[T09] Esercitazione 9

Istruzioni per l'esercitazione:

- Aprite il form di consegna in un browser e loggatevi con le vostre credenziali `uniroma1`.
- Scaricate e decomprimete sulla scrivania il codice dell'esercitazione. Vi sarà una sotto-directory separata per ciascun esercizio di programmazione.
  Non modificate in alcun modo i programmi di test `*_main.c`.
- Rinominare la directory chiamandola `cognome.nome`. Sulle postazioni del laboratorio sarà `/home/biar/Desktop/cognome.nome/`.
- È possibile consultare appunti/libri e il materiale didattico online.
- Rispondete alle domande online sul modulo di consegna.
- **Finiti gli esercizi**, e non più tardi della fine della lezione:
  - **zippate la directory di lavoro** in `cognome.nome.zip` (`zip -r cognome.nome.zip cognome.nome/`).
- **Per consegnare**:
  - inserite nel form di consegna come autovalutazione il punteggio di ciascuno dei test forniti (inserite zero se l'esercizio non è stato svolto, non compila, o dà errore di esecuzione).
  - fate **upload** del file `cognome.nome.zip`.

Per maggiori informazioni fate riferimento al regolamento delle esercitazioni.

Esercizio 1 (Parsing di una linea da `stdin` in C)

Questo è un esercizio di mera conoscenza del linguaggio C. Il canale `stdin` (di tipo `FILE*`) modella in C la sorgente di caratteri ASCII che proviene come input, salvo diversamente specificato, dal terminale. Si chiede di scrivere nel file `E1-get-cmd-line/e1.c` una funzione `void get_cmd_line(char* argv[]);` che legge la prossima linea di testo da `stdin`, estrae ciascun token (sequenza consecutiva di caratteri, esclusi spazi, tab `\t` e ritorni a capo `\n`) e produce un array di al più `n<=MAX_TOKENS=64` stringhe come segue:

- le stringhe prodotte (argv[0]…argv[n-1]) devono essere allocate dinamicamente con `malloc`;
- la stringa in ultima posizione (argv[n]) deve essere `NULL`, fungendo da "terminatore".

Assumere che la linea di testo letta da `stdin` contenga al più 1024 caratteri compreso il ritorno a capo `\n`.

Esempio: se la linea letta da `stdin` è `rm -f .DS_store`, la chiamata `get_cmd_line(argv)` restituisce in `argv` l'array di stringhe `{"rm", "-f". ".DS_Store", NULL}`.

Suggerimenti. Basandosi sul comando `man` (oppure la documentazione online cercando `funzione opengroup`) usare:

- la funzione `fgets` per leggere una linea da `stdio` di al più `MAX_LINE=1024` caratteri terminata dal ritorno a capo `\n`;
- la funzione `strtok` per tokenizzare la linea una volta letta da stdio.

L'array di stringhe prodotto deve essere deallocabile con la semplice funzione:

```
void free_args(char* argv[]) {
    while (*argv) free(argv++);
}
```

Usare il main di prova nella directory di lavoro `E1-parse-line` compilando con `gcc e1_main.c e1.c -o e1´.

Esercizio 2 (Scrittura di una semplice shell dei comandi Linux)

[Svolgere questo esercizio solo dopo aver svolto l'Esercizio 1]

Una `shell` è un programma che chiede all'utente di eseguire altri programmmi sotto forma di nuovi processi, passandogli eventuali argomenti specificati dall'utente. Una shell fornisce normalmente un `prompt`, vale a dire un breve testo (es. $, >, ecc.) che segnala all'utente che la shell è in attesa di ricevere comandi.

Si chiede di scrivere nel file `E2-shell/e2.c` una semplice shell sotto forma di una funzione `int do_shell(const char* prompt);` che prende come parametro la stringa di prompt e si comporta come segue;

1. stampa il prompt;
2. attende che l'utente inserisca in `stdin` un comando seguito dai suoi eventuali argomenti (es: `ls -l`, dove `ls` è il comando e `-l` è il suo unico argomento). Per ottenere comando e argomenti da `stdin` usare il risultato dell'esercizio 2;
3. se il comando è vuoto (NULL) tornare al punto 1;
4. se il comando è `quit` terminare con successo la shell;
5. creare con `fork` un nuovo processo che esegua il comando con gli argomenti dati usando `execvp`;
6. se il comando si riferisce a un programma inesistente riportare l'errore `unknown command` seguito dal nome del comando e tornare al punto 1;
7. attendere con `wait` la terminazione del processo e tornare al punto 1.

