---------- seti 24/10 ----------

cosa contiene un eseguibile?

dentro ad un eseguibile ci sono

- il codice macchina corrispondente al codice sorgente che abbiamo compilato -> che finisce in una sezione chiamata .text
- i dati, variabili globali inizializzate, variabili globali non inizializzate e la memoria allocata per loro -> finiscono in .data e .rodata (read only data, quindi costanti che non possono essere modificate)
- metadati che dicono al sistema l'architettura per cui l'eseguibile è stato compilato

queste sezioni vengono raggruppate in **segmenti**

il formato degli eseguibili è elf (executable and linkable format)

come è fatto un file elf?

un file elf ha un'intestazione, un header (che dice l'architettura, l'entrypoint, ecc), conterrà il codice, i dati e altro. se voglio mandarlo in esecuzione, dovrò mappare in memoria il codice e i dati -> il sistema operativo terrà traccia del fatto che ci saranno una serie di pagine in memoria che corrispondono a parti di questo file. come con i dati -> mentre un programma gira i dati cambiano, mentre il codice no -> quindi la parte di codice può essere mappata in modalità solo lettura in memoria fisica; la parte dei data non può essere mappata in read only

per far girare un programma ci serve sicuramente mappare:

- codice e dati
- **codice e dati della libreria da cui dipende**
- serve inoltre un'area heap, che contiene la memoria dinamica (che non è scritta nell'elf)
- ci deve essere un'area di memoria stack che cresce dinamicamente man mano che si chiama la funzione ricorsivamente
- il kernel, non accessibile in modalità utente ma utile per alcune chiamate di funzione

ci sono delle system call che ci dicono qual è il pid (identificatore del processo chiamante):

- **`pid_t getpid(void)`**
- **`pid_t getppid(void)`** -> pid del padre

Tutti i processi vengono creati con una system call chiamata fork
C'è una gerarchia ad albero, dove ogni processo viene creato a sua volta da un altro processo ecc. -> il punto iniziale è **init** e ha il PID numero 1 (su Ubuntu systemd)

Il kernel espone le informazioni tramite lo pseudo-filesystem **/proc** -> file "finti", quando leggiamo qualcosa da quel file in realtà stiamo chiedendo al kernel di darci le informazioni rispetto al processo

Ogni processo ha due directory:

- una **directory root**, che viene usata ogni volta che usiamo un percorso assoluto (ossia che inzia con slash /)
- una **directory di lavoro**, che è usata per i percorsi relativi (quindi la directory corrente)

Le system call per gestire i processi sono 4:

- **fork** -> crea un nuovo processo
- **_exit** -> termina il processo chiamante (exit senza _ funzione di libreria)
- **wait** -> aspetta la terminazione di un processo figlio
- **execve** -> esegue il programma, sostituisce completamente lo spazio di indirizzamento del processo che lo invoca -> quindi molte volte vedremo un execve dopo una fork, perché la fork copia il processo (exec e exec* funzioni di libreria)

PID_T FORK(VOID) :

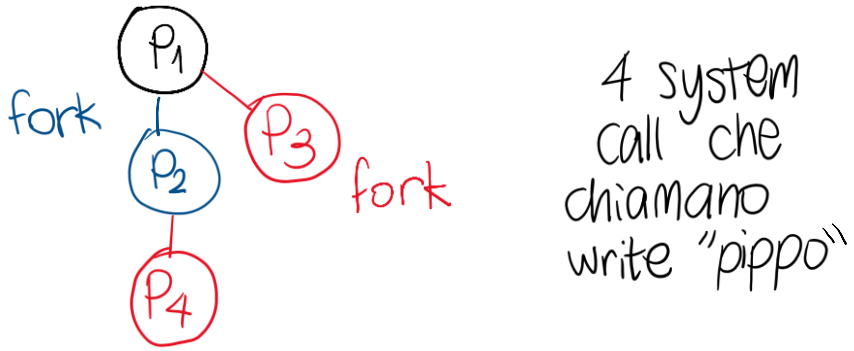
- clona un processo
    - crea un processo figlio che è una copia del processo che ha evocato la fork
- crea un nuovo processo, detto **"figlio" del processo chiamante**
    - cambiano ovviamente il PID e il PPID; i file descriptor (flag compresi) vengono duplicati); l'intero spazio di indirizzamento viene copiato (ottimizzata con copy-on-write)
- **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
        - il processo padre ritorna il PID del figlio

quando viene creato un nuovo processo con fork, i debugger non capiscono che ci sono due processi -> vscode ti fa scegliere se vuoi seguire il padre o il figlio con `set follow-fork-mode[child|parent]` -> posso anche decidere di seguire entrambi con `set detach-on-fork off`

esempio:

```c
#include <sys/types.h>
#include <unistd.h>

int main()
{
        // per far stare nella slide, non controllo i valori di
        // ritorno; NON fatelo nel codice "vero"
        char msg[] = "pippo\n";
        fork();
        fork();
        write(STDOUT_FILENO, msg, sizeof msg);
}
```



void _exit(int status) (in libreria c -> void exit(int status)) :

- **la funzione di libreria**
  - prima di uscire davvero, prima di chiamare la funzione _exit, chiama tutte le funzioni che sono state registrate tramite le funzioni `atexit` e `on_exit`
  - poi svuota i buffer di input/output
  - infine, elimina i file temporanei creati con `tmpfile`
- **in entrambi i casi**
  - vengono chiuse e rilasciate le risorse del processo (i file descriptor vengono chiusi ecc)
  - l'exit-status rimane registrato e dalla shell si può recuperare con $?
  - eventuali figli, ora orfani, vengono adottati da init (pid = 1)

pid_t wait(int *wstatus) :

- serve ad attendere il cambio di stato di un figlio -> si è fermato, è ripartito, è terminato
- wait aspetta solo un figlio qualsiasi, il primo che termina verrà ritornato da wait -> se si vuole aspettare uno in particolare si usa la system call waitpid che permette di specificare quale figlio aspettare
- se la wait va a buon fine, restituisce il pid del figlio che ha terminato, e se il puntatore wstatus è diverso da 0 va a scriverci dentro cosa è successo (se è stato ucciso da un segnale, quale numero di segnale o valore dello stato corrispondente a exit)