Esempio Shell-Pipe (manca)

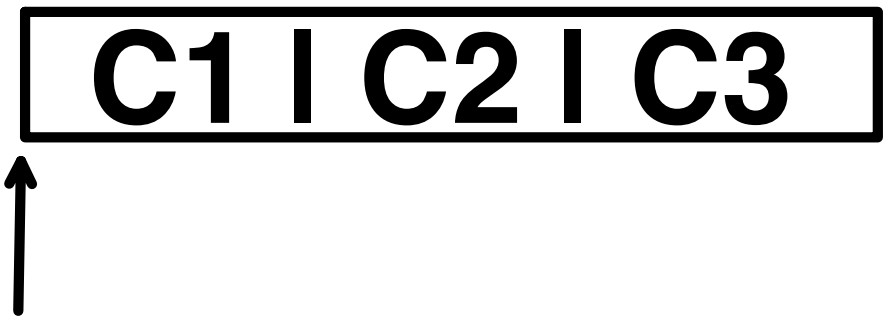
mercoledì 3 maggio 2023 16:38

ESEMPIO SHELL-PIPE

```c
#define COMMAND_LENGTH 1024

int count_args(char *start_command, int *pipe_present, char **next_command) {
    char copied_string[COMMAND_LENGTH + 1];
    char *temp, *temp_next;
    int args = 0, i;

    strcpy(copied_string, start_command);
    if (((temp = strtok(copied_string, " \n")) == NULL) || (strcmp(temp, "\0") == 0)) return(-1);
    args++;
    while (((temp = strtok(NULL, " \n")) != NULL) && (strcmp(temp, "|") != 0 )) {
      if (strcmp(temp, "\0") == 0 ) break;
      args++;
    }
    if (temp == NULL) {
      *pipe_present = 0;
      *next_command = NULL;
    } else {
   if ((temp = strtok(NULL, " \n")) != NULL) {
    *pipe_present = 1;
            for (i=0; ; i++) if (copied_string[i] == '|') break;
    *next_command = start_command + i + 2;
   }
   else {
    *pipe_present = 0;
    *next_command = NULL;
   }

    }
    return(args);
}

char **build_arg_vector(char *command, int arg_num, int pipe_present) {
  char **arg_vector;
  char *temp;
  int i;

  arg_vector = malloc((arg_num+1) * sizeof(char *));
  for(i=0; i < arg_num; i++) {
    if (i==0) temp = strtok(command, " \n");
    else temp = strtok(NULL, " \n");
    arg_vector[i] = temp;
  }
  arg_vector[i] = NULL;
  return(arg_vector);
}
```

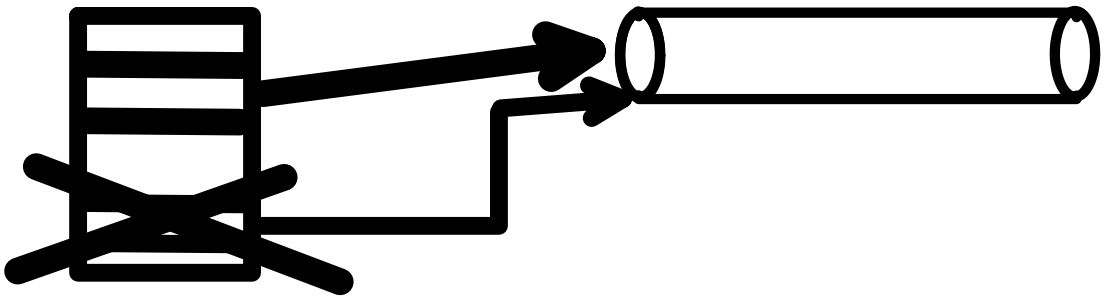
```c
int main () {

  char *command_pointer, *next_command;
  char line[COMMAND_LENGTH+1];
  char normalized_line[3*COMMAND_LENGTH+1];
  char **arg;
  int *old_pipe_descriptors;
  int *new_pipe_descriptors;
  int arg_num;
  int pipe_present=0, pipe_pending=0;
  int pending_processes;
  int i, j, status;

    while (1) {
   printf("\nExample Shell");
   if (geteuid() == 0) printf("#");
   else printf(">");
     fflush(stdout);
      fgets(line,COMMAND_LENGTH,stdin);
      line[COMMAND_LENGTH] = '\0';
      if (strlen(line) == COMMAND_LENGTH) {
          printf("\nCommand too long!\n");
          continue;
      }

    for (i = 0, j = 0; i< strlen(line)+1 ; i++, j++){
     normalized_line[j] = line[i];
     if (normalized_line[j] == '|'){
       normalized_line[j++] = ' ';
       normalized_line[j++] = '|';
       normalized_line[j] = ' ';
     }


    }
      pending_processes = 0;
      command_pointer = normalized_line;

       if (strcmp(command_pointer, "exit\n") == 0) break;
     do {
        arg_num = count_args(command_pointer, &pipe_present, &next_command);
        if (arg_num < 0) break;

          arg = build_arg_vector(command_pointer, arg_num, pipe_present);

          if(pipe_present) {
       new_pipe_descriptors = malloc(sizeof(int)*2);
            if (pipe (new_pipe_descriptors) < 0) {
              printf("Can't open a pipe for error %d!\n", errno);fflush(stdout);
             exit(EXIT_FAILURE);
            }
          }

          if ((i=fork()) == 0) {
       if (pipe_pending) {
         close(0);
              dup(old_pipe_descriptors[0]);
               close(old_pipe_descriptors[0]);
       }
       if (pipe_present) {
          close(1);
          dup(new_pipe_descriptors[1]);
          close(new_pipe_descriptors[1]);
       }

            execvp(arg[0],arg);
       printf("Can't execute file %s\n",arg[0]);fflush(stdout);
       exit(EXIT_FAILURE);
          } else if (i>0) {
            pending_processes++;
       if (pipe_pending) {
         close(old_pipe_descriptors[0]);
         free(old_pipe_descriptors);
       pipe_pending = 0;
      }
      if (pipe_present) {
          close(new_pipe_descriptors[1]);
          old_pipe_descriptors = new_pipe_descriptors;
       pipe_pending = 1;
      }
          } else {//else if
            printf("Can't spawn process for error %d\n", errno);fflush(stdout);
          }
       command_pointer = next_command;
     } while (next_command != NULL);

   for (i=0; i<pending_processes; i++) {
      wait(&status);
     }
  }

  printf("\nLeaving the Shell\n\n");
  return(0);

}
```