Per valutare il corretto funzionamento della shell, effettuare i seguenti quattro test:

1. inserire `ls -l` e verificare che listi la directory corrente
2. inserire `echo hello` e verificare che venga stampato hello
3. inserire `sergente hartman` e verificare che venga stampato un messaggio di errore
4. inserire `quit` e verificare la terminazione della shell

Il risultatao di **ogni** system call **deve** essere controllato e in caso di errore segnalato con `perror` e terminazione EXIT_FAILURE.

Esercizio 3 (Domande)

Rispondi alle seguenti domande, tenendo conto che una risposta corretta vale 1 punti, mentre una risposta errata vale 0 punti.

Domanda 1 Relativamente al seguente codice C (supponendo che la `fork` non generi un errore), quale tra le seguenti affermazioni è *VERA*?

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int main() {
    int x = 1;
    pid_t pid = fork();
    if (pid == -1) {
        perror("Errore nella fork");
        exit(1);
    }
    if (pid == 0) {
        printf("Figlio: %d\n", x);
        x = 2;
        _exit(0);
    }
    wait(NULL);
    printf("Padre: %d\n", x);
    return 0;
}
```

- **A**. Viene stampato "`Padre: 1`" e poi "`Figlio: 2`"
- **B**. Viene stampato "`Figlio: 2`" e poi "`Padre: 1`"
- **C**. Viene stampato "`Figlio: 1`" e poi "`Padre: 1`"
- **D**. Viene stampato "`Figlio: 1`" e poi "`Padre: 2`"
- **E**. Viene stampato "`Figlio: 2`" e poi "`Padre: 2`"
- **F**. L'ordine delle stampe è impredicibile, poiché dipende dallo scheduler
- **G**. Nessuna delle precedenti

Domanda 2 Relativamente al seguente codice C (supponendo che la `fork` non generi un errore), qual è l'output atteso?

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

#define N 3

int main() {
    int i, x = 10;
    for (i = 1; i <= N; i++) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("Errore nella fork");
            exit(1);
        }
        if (pid == 0) { // figlio
            printf("Figlio %d: x=%d\n", i, x);
            exit(0);
        }
        wait(NULL);
        x += 10;
    }
    printf("Padre: x=%d\n", x);
    return 0;
}
```

- **A**. Nell'ordine (una per linea): "`Figlio 1: x=10`", "`Figlio 2: x=20`", "`Figlio 3: x=30`", "`Padre: x=40`"
- **B**. Nell'ordine (una per linea): "`Figlio 1: x=10`", "`Figlio 2: x=10`", "`Figlio 3: x=10`", "`Padre: x=30`"
- **C**. Nell'ordine (una per linea): "`Figlio 1: x=10`", "`Figlio 2: x=10`", "`Figlio 3: x=10`", "`Padre: x=40`"
- **D**. Nell'ordine (una per linea): "`Figlio 1: x=10`", "`Figlio 2: x=20`", "`Figlio 3: x=30`", "`Padre: x=30`"
- **E**. Nell'ordine (una per linea): "`Figlio 1: x=10`", "`Figlio 2: x=30`", "`Figlio 3: x=60`", "`Padre: x=100`"
- **F**. L'ordine delle stampe è impredicibile, poiché dipende dallo scheduler
- **G**. Nessuna delle precedenti

Domanda 3 Relativamente al seguente codice C (supponendo che la execv non generi un errore), qual è l'output atteso?

