FILE E PROCESSI

Un processo non può interagire direttamente con l'hardware ma per farlo deve usare una system call (funzione wrapper del C). Queste funzioni prendono dei determinati parametri e poi a seconda della convenzione **del S.O.** preparano i registri ed effettuano la **system call**. Le convenzioni di chiamata possono essere diverse però:

- **I sistemi** assegnano un numero che le identifica e poi utilizzano dei **registri** per passare i parametri.
- Si esegue un'istruzione speciale che genera una **trap**(eccezione) a livello di processore e fa si che l'esecuzione passi a **modalità kernel.**
- Il kernel esegue ciò che deve fare
- Si ritorna in modalità utente

Una **system call può fallire** e in questo caso fa un semplice return **-1**. In caso di successo ritornerà un valore maggiore o uguale a 0.

Per stampare un messaggio d'errore leggibile dall'utente si può usare **perror**, che stampa su stderr un messaggio personalizzato seguito dalla descrizione dell'errore corrispondente a errno.
In alternativa, si può usare **strerror_r** che traduce da int a string l'errore con una frase decisa da noi in precedenza. **L'_r** indica che la funzione è rientrante ovvero safe per i thread.

Tutto l'input/output avviene tramite **file descriptors**, quindi le system call che scrivono qualcosa da qualche parte, sapranno dove scrivere perché glielo comunicheremo tramite file descriptor. Se io voglio scrivere qualcosa, ci sono tre file speciali per convenzione:

- **0 :** standart input
- **1 :** standard output
- **2 :** standard error

Se io voglio scrivere in un file per prima cosa devo aprire il file, per farlo uso la **system call open.**
Il **primo parametro** della open è il pathname, è una stringa che rappresenta il percorso del file che vogliamo leggere o scrivere. Il **secondo parametro** è **flag** che mi indica cosa voglio fare di questo file, ovvero se lo voglio aprire in lettura o scrittura. Questo parametro è un intero ed è chiamato **bitmask** ovvero non è

importante il valore completo dei bit ma i singoli insiemi di bit in quanto mi possono indicare qualcosa. Se flag specifica di voler creare un file nuovo allora ci serve un **terzo parametro** che rappresenta i **permessi di file.**

Sotto **unix** ogni file ha i permessi relativi a tre categorie di utente:

- Il proprietario del file
- Il gruppo a cui appartiene
- Tutti gli altri utenti del sistema

Per ognuno di questi insiemi di utenti, può specificare 3 bit che sono **r** (lettura), **w** (scrittura) e **x** (esecuzione). Questi permessi sono spesso codificati in ottale.
Per cambiare i permessi esiste il comando **chmod**, mentre per cambiare proprietario **chown** (ma solo se si è in root). **Root** (amministratore di sistema) può leggere, scrivere ed eseguire (purché ci sia almeno un bit x settato) qualsiasi file, non si fanno controlli sui permessi.

Per leggere o scrivere si usano le system call **Read** e **Write.** Il primo argomento è **fd** ovvero il **file descriptor** che dice su cosa voglio leggere o scrivere. Il parametro **buf** mi rappresenta l'indirizzo del buffer e infine il parametro **count** è il numero di byte da leggere o scrivere.

Per il sistema operativo **un file è semplicemente una sequenza di byte.** Il S.O. non cerca in nessun modo di interpretare quei byte. Un **file regolare** è un tipo di file che contiene semplicemente dei byte uno dietro l'altro. **Possiamo pensarlo come un grande array.**

Ad ogni file aperto è associato un file **offset/pointer** che indica in che posizione leggere / scrivere all'interno del file.

Quando apriamo il file **l'offset** parte da 0, se leggiamo 3 byte allora l'offset si sposta di 3.

