# Sistemi 5

Parliamo di come nascono i programmi eseguibili. Normalmente usiamo comandi gcc/clang che compilano i programmi. In realtà questi programmi si limitano a “pilotare” il processo di compilazione, lanciando: preprocessore, compilatore, assemblatore e linker. Compilatori e assemblatori producono file oggetto / rilocabili:  
\*.c + \*.h -> \*.s -> \*.o (+ \*.a +\*.so) -> a.out

Nei file oggetto si trova il codice macchina, ma al cui interno si trovano dei “buchi” (se una variabile viene inizializzata con un file il compilatore non le può inizializzare nel compilato), all’interno del codice ci saranno diversi metadati che vengono presi dal linker, che completa il programma rendendolo un eseguibile.

Il linking può essere statico o dinamico: se il linking è statico il linker fa tutto, copincolla il codice delle librerie, creando programmi auto-contenuti (e che non necessitano di includere altro). Il linking dinamico, che è quello di default, annota nel programma le dipendenze esterne senza effettuare copia e incolla; i pezzi mancanti saranno messi assieme durante l’esecuzione.   
Il vantaggio del linking dinamico è che permette di contenere le dimensioni degli eseguibili che produciamo, se compiliamo un semplice file .c (che stampa solo “ciao mondo”) normalmente questo peserà pochi kilobytes, se invece compiliamo lo stesso file con un linker statico (usando l’opzione -static) l’eseguibile peserà molto di più (probabilmente quasi un megabyte).  
  
Un altro vantaggio del linking dinamico è che se ci sono dei bug nelle librerie basta fixare la libreria buggata e ogni programma che la utilizza risulterà sistemato automaticamente, mentre col linking statico sarà necessario ricompilare il programma (vale però anche il viceversa, se combino un casino su una libreria questo si riflette su tutti i programmi che la linkano dinamicamente).

Il linking dinamico, inoltre, permette di risparmiare memoria RAM, perché se una libreria usa delle variabili costanti o carica sullo stack delle funzioni, allora tutti i programmi che linkano la libreria dinamicamente faranno riferimento alla stessa zona della RAM, senza allocare memoria per ogni programma (come invece accade staticamente).

Gli eseguibili, possono contenere:  
Codice macchina -> sezione .text  
Dati (e dati solo lettura) -> .data e .rodata  
Metadati -> architettura (intel / ARM, 32/64 bits, …), entry-point, che non è il main, è la libreria standard, che viene inizializzata per prima, quanto spazio riservare per variabili non inizializzate -> .bss

In un eseguibile le sezioni vengono raggruppate in Segmenti (che non c’entra con la segmentazione). Questi stanno nell’header Program. Tutte queste informazioni sono memorizzate in “elf”. -> vedi Slides.

Ci basta mappare codice e dati per far “girare” i programmi? No, servono anche lo stack – variabili locali / temporanei, parametri, indirizzi di ritorno (per ragioni storiche, cioè poiché l’heap cresce verso indirizzi crescenti, lo stack cresce verso indirizzi decrescenti) – lo heap e il kernel (non accessibile in modalità utente). In tutte le tabelle dinamiche di tutti i processi ci deve essere mappato anche il kernel (solo un pezzo, per questioni di sicurezza) per poter fare le system call. Il codice dei programmi può essere condiviso in RAM. Per esempio, se lanciamo 7 bash verrà caricato il codice della bash una volta sola, e ci saranno 7 Stack Pointer, 7 Frame Pointer, 7 Program Counters ecc. (condividerà però i dati statici/costanti).

I processi sono identificati da un PID, che, essendo un identificatore, è unico nel sistema: attenzione che il PID è unico in quel momento, quando un processo muore il suo PID viene riutilizzato. La system call per ottenere il PID è pid\_t getpid(void) (getppid(void) invece dà il pid del parent del nostro processo -> i processi formano un albero, con radice “init” o nel mondo moderno SystemD o Upstart ecc.., il cui PID è uguale a 1).

Il kernel espone le informazioni tramite lo pseudo-filesystem (/proc): c’è una directory per ogni processo, che contiene i file che descrivono un processo (con proc/$$ si guarda nella directory che contiene le informazioni sul processo shell che stiamo eseguendo, curiosa è la cartella /map).