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main() {
    printf("--- started ---\n");
    char *args[] = {"/bin/echo", "Hello", NULL};
    execv(args[0], args);
    printf("--- finished ---\n");
    return 0;
}
```

- **A**. Nell'ordine (una per linea): "`--- started ---`", "`Hello`", "`NULL`, "`--- finished ---`"
- **B**. Nell'ordine (una per linea): "`--- started ---`", "`Hello`", "`--- finished ---`"
- **C**. Nell'ordine (una per linea): "`--- started ---`", "`Hello`"
- **D**.Quello in aula 16 è Nell'ordine (una per linea): "`--- started ---`", "`--- finished ---`"
- **E**. L'ordine delle stampe è impredicibile, poiché dipende dallo scheduler
- **F**. Nessuna delle precedenti

Domanda 4 Data la seguente funzione f in linguaggio C e la sua traduzione in linguaggio assembly, dire quale tra le seguenti tecniche di ottimizzazione è stata applicata:

```c
int f(int x) {
    int s = 0;
    while (x > 0) s += x--;
    return s;
}
```

```
.globl  f
f:
    subl    $16, %esp
    movl    $0, 12(%esp)
    jmp L2
L3:
    movl    20(%esp), %eax
    leal    -1(%eax), %edx
    movl    %edx, 20(%esp)
    addl    %eax, 12(%esp)
L2:
    cmpl    $0, 20(%esp)
    jg  L3
    movl    12(%esp), %eax
    addl    $16, %esp
    ret
```

- A. Loop unrolling
- B. Register allocation
- C. Common subexpression elimination
- D. Loop invariant code motion
- E. Nessuna delle precedenti

Soluzioni

Esercizio 1 (Parsing di una linea da `stdin` in C)

`e1.c`

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "e1.h"

char* dup_string(const char* in) {
    size_t n = strlen(in);
    char* out = malloc(n + 1);
    strcpy(out, in);
    return out;
}

void get_cmd_line(char* argv[MAX_TOKENS]) {
    int argc = 0;
    char line[MAX_LINE];
    fgets(line, MAX_LINE, stdin);
    char* token = strtok(line, " \t\n");
    argc = 0;
    while (argc < MAX_TOKENS && token != NULL) {
        argv[argc++] = dup_string(token);
        token = strtok(NULL, " \t\n");
    }
    argv[argc] = NULL;
}
```

Esercizio 2 (Scrittura di una semplice shell dei comandi Linux)

e2.c

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include "e2.h"

#define MAX_LINE    1024
#define MAX_TOKENS  64

void do_cmd(char* argv[MAX_TOKENS]) {
    int res;
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork error");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        res = execvp(argv[0], argv);
        if (res == -1) {
            printf("unkwnown command %s\n", argv[0]);
            _exit(EXIT_FAILURE);
        }
    }

    res = wait(NULL);
    if (pid == -1) {
        perror("wait error");
        exit(EXIT_FAILURE);
    }
```

```
}

void deallocate_cmd(char* argv[MAX_TOKENS]) {
        while (*argv != NULL)
                free(*argv++);
}

char* dup_string(const char* in) {
    size_t n = strlen(in);
    char* out = malloc(n + 1);
    strcpy(out, in);
    return out;
}

void get_cmd_line(char* argv[MAX_TOKENS]) {
    int argc = 0;
    char line[MAX_LINE];
    fgets(line, MAX_LINE, stdin);
    char* token = strtok(line, " \t\n");
    argc = 0;
    while (argc < MAX_TOKENS && token != NULL) {
        argv[argc++] = dup_string(token);
        token = strtok(NULL, " \t\n");
    }
    argv[argc] = NULL;
}

int do_shell(const char* prompt){
    for (;;) {
        printf("%s", prompt);
        char* argv[MAX_TOKENS];
        get_cmd_line(argv);
        if (argv[0] == NULL) continue;
        if (strcmp(argv[0], "quit") == 0) break;
        do_cmd(argv);
                deallocate_cmd(argv);
    }
    return EXIT_SUCCESS;
}
```

Esercizio 3 (Domande)

1. **C**. Viene stampato "`Figlio: 1`" e poi "`Padre: 1`"
2. **A**. Nell'ordine (una per linea): "`Figlio 1: x=10`", "`Figlio 2: x=20`", "`Figlio 3: x=30`", "`Padre: x=40`"
3. **C**. Nell'ordine (una per linea): "`--- started ---`", "`Hello`"
4. **E**. Nessuna delle precedenti