

man 2 syscall

registro del
21-10-15 (h. 1:06)

man 3 commodity

cat /usr/include/x86_64-linux-gnu/
asm/unist_64.h

per vedere i `--NR_write`
ovvero i numeri delle syscall (e nomi).

Suddivisione Syscall

- gestione PROCESSI
- MEMORIA
- gestione DEVICE
- " FILE
- " FILE SYSTEM
- " TEMPO
- " UTENTI
- " RETE
- Inter Process Communication (ipc)
- debug
- miscellanea

Il processo "da solo" può
accedere alla sua memoria e
fare operazioni di CPU. accedere un
file richiede il dialogo con il Kernel.

pausa 1:15:31-1:23:20

curiosità sul manuale archivio delle "vecchie
syscall" del 1970 "sempre quelle".

Un concetto geniale che c'è dall'inizio dei tempi di UNIX e quello di separare concettualmente le due funzionalità:

- CREAZIONE di un PROCESSO ed
 - ESECUZIONE di PROGRAMMA
- in UNIX sono 2 operazioni DISTINTE

La creazione di un processo crea il processo ma non esegue il programma. Lo stesso processo continua a funzionare come dire... quasi in modo SCHIZZOFRENICO prende due personalità allo stesso tempo, invece l'esecuzione del programma non crea un nuovo processo ma fa sempre in modo che il processo corrente inizi ad eseguire il programma che vogliamo.

FORK

```
scrivete: // mystery.c
#include <stdio.h>
#include <unistd.h>
```

questo programma
può scrivere
tutte e due?

☐ Sì ☐ no

```
int main(...) {
    if (something) ← qui se sostituisco
        printf("something is true\n");
    else
        printf("something is false\n");
}
```

Fork()
a something
scrive
ENTRAMBE!

cosa fa `FORK()`?

1:31:00

Crea un processo figlio a immagine e somiglianza del padre (è biblico ☺)
l'unica cosa che cambia

dal momento della `FORK()`
il processo diventa due processi

stanno eseguendo lo stesso programma
hanno entrambi lo stesso program counter
come una divisione cellulare

Il processo genitore continua a esistere ma viene generato il figlio
al processo genitore restituisce "nullo"
l'identificativo del figlio
al processo figlio restituisce 0. ☺

AGGIUNGIAMO `GETPID()`

1:33:00

`printf("%d something is ... \n", getpid());`

`getpid()` id processo corrente


`pid_t` tipo degli id.

`getppid()` "con doppia P" id del padre

il figlio apprende l'identità del padre chiedendo `getppid()`.

aggiungiamo sleep(1) all'interno del codice del figlio e notiamo che il padre termina e dà il controllo alla shell (comporre il prompt e dopo 1 secondo sono comparsi l'output del figlio. quindi il figlio comincia ad avere vita autonoma ognuno terminerà quando la logica del programma termina.

la shell stava eseguendo il padre quindi attendeva che il padre terminasse.

quindi se scriviamo
a.out ; vi lo mette "sul terminale corrente" 
"padre aspetta che termini il figlio"
la syscall per fare l'attesa è
man 2 wait

wait attende un processo qualsiasi e
ritorna il processo terminato.

waitpid voglio aspettare "proprio quel processo lì" (specificando il pid).
(se ne termina un altro
rimane in attesa se pid = -1
fa lo stesso effetto di wait)

cosa è quel *WSTATUS?

valore finale del processo.

attenzione quell'intero il cui indirizzo
andremo a mettere come parametro di wait
non è il valore di ritorno, ma è una
mascara di bit che lo contiene assieme ad
 tante altre informazioni.

i processi non muiono mai per morte naturale, o per omicidio o per suicidio.

```
_exit(42); WEXITSTATUS(ws)
```

↑ nel manuale di wait.

```
if (WIFSIGNALED(ws))  
    printf("Killed %d\n", WTERMSIG(ws));  
else  
    printf("wait ok %d\n", WEXITSTATUS(ws));
```

ora stampa wait ok 42, quindi
fa come prima ma posso
aggiungere "int *boom = (void *) 42;
e poi " *Kaboom = 0;

e ora stampa Killed 11

man 7 signal

```
grep -r SIGSEGV /usr/include
```

↑
signal for segmentation violation

con Kaboom è un suicidio colposo,
per errore, non volontario.
con _exit è un suicidio volontario.

le systemcall segnalano un errore
ritornando -1. esiste una pseudo-varabile
chiamata ERRORNO che specifica quale
errore è avvenuto. si usa come
una variabile globale ma "è legata al
thread" (e due finiscono in contemp. con
un errore si vuole vedere solo il proprio).

include <errno.h>
existe strerror(errno)
→ man strerror → ci vuole string.h.
che traduce il codice d'errore
(errno significa error number quindi
è chiaramente un intero...
in una scritta comprensibile.