Qui guardiamo un comando che ci viene dato in input - se il comando ha più di una PIPE (quindi ci sono delle PIPE) - NOI CREIAMO QUESTE PIPE E RIDIREZIONIAMO LO STANDARD INPUT E LO STANDARD OUTPUT DELLE VARIE APPLICAZIONI CHE VENGONO AD ESSERE LANCIATE ESATTAMENTE PER LAVORARE SU QUESTA PIPE.
All'interno del main non abbiamo altro che un while(1), e ovviamente in fgets ci andiamo a prendere all'interno di un'area di memoria che è LINE, un comando: un comando può essere fatto esattamente come abbiamo specificato prima, quindi comando1 | comando2 | comando3 … etc!
C1 deve avere lo standard output ridirezionato sulla pipe di destra, C2 deve avere lo standard input ridirezionato sulla pipe di sinistra e lo standard output ridirezionato sulla sua pipe di destra e C3 deve avere lo standard input ridirezionato sulla sua pipe di sinistra.
Ciò che produce C1, con printf, viene ad essere letto da C2 magari con scanf(), e quindi non utilizziamo più la tastiera-utilizziamo questo oggetto di I/O per andare a far parlare questi comandi che stiamo lanciando.
Nel momento in cui abbiamo letto un comando e caricato all'interno di LINE, sicuramente andiamo a chiudere la sequenza di byte con uno \0, perché poi vogliamo lavorare all'interno di quest'array utilizzando la rappresentazione stringa della sequenza di informazioni che al più potevamo aver raccolto.
Poi andiamo a fare una string compare di quello che avevamo nell'altra area di memoria, che ora abbiamo assegnato ad command pointer, e vediamo se commandpointer non sia la linea exit. Se è uguale ad exit, chiamiamo break e usciamo dal while(1) e terminiamo la nostra shell. Altrimenti chiamiamo un count_args (quindi contiamo gli argomenti che all'interno di questa linea di comando devono essere utilizzati per quanti riguarda il prossimo comando da lanciare). Quindi se noi abbiamo registrato C1 poi c'è una PIPE (|) inizialmente abbiamo il puntatore all'inizio:



Quello che stiamo facendo è contare quanti sono gli argomenti che compongono C1.
Ad esempio >> ./a.out 16, sono due argomenti, il nome del file lanciato a.out in argv[0] e 16 in argv[1]. E andiamo a verificare se infondo a questo comando è presente una PIPE. Chiamanto count_args, che ovviamente è implementata dentro questo programma, facciamo questa analisi (tipicamente li dentro useremo strtok()) e poi analizzeremo i vari token che abbiamo generato usando strtok.
Detto questo, contiamo gli argomenti per il prossimo dei comandi che dobbiamo lanciare nella linea di comando, ci viene detto qual è il loro numero e, in base a questo numero, andiamo a costruire il vettore che ci servono sulla exec, lo facciamo chiamando questa build_arg_vector, e passiamo il puntatore all'area di memoria che contiene il comando di cui abbiamo contato gli argomenti poi il numero di argomenti che, in quell'area di memoria puntata, sono presenti; Infine pipe_present che nell'attuale implementazione non viene ad essere considerato.
Però pipe_present lo avevamo passato in count_args come puntatore all'area di memoria per andare a capire - nell'analisi che stiamo facendo - se il prossimo comando ha bisogno/necessita di una PIPE.
ATTENZIONE: Se il prossimo comando ha bisogno di una PIPE, quindi if(pipe_present), quando noi generiamo un processo figlio per far sì che possa fare la exec del comando che deve essere lanciato, prima di far questo ovviamente dobbiamo generare una pipe (105), quindi se c'è una pipe presente su questo comando, generiamo una pipe in modo tale da poterla anche utilizzare e farla anche ereditare (quindi ereditare i canali) al processo figlio che noi lanciamo usando la fork() sotto (111).

Quindi allochiamo un array di due interi, ci facciamo dare l'indirizzo in new_pipe_descriptors e passiamo quest'ultimo alla chiamata di pipe in riga 105.
Nel momento in cui con la fork andiamo a generare un processo figlio verifichiamo se il valore di ritorno è uguale a zero - stiamo facendo questa verifica a seconda se c'era una PIPE presente DEL COMANDO CHE QUESTO PROCESSO ADESSO DEVE LANCIARE nella execvp, noi stiamo chiudendo 1 (ossia chiudiamo std::out), duplichiamo il canale di scrittura della pipe e la duplicazione avviene sullo standard output che è il primo dei canali che attualmente abbiamo chiusi; E poi chiudiamo il canale di scrittura sulla PIPE - tanto sulla PIPE ci arriviamo utilizzando l'altro canale con codice 1 grazie alla duplicazione - in quel processo che dovrà fare la exec di un comando che, utilizzando lo std::out, dovrà SCRIVERE i dati in questa PIPE, bene, il canale che ci portava a scrivere i dati sulla PIPE, lo stiamo duplicando col canale col codice numerico 1, e poi chiudiamo quello vecchio.



In questo while è possibile che io mi trovi che sono arrivato ad eseguire un comando su un processo generato con quella fork(), e il comando precedente aveva una PIPE, quindi c'è una pipe_pending per cui il programma attuale in realtà NON deve leggere da std:in ma deve leggere da PIPE. Ovviamente in questo caso la PIPE già esiste perché è stata generata in precedenza per eseguire il comando che scrive su questa PIPE, e noi sappiamo esattamente qual è questa PIPE old_pipe_descriptors[0] ci dice qual è il suo estremo di lettura, chiudiamo lo standard input, duplichiamo sullo standard input quel canale, chiudiamo il canale di lettura e stiamo apposto. Nel momento in cui siamo nel parent, se c'è una PIPE_PRESENT nell'ultimo comando che è stato lanciato, dobbiamo andare ad indicare che c'è una old_pipe_descriptors che è il puntatore a due interi che deve puntare alla new_pipe_descriptors che è la PIPE che è stata utilizzata per l'output di quel comando in modo tale che il prossimo comando ancora si troverà la PIPE pendente e in old_pipe_descriptors i canali che ci portano su quella PIPE.
Quindi siamo in grado di far sì che quando lanciamo un processo con la fork() siamo in grado di capire se c'è una pipe_pending da cui dobbiamo leggere quando facciamo la exec sotto e dobbiamo eseguire l'applicazione - ovviamente utilizzando l'informazione che ci va indicando che la tabella dei file descriptor deve portare questa applicazione a leggere da std::input e standard output ridirezionato sull'altra PIPE, quindi abbiamo la new_pipe_descriptors e la old_pipe_descriptors che ci permettono di arrivare sui descrittori delle due PIPE.

```
/shell-with-pipes> ./a.out

Example Shell>ls
a.out  shellpipe-unix.c

Example Shell>ls -la
total 32
drwxr-xr-x 2 francesco users  4096 Apr 22 12:51 .
drwxr-xr-x 7 francesco users  4096 Apr 29  2020 ..
-rwxr-xr-x 1 francesco users 17640 Apr 22 12:51 a.out
-rw-r--r-- 1 francesco users  3565 Apr 20 12:29 shellpipe-unix.c

Example Shell>cat shellpipe-unix.c | more
```