Processi (e Pipex)

1. Introduzione

Un processo è definibile come una esecuzione di un programma. In particolare, in UNIX un processo è caratterizzato da quattro sezioni logiche:

- codice, in cui è contenuto il codice del processo;
- dati, in cui sono contenuti i dati statici;
- heap, in cui sono contenuti i dati allocati dinamicamente;

Un processo è inoltre caratterizzato dai seguenti ID (numeri interi positivi) assegnati dal kernel (il "cuore" del sistema operativo) al processo all'atto della sua creazione:

- Process-ID (PID), identificativo unico del processo;
- Process-ID del processo padre (PPID), identificativo unico del processo padre;
- User-ID reale (real UID);
- User-ID effettivo;
- Process Group ID (PGID);
- User Group-ID reale (real UGID);
- User Group-ID effettivo.

Una volta in esecuzione il kernel provvede ad inizializzare le proprie strutture dati, ad effettuare alcuni controlli di sistema, e quindi crea il processo init che costituisce il processo "padre" del sistema in quanto tutti i processi successivi sono da esso generati attraverso un meccanismo di duplicazione (fork, vedi 1.3).

Le informazioni relative ai processi sono mantenute in una struttura dati del kernel detta Tabella dei Processi, costituita da un numero predefinito di locazioni (e tale numero determina il numero massimo di processi che possono essere simultaneamente in esecuzione in un dato istante). Quando un processo è creato, il kernel lo alloca in una locazione della Tabella dei Processi e lo dealloca quando il processo è distrutto.

Poiché il kernel è esso stesso un processo, la locazione 0 della Tabella dei Processi è ad esso riservata, mentre il processo init occuperà la locazione 1. L'identificativo della locazione è utilizzata quale identificatore (PID) del processo stesso.

Un processo può trovarsi in vari stati:

- idle, stato iniziale, tipico di un processo appena creato tramite una fork();
- runnable, pronto per l'esecuzione e in attesa che la CPU sia disponibile;
- running, in esecuzione (occupa la CPU);
- sleeping, in attesa di un evento per riattivarsi, ad esempio se un processo esegue una read(), si addormenta fino a quando la richiesta di I/O non è completata;
- suspended, il processo è stato "congelato" (frozen) da un segnale, come ad esempio SIGSTOP; il processo è "scongelato" solo quando riceve il segnale SIGCONT;
- zombified, risorse rilasciate ma ancora presente nella Tabella dei Processi;

Lo stato di un processo può essere visualizzato tramite il comando ps:

$nelly ~> ps w

PID TTY	STAT	TIME COMMAND
30142 pts/6	S	0:00 -bin/tcsh
30179 pts/6	R	0:00 ps w

Un processo può cambiare il codice che esegue attraverso la chiamata ad una delle procedure di sistema della famiglia exec(). Tale meccanismo è tipicamente utilizzato da un figlio per differenziarsi dal padre. Quando il processo figlio termina (ad esempio, in maniera volontaria tramite la chiamata exit()), attraverso il meccanismo delle interruzioni software (segnali), la sua terminazione viene comunicata al padre. Il padre, che solitamente si sospende (tramite la chiamata della wait()) in attesa della terminazione di uno dei suoi figli, quando riceve la segnalazione della terminazione del figlio si risveglia (permettendo la deallocazione del processo figlio dalla tabella dei processi).

2. PID

Un processo può conoscere il suo PID e quello del padre (PPID) attraverso:

- getpid

- getppid

Ad esempio Il PPID del processo 1 (init) è 1.

3. ZOMBIE

Uno stato particolare in cui un processo può venire a trovarsi è quello di zombie. Un processo entra in tale stato se termina, ma suo padre non accetta il segnale SIGCHLD ad esso relativo. Si noti che questa situazione non avviene mai se il processo figlio è adottato in quanto il processo init accetta automaticamente il segnale. Tale situazione, invece, si determina quando il padre è comunque vivo, ma non esegue una chiamata di sistema wait() la quale è responsabile per la ricezione e il servizio del segnale SIGCHLD.

Un processo zombie non occupa nessuna risorsa del sistema tranne la propria locazione della Tabella.

4. PIPEX

Pipex è un programma Unix-like che consente di concatenare il risultato dell'esecuzione di comandi diversi in uno stesso output attraverso l'uso della cosiddetta shell pipeline.

In pratica, Pipex legge l'input da standard input o da un file, lo manipola eseguendo una serie di comandi, e quindi scrive l'output su standard output o su un file.

Per fare ciò, Pipex crea due processi figli con la system call fork() e li collega tramite una pipe creata con la system call pipe(). Uno dei due processi esegue la prima parte della pipeline (solitamente un comando che legge l'input) e invia i dati letti all'altro processo tramite la pipe. Quest'ultimo invece esegue la seconda parte della pipeline (ad esempio un comando che elabora i dati) e invia l'output alla console o a un file.

