# Sistemi 6

Avevamo parlato di processi, fork, e quindi processi zombie. Un processo padre può usare la wait (o waitpid) per leggere l’exit status di un processo terminato. Finché l’exit status di questo processo non viene letto, un processo terminato rimane nello stato di processo zombie, dunque, la memoria occupata da esso non può essere liberata fino a quel momento. Nel caso di orfani zombie non c’è problema, perché gli orfani sono adottati da init, che fa wait in continuazione. Quando un processo termina al parent arriva il segnale SIGCHLD, di default questo segnale è ignorato, però può essere catturato per “aspettare” i figli.   
Segnali successivi non si accodano: se muoiono due processi uno dopo l’altro prima che il parent abbia chiamato la wait, arriva solo un segnale di SIGCHLD al parent (conviene quindi che nell’handler ci sia un ciclo).

Avevamo accennato a execve, che sostituisce lo spazio di indirizzamento di un processo per eseguire un nuovo comando. A seconda di che argomenti abbiamo possiamo usare più varianti wrapper all’interno della libreria C. La system call effettiva prende come argomenti il path di un file (eseguibile elf / script della shell), un array di stringhe (argv[], sono gli argomenti che arrivano al main del comando lanciato) e le stringhe di ambiente (envp[]), che servono per impostare le variabili d’ambiente.

L’unica vera syscall (di tipo exec) è la execve, tuttavia ci sono tutta una serie di varianti, le versioni “con la l”, che permettono di specificare gli argomenti non come array, ma passandoli direttamente come argomenti alla funzione e terminando le sequenze con NULL (altrimenti la “v” indica che l’exec vuole dei vector), oppure le versioni “con la p”, in cui al posto del pathname si può indicare dirittamente un “file”, e la funzione cercherà il file in tutte le directory presenti nella variabile d’ambiente PATH. Le versioni “con la e” infine, indicano anche la presenza del parametro envp[], che altrimenti è omesso.

L’exec non cambia il PID di un processo, però fa in modo che tale processo faccia girare un programma completamente diverso. Un’altra cosa che non viene toccata dalla execve sono i file descriptors (a meno di FD\_CLOEXEC). È importante che la exec non tocchi minimamente standard input, standard output e standard error. La FD\_CLOEXEC serve per chiudere i file descriptor, quando apriamo files che non siano poi accessibili da altri programmi.

Quando eseguiamo un programma il sistema (kernel + linker, in caso di linking dinamico) perpara:

il codice, mappato r-x (nella sezione .text)

i dati a sola lettura, mappati r—(nella sezione .rodata)

i dati, mappati rw- (in .data, .bss, .heap e .stack)

il kernel, le lebrerie (separate in codice e dati)

Le variabili d’ambiente e gli argomenti della linea di comando vengono scritte nello stack.

La exec dovrebbe essere preceduta da una fork, perché, se ad esempio si chiama l’exec e basta all’interno della bash, al termine del processo non si avrà più una bash a cui tornare e la finestra si chiuderà.

Si ricorda che la fork fa una copia furba delle pagine in RAM (usando tabelle delle pagine che in realtà puntano alle stesse celle, che saranno solo leggibili, ed effettuando una copia effettiva solo quando è richiesto l’accesso in scrittura).

Se in un programma C si chiama execlp(“ls”,”-l”,NULL), il programma che la chiama sarà rimpiazzato da ls -l.  
Se si chiama l’execve, ma il path indicato non è presente sul disco (o esiste ma non è accessibile), la chiamata restituisce -1 impostando errno.  
Se il file esiste ed è un file ELF (che quindi avrà un’intestazione particolare e in cui ci saranno almeno due sezioni di “load” in cui viene indicato cosa caricare come r-x e cosa caricare come rw-) il Kernel prenderà una tabella delle pagine vuota e vi inserirà (se stesso e) i dati, il codice, lo stack, lo heap ecc. come indicato dai Load del file ELF. Poiché c’è però bisogno anche di un interprete, il Kernel inserirà nella tabella anche il linker, che cercherà le librerie necessarie per il file ELF, in maniera ricorsiva (scendendo sempre di più di livello di astrazione), e le inserirà nel nuovo spazio di indirizzamento. Se tutto va a buon fine, il kernel prende il nuovo spazio di indirizzamento e rimpiazza quello del processo precedente con quello.

Se la syscall ritorna al chiamante, vuol dire che qualcosa è andato storto (quindi non serve controllare che il valore di ritorno sia -1 per chiamare perror(), se ha ritornato e siamo ancora nel programma di partenza è sicuramente fallita).

Tutte le redirezioni, le pipeline ecc. sono gestite dalla bash: ai comandi non arrivano come argomenti le redirezioni, è la bash a occuparsi di tutto tramite syscalls. Un modo brutale di fare ciò è chiudere i file descriptors 0, 1, 2 (che corrispondono a stdin stdout e stderr) e poi chiamare la open su altri percorsi per riassegnare i file descriptors 0, 1, 2 all’interno del programma. Il modo “meno brutale” è l’utilizzo della funzione int dup(int olfd). Questa funzione crea un file descriptor equivalente a oldfd (punterà allo stesso file aperto del file descriptor originale) e come open, restituirà il più piccolo FD possibile.  
In Posix i file descriptor sono infatti degli indici in una struttura dati chiamata “Tabella dei file descriptors” (ogni processo ha la sua, e sono tutte contenute in una struttura dati particolare presente in memoria), che contiene i puntatori a tutti i file aperti. Tali puntatori in realtà puntano a un’altra struttura nel kernel, che contiene l’offset del file, i flags di status e il puntatore a i-node.

File descriptors diversi, quindi, possono puntare comunque allo stesso i-node. Tuttavia, ognuno dei file descriptors avrà il proprio file offset (oltre che i flag ma se li apriamo tutti in lettura conta poco). Se però si usa la dup, invece, si fa in modo che entrambi i file descriptors puntino alla stessa riga della tabella dei file aperti (quindi condividono anche il file offset, la status flag e l’inode ptr).

La dup ci permette quindi di duplicare lo standard input su un secondo file descriptor, chiudere il fd 1 e aprirlo per riceverlo da qualche altra parte (con una open): in tal modo anche se la open fallisse, per ripristinare lo standard input basterebbe chiudere il fd 1 e riduplicare il secondo fd.

Il fatto che con la fork siano duplicati i file descriptor torna comodo perché, puntando alla stessa riga della tabella dei file aperti, se uno dei due processi stampa su standard output, l’altro non ci stamperà sopra (perché avrà il file offset aggiornato).

Si nota che tutta questa redirezione degli input/output ha senso perché la exec non tocca i file descriptors.