

Osservazioni informali sui progetti di LabSO 2016-2017

r. 001-2017_05_19

- Nel “pmanager” l’azione di “clone” richiede un processo di riferimento il quale deve eseguire al suo interno un “fork”. Quindi un `pclone NOME` dovrà comunicare al processo “NOME” tale intenzione così che esso effettui un “fork”. Tali cloni sono dunque “figli” del processo “NOME”.
- In caso di problemi (in particolare in fase di esecuzione) con i riferimenti al “path” (ad esempio se si vuole fare riferimento a contenuti dentro la cartella di lavoro) si può pensare ad uno script bash di boot (richiamabile direttamente e/o con una regola Makefile tipo “make run”) che come prima azione si posizioni sulla cartella del progetto stesso e poi lanci l’eseguibile. Si può partire da un esempio come:

```
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
cd $DIR
./nomeprogetto [qui si lancia l'eseguibile vero e proprio]
```

- Nel “pmanager” le omonimie devono essere gestite in modo che non ci siano inconsistenze o errori logici. Come minimo si può pensare ad una forma di controllo sui nomi in modo da evitare di avere processi omonimi, altrimenti si può fare qualcosa di meglio consentendo tale situazione (in tal caso le azioni devono essere in grado di supportare anche elenchi di processi, come ad esempio nel caso di `pinfo NOME` dove all’utente sarà mostrata una lista e/o data la possibilità di scegliere su quale elemento intervenire con interazioni o opzioni aggiuntive)
- In caso di chiusura di un processo “figlio” se il padre non gestisce tale situazione (aspettando la chiusura o ignorando esplicitamente tale evento) esso diventa un processo pendente (“zombie”). Per evitare questo si deve opportunamente intervenire.

Un esempio utilizzabile nel “padre” è:

```
#include <signal.h>
...
struct sigaction sigchld_action = {
    .sa_handler = SIG_DFL,
    .sa_flags = SA_NOCLDWAIT
};
sigaction(SIGCHLD, &sigchld_action, NULL);
...
```

In questo caso l'ultima riga indica che l'evento `SIGCHLD` (inviato al "padre", scatenato dalla terminazione di un "figlio") deve avere come handler `SIG_DFL` (il default) con flag `SA_NOCLDWAIT` che è quello apposito per risolvere tale situazione... alternativamante si può usare come handler `SIG_IGN` (ci sono delle leggere differenze tra i due approcci che si possono ricavare dai manuali):

```
signal(SIGCHLD, SIG_IGN);
```

- La gestione degli errori deve essere completa: non devono esserci "crash" e nemmeno errori nella logica della gestione dei processi (ad esempio chiedere informazioni su un processo e vedere quelle di un altro per problemi con le denominazioni o per un'errata gestione delle omonimie), in particolare nell'uso delle syscall (verificare sempre i valori di ritorno di tali chiamate!).
- Dove si parla di processi "indipendenti" si vuole semplicemente far riferimento a processi "distinti", ossia due processi e non uno solo.
- Nel "countdown" stato/colore dei led (on/off o in maniera più completa colore_1, colore_2, ..., colore_n/off) sono da far gestire ai singoli processi "foglie" della struttura descritta. L'idea è che l'informazione di ciascun led è a carico di uno specifico processo che riceve le informazioni necessario per impostarlo e ne comunica lo stato quando necessario agli altri.
- In caso di uso di file "temporanei" (anche per le "fifo") si può utilizzare qualsiasi cartella: l'ideale è comunque avere una macro-cartella di appoggio (preferibilmente dentro la cartella di esecuzione e/o nella cartella temporanea di sistema) con sotto-cartelle, se la struttura ha contenuti di vario genere. L'importante è che ci sia ordine.
- La regola "assets" va utilizzata solo se necessaria (per esempio per creare dei file "di appoggio" per qualche operazione), altrimenti si può evitare completamente o crearla "vuota".
- Nel "countdown" il metodo di test prevede che il programma sia richiamabile passando come argomento i secondi per il timer: deve poi partire la shell interattiva con il timer già avviato.
- Nel "pmanager" `plist` e `ptree` nell'implementazione ottimale dovrebbero mostrare tutti i processi generati, quindi anche quelli chiusi, ma almeno - ovviamente - quelli attivi. La differenza tra i due è sostanzialmente solo di organizzazione dell'output (nel caso della visualizzazione "ad albero" avere anche i processi chiusi è un vantaggio, altrimenti bisogna comunque avere un modo di mostrare la gerarchia correttamente). Una questione a parte è data dalla possibilità di gestire efficacemente la chiusura dei processi dall'esterno (ad esempio dalla shell): è importante almeno evitare "crash" se si

