Laboratorio SO 21/05

Directory: a livello di implementazione è simile a un file. In quanto non è altro che un elenco di altri oggetti.

Supponiamo che la libreria stia usando il buffer di un processo. Tre thread/processi potrebbero usare lo stesso buffer e sovrascriverlo. In alcuni casi, alcune chiamate di sistema, per come sono fatte, non possono essere rese thread-safe. In questo caso si cerca la versione thread safe della libreria.

```
lstat: // se uso questo, vedo l'oggetto come directory, non c
ome link.
// Così creo un loop infinito di link simbolici.
// Un po' come hard link e soft link. Gli hard link non si po
ssono creare, e non
// avremo modo di distinguere se stiamo attraverso o no quell
a libreria.
// lstat mi dice che questo oggetto è un link simbolico, e qu
indi sta a me scegliere
/// se seguirlo o no, ma è una pessima idea perché si crea lo
op ricorsivo.
```

Per definizione, nei sistemi UNIX le directory “nascoste” vengono indicate con un punto iniziale.

. e .. sono directory fittizie, e fanno parte delle leggi di attraversamento.

Nota: ovviamente non ci sono comandi per modificare le directory, dato che per farlo basta modificare internamente gli oggetti che ne fanno parte.

Hard-link: usato strutture dati su disco chaimate i-node che contengono vari metadati e informazioni sull’oggetto e ci fanno capire dove stanno i vari pezzi del file stesso.

Impossibile fare collegamenti diretti fra directory sui sistemi UNIX.

Unlink: cancella riferimento. Ovviamente mi aspetto che l’altro riferimento continui a funzionare. Dentro l’i-node c’è un campo che indica il numero di hard-link.

Questi altri ovviamente devono continuare a funzionare.

Messa dentro l'i-node questa informazione anche se è ridondante. Potremmo scoprirlo analizzando tutto il sistema, ma ovviamente è più comodo e semplice averlo così. Anche se è ridondante, conviene. Per ragioni di efficienza. Scoprire quali sono tutti gli hardlink partendo solo dal numeretto e senza avere suggerimenti è estremamente difficile. Bisogna scansionare tutto il sistema.

remove: combinazione di rmdir e unlink.

symlink per creare link simbolico.

mv: sposta oggetti fra cartelle o anche fra file system diversi (dischi diversi).

Spostare l'oggetto: creo link simbolico verso l'oggetto già puntato, e poi elimino il primo link. Così ho rinominato l'oggetto. Naturalmente vale solo dentro lo stesso file system. L'i-node 100 di un file system è diverso dallo stesso di un altro file system. In questo modo possiamo spostare file che esistono, anche giganteschi, in un attimo. Utile invece di ricopiare tutto l'oggetto.

Se questo disco lo faccio su due partizioni dello stesso disco o su dischi diversi, quindi in sostanza su due file system diversi, allora non posso farlo.

Le chiamate di sistema non generano errori leggibili dall'umano, ma solo numeri.

Se cancello/sposto l'oggetto puntato dal symbolic link, se non lo aggiorno, allora si rompe il symbolic link. Con gli hard link non si rompono. Servono entrambi. [Da recuperare]

Storicamente, su molti SO della famiglia Windows non ci sono gli hard link. Aggiunti solo da poco.

truncate: può troncare un file normalmente a un certo punto, e può farlo anche allargandolo. I file system moderni evitano di sprecare spazio, e se un file ha una lunga sequenza di blocchi vuoti, quelli non vengono allocati. Sarebbe uno spreco. Quando il SO nota queste situazioni, risparmia spazio. Se creiamo un file con una dimensione da 1 TB, non creiamo veramente un file che occupa 1TB, dato che è pieno di blocchi vuoti, infatti, occupa pochi kb. Tecnica simile al copy on write, ma applicata ai file/file system.

mmap: manipolazione della tabella delle pagine, per cui certi blocchi rappresentino quel file, anche se quel file non è presente in RAM. Fatta la mappatura. E tecnicamente posso leggere a qualunque locazione del range

mappato, e questa cosa funzione anche in scrittura. Posso mappare e leggere/scrivere file anche se stanno su disco ma “mappati” in RAM. Anche se poi in RAM non sono presenti. Già visto nella teoria.