Avete presente che ci sono delle
scritte standard quando usate UNIX
o Linux. per esempio

ccat dCxbDFlgJK

→ bash: ccat: command not found

cat dCxbDFlgJK

→ cat: dCxbDFlgJK: No such file or directory

"perché tutti i programmi mi mandano
la stessa scritta nello stesso modo così
poco fantasiosi i programmatori"

NO perché ci sono le traslitterazioni
standard di tutti gli errori.

che tra l'altro (AHI NO!)
vengono talvolta tradotti nelle
lingue nazionali.

la -exit potrebbe non avere
successo? basta chiederlo al
manuale: (scrive il comando
man -exit e poi digita /RETURN).

These function do not return.

Note:

- per ora siamo capaci di creare processi ma non di eseguire programmi
- fork al momento del `Fork()` crea due processi CON MEMORIA PRIVATA! Quindi al momento del `fork()` se avete una variabile globale il processo genitore e il processo figlio hanno il valore uguale ma non è in memoria comune è memoria privata quindi se dopo avvengono modifiche non vengono viste dall'altro, la memoria viene "replicata" per tutti e due. Così possono vivere di vita propria e se vogliono delle strutture dati CONDIVISE lo devono chiedere esplicitamente ed entreranno in ballo tutti i problemi (di CONCORRENZA) che vedremo giovedì.

Che differenza c'è tra processo e thread?

2021-10-22

"La differenza c'è solo di venerdì (laboratorio), il giovedì no." Tecnicamente processo diventa il titolare dei diritti sulle risorse. Questo processo può avere uno o più fili esecutivi thread, esecuzioni concorrenti che condividono le risorse in particolare la memoria.

Quando si parla di programmazione multithreading si usano i pthread che sono previsti da POSIX e le varie parti del programma vedono la stessa memoria quindi se nominano la stessa variabile condivisa o con un puntatore fanno accesso a una variabile, è la stessa variabile per tutti e due. Se invece lei ha più PROCESSI magari ognuno con un thread solo (Classicamente all'inizio non c'era multithreading quindi processo e thread coincidevano come concetto) può utilizzarli per risolvere un problema insieme? Sì, però deve stare attento che ognuno di questi ha la sua memoria, quindi non può nominare un puntatore e dire dall'altra parte "vai a prendere quel puntatore". Quel puntatore nella memoria dell'altro processo non ha alcun senso, le risorse sono separate. Come collaboreranno? Con altri metodi, per esempio utilizzando tecniche di MESSAGE PASSING. Si manderanno dei messaggi per dire "questo è il dato che devi elaborare" e l'altro processo prenderà il messaggio e continuerà l'elaborazione.

Quindi
PROCESSI \Rightarrow MEMORIA PRIVATA
THREAD \Rightarrow MEMORIA CONDIVISA

È ovvio che poi potete avere la combinazione dei due ovvero un PROGRAMMA che utilizza PIÙ PROCESSI, che parlano tra loro con MESSAGE PASSING, ognuno dei quali è MULTITHREADING, quindi ha all'interno fili esecutivi che condividono la memoria all'interno di ciascun processo.

Perché solo di venerdì? Perché nella teoria di programmazione concorrente si dice se il modello è a memoria privata o a memoria condivisa e poi chiamiamo tutti processi. Quindi il giovedì i thread non contano perché li chiamiamo processi in ambiente - nel modello a memoria condivisa.

STRACE

vi fa vedere
tutte le systemcall
che il processo
ha chiamato.

cp ../20210930/hello.c .

vi hello.c

gcc -static -o hello hello.c

strace ./hello

strace ./hello

↳ ho chiamato `execve`

↳ ho chiamato `brk` (la vedremo) serve per chiedere memoria per l'esecuzione. Questa domanda `brk(0x ...)` chiede memoria per lo HEAP.

↳ poi guarda varie cose

↳ vedete io nel codice ho scritto `printf`.