In Unix tutto è un file, anche le connessioni di rete o tastiera. Questi però non sono file regolari ma **stream,** ovvero flussi di dati che arrivano nel tempo. Se stai leggendo da una **connessione di rete**, non puoi dire "voglio tornare indietro di 10 byte", perché **non puoi tornare indietro nel tempo**

Come facciamo a spostare l'offset? Con la system call **lseek.** Abbiamo 3 parametri:

1. **Fd:** file descriptor
2. **Offset:** di quanto ci vogliamo spostare
3. **Whence:** dove partiamo

Whence può essere:

- **SEEK_SET:** inizio del file
- **SEEK_CUR:** posizione corrente
- **SEEK_END:** fine del file

Volendo questo cursore può essere spostato anche "oltre" la fine del file. Se cerco di leggere fallisce ma se ci scrivo creo soltanto un file con dei "buchi" che poi verranno riempiti con degli zero.

I metadati sono le informazioni sul file ad esempio: nome, dimensione, permessi ecc. Nei **sistemi Unix** queste info si trovano in una struttura chiamata **inode.**

Come ottengo i metadati? Mediante l'uso di tre system call: **stat, lstat** e **fstat.** Tutte riempiono una struttura stat con i dati del file.

Stat e **lstat** prendono come parametro il percorso sottoforma di stringa, ma la seconda non segue i **link simbolici**, ossia un tipo di file il cui contenuto punta a qualcos'altro. **Fstat** al posto di prendere un percorso come parametro, prende un file descriptor, quindi, prima si deve aprire il file e poi chiedere i dati di quel file.

Nella **struct stat** possiamo trovare:

- **UID** del proprietario del file
- **Timestap** ultima modifica del file
- Tipo di file
- Dimensione del file

Che cosa sono i programmi binari?
Quando abbiamo un sorgente in **C** o **C++,** per poterlo eseguire devo compilarlo; questo perché i processori non hanno idea di cosa sia C, C++. Il linguaggio che parla il processore **è il codice macchina**, e ogni processore ha il suo codice macchina. Quindi il **compilatore** prende il nostro codice ad alto livello e lo traduce nel codice macchina della piattaforma target.

- **Compilatori e assemblatori** sono quelli che producono gli **object file**, detti anche file oggetto o file rilocabili: sono essenzialmente del codice macchina con dei **buchi** e con dei **metadati.**
- Ld, il **link editor** (linker statico), mette insieme file oggetto e librerie, eseguendo rilocazione e risoluzione dei simboli

Quando il **linker** mette insieme i pezzi, crea un **file eseguibile** dove tutto il codice necessario per l'esecuzione è dentro a quel file. Quindi il file conterrà il **nostro codice** e **parti della libreria C** che servono per l'esecuzione del nostro codice.

Ha alcuni grossi **svantaggi**: uno svantaggio è che maggior parte dei file e dei programmi usa la libreria C, quindi quasi ogni eseguibile ha la sua copia delle funzioni principali in C, e questo occupa spazio disco e di ram.

L'approccio moderno è il **linking dinamico** dove NON si copiano le librerire ma si scrivono solo dei metadati che dicono di quali librerie il programma ha bisogno. **L'unico svantaggio** è che se un programma dipende da una libreria che non è presente sul computer dove lo lanciamo allora non funzionerà.

Dentro ad un eseguibile abbiamo:

- **Codice macchina** corrispondente al codice sorgente che abbiamo compilato, finisce in una sezione chiamata .txt
- I dati che finiscono nella sezione .data
- Metadati

Il formato degli eseguibili si chiama **ELF**. Le sezioni elencate prima si chiamano **segmenti.**

Un file **ELF** ha un header. Conterrà il codice, i dati ecc. Se voglio mandarlo in esecuzione dovrò mappare in memoria il codice e i dati.
Il sistema operativo però non lo copia semplicemente tutto nella **RAM**, ma mappa le diverse parti del file su specifiche aree di memoria chiamate **pagine.** Il codice del programma, che non cambia mai durante l'esecuzione, viene caricato in pagine di memoria protette in solo lettura mentre I dati sono caricati sia in lettura che scrittura.

Per fare girare un programma servono anche:

- Stack
- Heap
- Kernel

PID = identificatore del processo chiamante.
PPID = parent pid

Tutti i processi vengono creati con una system call chiamata **fork.** Abbiamo una gerarchia ad albero dove ogni processo viene creato a sua volta da un altro processo. Il punto iniziale è **init** e ha pid numero 1. Init è l'unico processo che non posso creare con fork.