L'utilizzo di Pipex può essere realizzato anche dall'utilizzo della funzione popen() presente nella libreria C standard, che permette di aprire un file stream per l'esecuzione di processi. Tuttavia Pipex offre maggiore controllo sul processo attraverso la gestione diretta delle pipe e la creazione di nuovi processi tramite fork().

5. FUNZIONI

4.1 Fork

Meccanismo di duplicazione invocato da una chiamata di sistema (fork()) che causa la duplicazione del processo chiamante. Il processo che invoca la fork() è detto processo padre, mentre il processo generato (duplicato) è chiamato figlio. A parte alcune informazioni, fra cui il PID e PPID, il processo figlio è del tutto identico al processo padre: il codice, dati, heap e stack sono copiati dal padre. Inoltre, il processo figlio, continua ad eseguire il codice “ereditato” dal padre a partire dalla istruzione seguente alla chiamata della fork().

Se la chiamata ha successo, fork() ritorna il PID del figlio al padre e 0 al figlio. Se fallisce, restituisce valore -1 al padre e nessun processo figlio è creato. Notare che le variabili e puntatori del padre sono duplicati e non condivisi da padre a figlio. Al contrario, i file aperti del padre al momento della chiamata sono condivisi. In particolare,

- sono condivisi anche i puntatori ai file usati per I/O (ma vengono mantenute copie distinte per ogni processo);
- l’ I/O pointer è modificato per entrambi i processi in seguito ad operazioni di lettura/scrittura da parte degli stessi.

4.2 access

In C, la funzione "access" serve per verificare se il processo in esecuzione ha i permessi di accesso su un file o una directory specifici.

Ecco un esempio di sintassi della funzione access:

```
int access(const char *path, int mode);
```

Il primo argomento, "path", specifica il percorso del file o della directory la cui accessibilità deve essere determinata. Il secondo argomento, "mode", specifica il tipo di controllo di accesso che deve essere effettuato e può assumere uno dei seguenti valori o una combinazione di essi:

R_OK: controlla i permessi di lettura.

W_OK: controlla i permessi di scrittura.

X_OK: controlla i permessi di esecuzione.

F_OK: controlla l'esistenza del file.

La funzione restituisce 0 se il processo in esecuzione ha i permessi richiesti sul file o sulla directory specificati e -1 altrimenti. Nel caso in cui la funzione restituisca -1, è possibile utilizzare la variabile "errno" per determinare il tipo di errore verificatosi.

Si noti che la funzione "access" determina soltanto se il processo in esecuzione ha i permessi richiesti sul file o sulla directory; non modifica in alcun modo i permessi.

4.4 Unlink

In C, la funzione "unlink" viene utilizzata per eliminare un file dal file system.

Ecco un esempio di sintassi della funzione unlink:

```
int unlink(const char *filename)
```

L'unico argomento della funzione "unlink" è "filename", che rappresenta il nome del file da eliminare dal file system.

Il valore restituito dalla funzione "unlink" è 0 se la rimozione del file è avvenuta correttamente, -1 in caso di errore. In caso di errore, la variabile "errno" può essere utilizzata per determinare il tipo di errore verificatosi.

Si noti che la funzione "unlink" non può essere utilizzata per eliminare una directory. Per eliminare una directory vuota, invece, si può utilizzare la funzione "rmdir".

4.4 Exit

exit() chiude tutti i descrittori di file del processo chiamante; dealloca il suo codice, dati, heap e stack, ed infine termina il processo chiamante. Inoltre invia il segnale di terminazione SIGCHLD al processo padre del processo chiamante ed attende che il codice di stato di terminazione (status) sia accettato. Solo gli 8 bit meno significativi di status sono utilizzati, quindi lo stato di terminazione è limitato ai valori nel range 0-255.

Nel caso in cui il processo padre sia già terminato, il kernel assicura che tutti i processi figlio diventino "orfani" e siano adottati da init, provvedendo a settare i PPID dei processi figli al valore 1 (cioè il PID di init). Ovviamente, exit() non ritorna alcun valore. Se eseguita in main è equivalente ad una return().

4.5 wait e waitpid

Il modo corretto per il padre di attendere la terminazione di un figlio viene realizzato in modo passivo (attesa passiva) dalla chiamata di sistema wait() e waitpid().

wait() sospende un processo fino alla terminazione di uno qualunque dei suoi processi figli. In particolare, la chiamata attende la terminazione di un processo figlio e ne ritorna il PID, mentre nel parametro di uscita status vengono restituiti, attraverso una codifica "a byte", il motivo della terminazione e lo stato di uscita del processo che termina. Se al momento della chiamata esiste un processo figlio zombie, la chiamata serve immediatamente il corrispondente segnale e termina.