Ogni processo ha una directory root, che viene utilizzata per descrivere percorsi assoluti, e una directory di lavoro, che viene usata per descrivere i percorsi relativi (per vedere qual è la directory di lavoro c’è getcwd() oppure sul prompt pwd). I percorsi assoluti iniziano con /, mentre quelli relativi iniziano senza /.

Chdir modifica la directory di lavoro.

Veniamo alle 4 system call principali per i processi: fork, l’unica che crea un nuovo processo, \_exit, che termina il processo chiamante istantaneamente(a differenza della exit del C non fa il flush dei buffer e non esegue altre funzioni), wait, che aspetta la terminazione di un processo figlio, e la execve che esegue un nuovo programma nel processo chiamante, Sostituendo l’intero spazio di indirizzamento: quest’ultima è spesso invocata dopo fork (per lanciare nuovi programmi).

La fork crea un processo, ed è l’unica. Crea il nuovo processo clonando quello chiamante (init è speciale perché viene creato dal sistema). Di fatto, lo spazio di indirizzamento del processo figlio è uguale a quello del padre (compresi i file descriptor, ciò non è costoso per via della copy-on-write nei sistemi moderni; in passato si usavano trucchi sporchi, come vfork, per evitare la copia), con l’unica differenza che il figlio ha un PID e un PPID diverso e che ritorna il PID del figlio al padre e 0 al figlio.

Nei sistemi moderni esiste la virtualizzazione della memoria, quindi quando si copia un processo basta modificare la tabella delle pagine in modo che gli indirizzi del nuovo processo puntino alle pagine di quello vecchio, ma impostando tutte le pagine come se fossero in sola lettura: se un processo prova a scriverci, la pagina viene effettivamente copiata in memoria (grazie a una trap che viene sollevata) e questa pagina copiata diventa scrivibile dal processo che voleva scrivere.

Normalmente quando debuggiamo un programma che fa fork() gbd segue solo uno dei due processi, di base segue il padre, ma se usiamo set follow-fork-mode [child|parent] si può scegliere quale seguire. Se si imposta la modalità detatched (con detatch-on-fork off/on), con “inferior” seguito dall’id è possibile spostarsi da processo padre a figlio e viceversa (per vedere quale processo è parent si usa info inferiors).

Supponiamo adesso di avere una double fork all’interno di un programma, il programma si divide due volte, quindi avremo un processo padre e un processo figlio che chiamano entrambi la fork (dunque il padre ottiene un altro processo figlio e il processo figlio diventa padre di un altro processo). Si hanno quindi in totale 2^n processi, dove n è il numero di fork, in questo caso 2^2 = 4.

Come abbiamo visto prima, ci sono due exit, la exit dello standard C e la \_exit dello standard POSIX (quest’ultima è una syscall).

La exit chiama le funzioni atexit() e on\_exit(), poi svuota i buffer di I/O e elimina i file temporanei con tmpfile, la \_exit non fa nessuna di queste cose. In entrambi i casi, però: vengono chiuse/rilasciate le risorse del processo, il processo termina con un exit status (dalla bash si può vedere questo exit status con $?), uguale a (status & 0xff), e eventuali figli, ora orfani, vengono adottati da init (PID = 1).  
Restituire un valore dal main equivale a fare un exit con quel valore.

Nota: Se si chiama \_exit e ci sono ancora valori nel buffer di stdout, questi non verranno stampati.

Si può osservare cosa succede a un figlio usando la syscall wait(): la wait attende un cambio di stato (la terminazione o uno stop/ripartenza). La wait ritorna al primo figlio che termina, se ne si vuole aspettare uno particolare si usa waitpid().

La wait può fallire se non si hanno figli. Se si passa alla wait un puntatore a intero (che è \*wstatus), al suo interno vengono salvate varie cose quando il figlio termina. Per esempio WIFEXITED() è vera se il programma figlio è uscito con exit status, questo permette di recuperare l’exit status con WEXITSTATUS(), altrimenti WIFISGNALED() è vera se il programma è terminato tramite una segnalazione, questo permette di recuperare il segnale con WTERMSIG().

Un processo terminato, non aspettato dal padre, è uno zombie.