Il kernel per esporre le informazioni utilizza uno pseudo file system **/proc.** Quando leggiamo da questo file in realà stiamo chiedendo al kernel di darci l'informazione rispetto al processo.

Ogni processo ha due directory:

- **Directory root:** viene usata ogni volta che usiamo un percorso assoluto (con /)
- **Directory di lavoro:** usata per i percorsi relativi

Le system call per gestire i processi sono 4:

- **Fork:** per creare un processo
- **_exit:** termina processo chiamante
- **Wait:** aspetta la terminazione di un processo figlio
- **Exceve:** esegue un nuovo programma nel processo chiamante. Sostiuisce completamente lo spazio di indirizzamento del processo che lo invoca. Infatti exceve viene usata dopo la fork.

Pid_T Fork(void) = Clona un processo, crea un processo figlio che è una copia del processo che ha evocato la fork. **Ritorna due volte:** quando la chiamata ha successo clona il processo e in entrambi i processi la fork ritorna. L'unica cosa che cambia è il valore di ritorno. Il figlio del processo ritorna 0 e il processo padre ritorna il **PID** del figlio.

void exit(int status) = librearia del C. Questa funzione prima di chiamare **_exit** chiama tutte le funzioni che sono state registrate con **atexit** e **on_exit** , svuota i buffer di **I/O** ed elimina i file temporanei creati con **tmpfile.**

In tutti i casi vengono chiuse e rilasciate le risorse del processo.

Pid_T wait(int wstatus) = aspetta la terminazione di un processo figlio. Se la wait va a buon fine, restituiscei il PID del figlio che ha terminato.

Redirezione input/output

Quando apriamo un file il SO restituisce un file descriptor ossia un numero intero non negativo che corrisponde al file aperto. Questo numero usato dalla system call

(del programma in esecuzione) viene usato come indice all'interno della tabella del file descriptor di quel processo. Fd0 è input, fd1 è ouput e fd2 è error.
Questa tabella è formata:

- Da una colonna che rappresenta i flag di quel file descriptor
- Un puntatore ad un altra struttura dati che rappresenta i file aperti del sistema(open file table). Questa struttura dati ha un offset, altri flag e un puntatore ad un i-node (ossia una struttura dati che corrisponde ad un file all'interno del file system, e contiene i vari metadati dei file, tranne il nome perchè si trova nella directory, dove si trova l'associazione tra nome e numero di i-node).

Quando si apre un file alla open si passa una stringa che è il percorso per raggiungere il file e a quel file corrisponderà un i-node.

Esempio: Supponiamo che il file pippo sia **l'inode 224.** Se un processo apre il file che corrisponde **all'inode 224** , l'inode sarà portato in memoria e verrà creata una entry nella tabella open file table. L'offset sarà 0 e i flag specificano come voglio aprire il file (read, write o entrambi). Viene quindi creata un entry nella tabella del file descriptor e verrà restituito quel file descriptor al processo (open ritorna 3, per esempio). Se un altro processo fa una open dello stesso file con offset o flag diversi , anche se l'inode è uguale, avremo un'altra entry in open file table, che corrisponderanno a due file descriptor diversi (esempio: fd 3 read only, fd 10 write only). Se chiudo uno dei file descriptor , per esempio fd 3, perdo l'entry nella file descriptor table e la entry nella open file table ma non perdo 224 ossia pippo perchè è ancora puntato dall'altro fd, nell'esempio 10. Se chiudo anche 10 allora verrà perso anche il file.

Un altra possibilità è avere due fd diversi che fanno riferimento alla stessa entry della open file table. Per quel processo non avrà nessuna differenza usarne uno o l'altro. Questo succede con la system call DUP , che restituisce un altro file descriptor che però punta allo stesso file del fd duplicato.

Un ultimo caso è quello dove entrambi i fd table di due processi diversi puntano alla stessa entry nella open file table. Questo succede perchè probabilmente il secondo processo è stato creato tramite la fork del primo processo, quindi, eseguono le stesse cose ma dopo la fork ognuna ha continuato diversamente in maniera indipendente.