mmap: mappare una porzione lunga len byte, saltando un certo offset (non è detto che io voglia mappare l’intero file; vero è che non spreco nulla in più, ma magari è una contromisura contro sporcature involontarie), e chiediamo di mapparlo da qualche parte in RAM. Posso o dire io dove voglio che venga mappato, oppure lascio che il SO scelga da solo un’area libera per mappare l’oggetto. Dipende dal parametro *addr. Noi metteremo NULL come primo parametro, lasciando scelta al SO. Specificarlo poi è per cose più avanzate. Quando può servire? Se vogliamo materializzare il file all’interno di una mia struttura dati esistente. Es. lista linkata dove ogni nodo rappresenta blocco di un file/file intero, e chiedo al SO di mappare il file su quel segmento di memoria. Questo tipo di richiesta funziona solo se l’indirizzo è allineato alla dimensione della pagina. Il problema è che la mappatura viene gestita in termini di pagine, quindi deve coincidere con pagine vuote. Mi servono 6,5 pagine? Il SO me ne mappa 7, anche se la metà della pagina poi non viene effettivamente mappata. La richiesta potrebbe fallire se richiedo di mappare questo file in un punto che non corrisponda all’inizio di una pagina.

EXEC collegato al discorso delle librerie dinamiche. Queste contengono codice, e vengono portate virtualmente nello spazio di indirizzamento tramite la mappatura. Metodo efficiente per leggere questi oggetti. Per questo alcune porzioni mappate del file, oltre ad essere mappate in lettura, devono essere mappate in esecuzione.

Come passo il file? Devo passare il file descriptor. Quindi prima devo aprire il file, ovvero una open, e poi gli passo il fd per mapparlo.

flag: propone varianti sul modo in cui potrei effettuare questa lettura. Due modalità, mappatura privata e condivissa. La **MAP_SHARED:** se faccio mappatura in scrittura e specifico shared, sto chiedendo che voglio poter modificare il contenuto e voglio che le modifiche che faccio siano visibili anche agli altri. Vale anche per processi diversi. Modo per modificare in modo concorrente in un file. Modo per comunicare. Sui modelli windows i modelli di comunicazione a sistemi condivisi non esistono storicamente. Sui sistemi UNIX, invece, si può fare. Su windows si può ottenere semplicemente facendo mappare lo stesso file a entrambi i processi. Non significa che comunichiamo attraverso il disco, ma in

RAM. Sto facendo quello che viene fatto sui sistemi UNIX. Lavoro sulla stessa memoria fisica.

Se non è shared, è **PRIVATE:** significa che le modifiche che faccio restano private. Il processo vuole essere libero di sporcare quest'area di memoria, ma non vuole condividerle con altri. Non succede nulla su disco. È come se copiasse il file e lo modificasse. Quindi anche se non ho permesso in scrittura potrebbe farlo [?].

L'area **data** mi aspetto di poterla modificare, dato che è mappata da file. Sono le variabili inizializzate. Ma come faccio? Sto facendo una scrittura su quest'area di memoria, anche se non ho il diritto di scrivere su quel file. Stiamo infatti facendo una scrittura privata. Mappatura che viene fatta con i permessi in scrittura con la modalità privata.

Se avessi fatto mappatura shared avrei due problemi: 1) mi aspetto di avere i permessi di modifiche del file. E il fatto che io lo modifichi fa sì che al prossimo run il valore sarà diverso.

Ma la mappatura dove è avvenuta quindi? Attraverso il valore di ritorno ***void,** ci dirà dove è stato mappato il file. 1) per sapere se la mappatura è andata a buon fine, 2) per sapere dov'è il file. In effetti sennò non sarebbe chiaro dove leggere o scrivere.

Grazie a questo puntatore, con [i], posso accedere alla zona specifica del file.

Non posso farlo con **read** e **write?** Posso fare le stesse cose, ma dal punto di vista semantico, con la mappatura del file posso fare le stesse cose, ma in modo più conveniente. Non bisogna pensare a file piccoli da 3kb. Es. devo leggere il file, scrivere una parte, spostarla e così via, e quindi mi serve un buffer che non deve essere così piccolo. Mappando l'intero file, posso mappare dove ci capita, come ci interessa. Anche se è enorme. Se il file è enorme, di svariati GB, porterà in RAM solo pochi KB, ovvero quelli che andiamo a leggere.