riferiscono processi non più esistenti.

Si possono implementare regole/azioni/opzioni aggiuntive se ragionevoli e che aggiungano elementi interessanti (ad esempio opzioni aggiuntive per gestire la fase di test con varie modalità).

Quando si effettua un FORK la “copia” delle informazioni non è una “clonazione tout-court”... in particolare i riferimenti agli indirizzi di memoria (ad esempio heap/stack) sono relativi al processo (indirizzi virtuali) per cui non c’è “sovrapposizione” (e nemmeno possibilità di scambiare informazioni come se si avesse uno spazio di memoria condivisa).

- Nel “pmanager” l’azione `pclose` si intende che porti alla TERMINAZIONE del processo coinvolto (non basta che resti attivo con un qualche “flag” interno che ne indichi lo stato)

- Nel “pmanager” la gestione degli indici per i cloni deve essere verificata correttamente: se ad esempio si crea un processo “NOME”, poi si genera un clone “NOME_1” e lo si elimina subito dopo, poi si rigenera il clone prestare attenzione alla denominazione... si avrà “NOME_1” nuovamente o “NOME_2”? Dipende se si è in grado di gestire le omonimie (o come si gestiscono comunque). Questo problema è anche da valutare se l’utente crea ex-novo un processo che si chiama “NOME_x” (x intero)... se è consentito bisogna stare attenti alle azioni che potrebbero creare conflitti. Questo vale per tutti i casi problematici.

La soluzione ottimale è in grado di gestire tutti questi casi.

- Tutte le “keywords” valide del C sono accettabili (non vietate), ma possono esserci situazioni che nascondono insidie. Ad esempio l’uso del “goto” con `goto` massima probabilità è indice di una soluzione non ottimale.

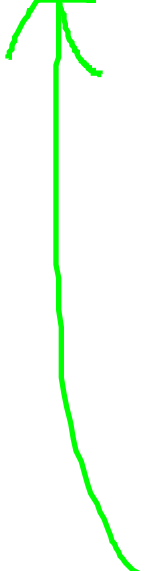
- Nel “pmanager” le informazioni sui processi (nei feedback delle azioni e/o in risposta a `pinfo` ad esempio) devono comprendere i dati minimi utili (come pid, ppid e nome processo): sono buone soluzioni quelle che aggiungono dati utili aggiuntivi quali potrebbero essere (sono solo esempi!): lo stato (attivo/chiuso), i tempi (creazione, durata, chiusura), etc.

- Nel caso si voglia gestire una sincronizzazione con i segnali con un modello del tipo:

P1: azione (es.: scrittura di un dato) / pause (per attendere l’altro processo)

P2: azione / invio segnale a P1 che “sblocchi” l’attesa

Questo funziona se P1 va in pausa prima del segnale di ritorno, altrimenti si fermerebbe in tal punto.



Per ovviare a questo anziché usare pause si può sfruttare un ciclo controllato (a mo' di semaforo) ad esempio con un while su una variabile con qualcosa tipo (solo come esempio di riferimento):

P1: x=1 / azione / while (x=1) { sleep(1); }; [nell'handler del segnale si fa x=0]

P2: azione / invio segnale

così facendo il segnale può arrivare prima o dopo del ciclo di attesa: nel primo caso il ciclo "uscirà" subito, nel secondo caso sarà "fermato"

- Alcune informazioni secondarie possono essere ricavate in maniera indiretta (ad esempio il parent_id di un altro processo accedendo alla cartella di sistema /proc/<proc_id>).