`printf("Hello world\n");`

ma qui la systemcall chiamata è `write`, e cosa scrive?

`< write(1, "Hello world\n", 12Hello world) = 12 >`

scrive sul file 1 la stringa "Hello world\n" (`\n = 2` `caratteri`) per la lunghezza complessiva di 12 caratteri ecco questo 12 con attaccato Hello è perché quello che segue non l'ha scritta `strace`, questo è il vero output del programma, infatti vedete che compare dopo l'ultimo argomento della `write` e

prima della parentesi tonda.

Strace prima ha scritto
<write(1, "Hello world\n", 12)>
poi ha mandato la systemcall
e poi ha atteso il valore di risposta
che è 12. La write come valore di
risposta restituisce il numero di caratteri
che ha effettivamente scritto. Uno dice
lo sai già glieli hai messi come parametro
Non è detto! Perché se avete una qualche
struttura limitata e avete riempito completamente
la struttura potete chiedere di scrivere 12 e
vi risponde 5 perché dopo il 5° non ci
stava più.

Quindi vi INVITO a provare strace perché
vi fa vedere come l'esecuzione dei vostri
programmi si è comportata nei confronti
del Kernel, quali sono i servizi che ha
chiesto al Kernel. Fa vedere il dialogo che
c'è tra il programma applicativo e il
sistema operativo. "Sono le righe in
corsivo del cuoco".

Perché ho messo -STATIC ?

Se io lo compilo senza -static funziona
lo stesso ma in realtà "temevo peggio"
fa un sacco di altre cose perché alla
partenza in verità non parte subito
il vostro programma ma parte il
LINKER DINAMICO che cerca tutti i
pezzi che servono per l'esecuzione e
poi esegue codice.

man exec \longrightarrow EXEC(3)

esistono molte exec ma sono tutte del manuale 3, una sola è la syscall che veramente fa l'exec che è la EXECVE.

man execve \longrightarrow EXECVE(2)

infatti vedete che execve è nel manuale 2. (Il manuale 2 contiene syscall, il manuale 3 contiene "utility" ovvero funzioni che in pochi minuti potremmo scrivere anche noi ma servono "per sfruttare anni di esperienza per quanto riguarda efficienza, sono ottimizzate e standard").

Tutte le altre (man 3) fanno delle trasformazioni per rendere più AGEVOLE la chiamata, ma sono solo funzioni di comodo che chiamano poi la execve.

proviamo execve $\left(\begin{array}{l} \text{const char *pathname,} \\ \text{char *const argv[],} \\ \text{char *const envp[]} \end{array} \right)$ 0:16:20

[pathname	path del File da eseguire
		argomenti \rightarrow (argc, argv)...
		ambiente \rightarrow printenv

% printenv
% File hello

(prima exec
poi hello)

abbiamo cambiato il programma in esecuzione ma non è cambiato il processo che li ospita.

% get env

se facciamo
execve (...); poi un'altra
funzione se execve ha successo
questa funzione non verrà MAI
eseguita

man exec

Come si distinguono le varie exec?
tramite i suffissi, basta conoscere
il formato dei suffissi e si sanno tutte.

exec1	execv	var globale
exec1p	execvp	extern char ** environ
execle	execvpe	

L \Rightarrow sta per long format. invece di mettere
char *newargv[] = { str 1, str 2, ..., NULL };
execve(path, newargv, env);
si può fare
execle(path, str 1, str 2, ..., NULL, env);

V \Rightarrow vettore ("scomodo rispetto a L")

P \Rightarrow invece di specificare il path file
specifico solo il file cercando
come path (variabile di env) delle syscall.
esempio: (per eseguire cat)
execve("/usr/bin/cat", ..., ...)
exec_p("cat", ..., ...)

E \Rightarrow environment

0:55:20

con `perror("exec");`
utilizziamo gli errori standard.

possiamo combinare la `Fork()`
con `exec(...)`, ad esempio
facendo in modo che il
padre aspetti che il figlio faccia
l'`exec` e fare cose dopo.

Noi possiamo creare processi solo con
`fork`. L'unico processo iniziale
è quello avviato dal Kernel `init`
che se termina fa shutdown il
sistema.

La stringa dentro `perror("...")`
è quella che mette prima dei :
per vedere dove è successo il guasto.

```
# /usr/bin/env python 3
```

```
which cat → /bin/cat
```