In caso di errore, invece del PID, viene ritornato il **valore -1**. Se il processo chiamante non ha figli, la chiamata ritorna immediatamente con **valore -1**.

waitpid() si comporta in modo simile a wait(), però permette di specializzare l'attesa. In particolare, il processo chiamante viene posto in attesa della terminazione del figlio con PID pid. Il valore di pid può assumere i seguenti range:

- **< -1** prescrive l'attesa della terminazione di un qualunque processo figlio il cui PGID (identificativo di process group) è uguale al valore assoluto di pid;
- **-1** prescrive l'attesa della terminazione di un qualunque processo figlio; il comportamento è quindi identico a quello della wait();
- **0** prescrive l'attesa della terminazione di un qualunque processo figlio il cui PGID (identificativo di process group) è uguale a quello del processo chiamante;

- **> 0** prescrive l'attesa della terminazione del processo figlio il cui PID è uguale al valore di pid.

Il valore di options è il risultato dell'OR di zero o più delle seguenti costanti

- WNOHANG prescrive alla chiamata di ritornare immediatamente se nessun processo figlio è terminato (wait() non bloccante);
- WUNTRACED prescrive alla chiamata di ritornare anche waitpid() ritorna valore -1 negli stessi casi di wait().

waitpid() ritorna valore -1 negli stessi casi di wait(). Inoltre, se c'è l'opzione WNOHANG e non ci sono figli terminati, ritorna 0.

4.6 execve

Un processo può decidere di cambiare il codice che sta eseguendo. In particolare, esso può rimpiazzare il suo codice, dati, heap e stack, con quelli di un eseguibile attraverso la famiglia di chiamate di sistema exec()

I caratteri che seguono la stringa exec nei nomi delle chiamate di sistema individuano le funzionalità peculiari delle stesse. In particolare:

- v indica una chiamata che riceve un vettore nello stesso formato di argv[];
- e indica una chiamata che riceve anche un vettore di ambiente envp[] invece di utilizzare l'ambiente corrente.

Pur perdendo la condivisione della regione del codice col processo padre, oltre al cambiamento delle regioni dei dati, heap e di stack, il processo figlio mantiene la condivisione dei file aperti. Se il file eseguibile non è trovato, la chiamata di sistema ritorna **valore -1**; altrimenti, il processo parte con l'esecuzione del nuovo codice e non ritorna mai.

La execve() che è l'unica vera chiamata di sistema che viene utilizzata per eseguire un nuovo programma in un processo separato. La funzione prende tre argomenti:

- Una stringa contenente il nome del programma da eseguire (*arg)

- Un array di stringhe contenente gli argomenti da passare al programma. L'ultimo elemento dell'array deve essere NULL.(*argv[])
- Un array di stringhe contenente le variabili d'ambiente da passare al nuovo processo. L'ultimo elemento dell'array deve essere NULL.(*envp[])

ATTENZIONE: una exec() non prevede di tornare al programma chiamante, e non produce nuovi processi.

4.7 Ridirezione: dup e dup2

Vediamo come attraverso l'utilizzo della fork e di chiamate di sistema per la duplicazione di descrittori di file si riesca a realizzare la ridirezione dello standard input o output di un processo. La duplicazione di descrittori di file si può ottenere tramite le seguenti chiamate di sistema:

dup() e **dup2()** creano una copia del descrittore di file oldfd. Se le chiamate di sistema ritornano con successo, il vecchio e nuovo descrittore possono essere usati intercambiabilmente. Essi condividono i lock (accesso in mutua esclusione), i puntatori alla posizione nel file, e (quasi) tutte le flag. dup() cerca la prima (cioè con indice più basso) locazione vuota (libera) nella tabella dei descrittori di file e vi ricopia il contenuto della locazione occupata da oldfd. dup2() chiude newfd se questo è attivo e quindi vi ricopia il contenuto della locazione occupata da oldfd. Se hanno successo, entrambe le chiamate restituiscono l'indice della locazione del nuovo descrittore di file, altrimenti restituiscono il valore -1.

4.8 Pipe

La funzione pipe in C consente di creare un canale di comunicazione senza nome tra due processi (o thread) correlati.

La funzione prende un argomento, un array di due interi, e restituisce 0 se ha successo o -1 se si verifica un errore.

L'array di due interi viene utilizzato per restituire i descrittori di file per la lettura e la scrittura nella pipe. Il primo elemento dell'array è il descrittore di file per la lettura dalla pipe, mentre il secondo elemento è il descrittore di file per la scrittura nella pipe.

In altre parole, sulla pipe si possono fare due operazioni: leggere da essa e scrivere su di essa. Quando si invia qualcosa sulla pipe, tutto ciò che viene scritto attraverso il descrittore relativo alla scrittura(il secondo) verrà letto in ordine dal descrittore relativo alla lettura.

Un uso comune della funzione pipe è quello di fornire un metodo di comunicazione tra i processi padre e figlio di un programma. Ad esempio, il padre può scrivere dei dati sulla pipe e attendere che il figlio li legga, oppure il figlio può inviare notifiche al padre attraverso il canale di comunicazione senza nome creato dalla pipe.