di meno leggiamo comunque l'intero blocco)

- Nella memoria interna possiamo memorizzare al massimo M elementi, quindi possiamo tenere solo $\lfloor M/B \rfloor$ blocchi in memoria interna

- Contiamo solo il numero di letture e scritture di blocchi, ogni computazione intermedia ha costo zero

- Assumiamo $N \gg M \gg B$

Accedere alla memoria di massa

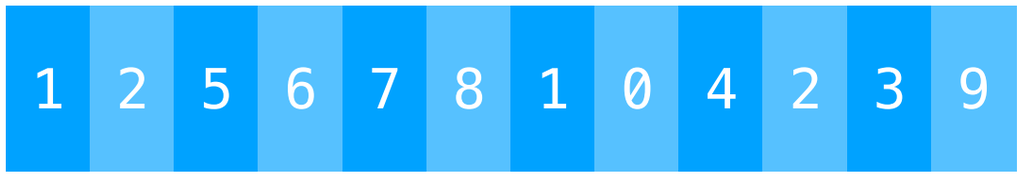
Modelli di costo

- Il nostro modello di costo conterà il numero di operazioni di input/output
- Notate che il modello può anche essere utilizzato considerando cache L1/L2/L3 come interne e memoria ram come esterna (i.e., è un modello generale)
- Per la memoria di massa solitamente B è da alcuni KB fino a un MB, M è di alcuni GB (ma può crescere fino ad alcuni TB)

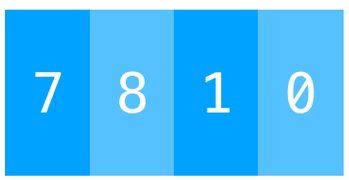
Array Scan

E costo

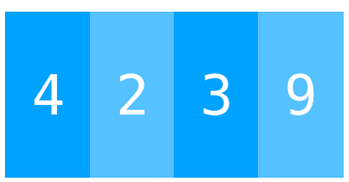
Dimensione dei blocchi



Blocco 1



Blocco 2



Blocco 3

- Consideriamo un file di dimensione N contenente dei valori interi
- Vogliamo sommare tutti i valori contenuti
- Leggendo a blocchi consecutivi di B bytes
- Dobbiamo effettuare $O(\lceil N/B \rceil)$ operazioni di I/O

Strutture ad albero

E fanout elevato

- Pensiamo di voler memorizzare un albero binario in memoria di massa
- Se contiene N elementi e ogni nodo è interamente contenuto in un blocco allora per accedere a una foglia necessitiamo di $\log_2 N$ operazioni
- Ma se un blocco potesse contenere k puntatori ai nodi figli (i.e., un albero k-ario?
- Allora ci basterebbero $\log_k N$ operazioni. Se $k = 1000$ allora per $N = 10^9$ si passa da 30 a 3 operazioni!