E se non viene mappato? Ritorna **MAP_FAILED,** ovvero puntatore a NULL, un indirizzo che non verrà mai scelto dal sistema. Segmento 0 è dove inizia il segmento di testo. Perché potrebbe fallire la mappatura? Potrei non avere i permessi per leggere. Prerequisito della mappatura: dobbiamo sapere quanto è grande il file. **Esempio compito d'esame:** ci si chiede di leggere l'intero file con mmap. Per farlo, bisogna sapere quanto è grande il file. Se ho fatto la mappatura con un dato indirizzo e con una data lunghezza, possiamo fare tranqui

msync: se ho fatto qualcosa in modalità shared, mi aspetto che le modifiche poi me le ritrovo. Ma quand'è che queste modifiche vengono applicate? Subito? No. A meno che di mettere un flag che forzi le scritture, ma per le prestazioni non è una buona idea. Momento in cui viene sincronizzata è quando il processo termina. Se termina dopo giorni, resta lì per giorni. MDR non sa però se funziona esattamente così su linux.

Se faccio mappatura e scrittura con shared, sto chiedendo effettivamente di fare modifiche su disco.

Anche se leggiamo tutto il file, comunque viene letto man mano, quindi anche se il file è da 50 GB non avrò 50 GB in RAM.

Dal punto di vista delle prestazioni, tra mappare l'intero file e solo la parte che mi interessa cambia poco.

Un file portato in memoria non è come una stringa valida in C. Anche se in realtà la mappatura non è l'ideale per i file di testo in C, o per i file binari. Sappiamo quando fermarci perché troviamo un byte a 0. Se non lo troviamo la printf va avanti fino a leggere valori spazzatura oppure chissà dove va a finire e andrà in segmentation fault.

Per fare la mappatura, il file deve esistere. Non posso accodare contenuti ad un file tramite la mappatura. L'idea di allungarlo così è estremamente strano, difficile da fare. Questo perché la mappatura funziona su un file che esiste, non posso farlo su un file vuoto.

memcpy: estremamente efficiente con istruzioni a bassissimo livello.

Esercizio: facciamo reverse di un byte. Prendiamo il primo byte, e al suo posto gli metto l'ultimo, scambiandoli, e così via. Dovrei usare due buffer, quando iniziano a riempirsi li sposto. Deve funzionare su file arbitrariamente grandi. Vedi mmap-reverse.c

Con la mappatura è estremamente semplice, con altre funzioni a cui siamo abituati è molto più complesso invece.

Fork fallisce: magari l'amministratore ha posto un limite sul numero di processi che posso creare. Fare fork bomb (ciclicamente faccio fork) non è permesso. In generale, manderebbe in crash il sistema.

fork.c: gestito con la pausa di 8 secondi, a volte il padre altre volte il figlio.

In generale, una buona abitudine può essere quella di mettere una exit alla fine del blocco, per capire esattamente cosa fa il figlio.

Nell'esempio, conn ./fork slow-child sto chiedendo al padre di uscire subito, e al figlio di uscire dopo.

In ogni sistema, c'è un adattatore predefinito. Solitamente è il processo init.

A volte systemd, ovvero 4427. Alcune distro usano systemd altre no. Fedora non usa systemd

Vedi fork-buffer-glitch

Se non metto il ritorno a capo, "sono il padre e sono appena partito... (*)" viene stampato due volte. Quando il padre ha fatto printf l oha fatto sullo stdout, uno stream. È bufferizzato.

Dato che non c'è il ritorno a capo, rimane nel buffer del padre. Quando faccio la fork, viene copiato nel figlio. E per questo lo vedo due volte. Una volta per il padre, e l'altra per il figlio, quando il buffer viene svuotato.

Durante gli esami non usiamo la fork, quindi non dovremmo ritrovarci in questo tipo di problemi.

Per evitarlo, potremmo anche fare setbuf(stdout, NULL)

Su multi-fork: il fatto che io veda interlacciamenti solo fra figlio1 e figlio2 e non con il padre è normale. Il padre fa compiti molto semplici, e ancora i figli, nonostante la fork sia stata fatta, devono entrare a regime. Su un sistema con una sola CPU, il tipo di interlacciamento potrebbe essere diverso.

Se redirigo l'output su file, non sono più interlacciati, ma perfettamente serializzati. Perchè? Quando la shell ha predisposto la stdout a un file, il meccanismo del buffer è quello del fully buffer. Cioè non è un terminale che è min-buffered, ma fully. Sia il padre che i figli hanno lavorato su uno stdout che era fully buffered, e poi gli output su file si sono riversati solo quando i singoli buffer si sono riempiti. Per questo non si sono miscelati. Perché prima di riversare l'output su file, devono prima riempirsi i singoli buffer.