[T11] Esercitazione 11

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/studente/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`.
  - **importante**: verificate di aver ricevuto mail di conferma per la sottomissione del form
- Se siete in laboratorio, prima di uscire:
  - **importante**: fate logout dal vostro account Google!
  - eliminate dal desktop la directory creata (`rm -rf cognome.nome`).
  - rimettete a posto eventuali **sedie** prese all'ingresso dell'aula!

Per maggiori informazioni fate riferimento al regolamento delle esercitazioni.

Esercizio 1 (suffisso di una stringa)

Tradurre in IA32 la seguente funzione `is_suffix` definita in `E1-is-suffix/e1.c` che, date due stringhe `s1` e `s2`, verifica se la `s1` è un suffisso di `s2`. Ad esempio `ger` è suffisso di `hamburger`, mentre `gur` non lo è.

`e1.c`

```c
int is_suffix(const char* s1, const char* s2){

    const char *s1_aux = s1;
    const char *s2_aux = s2;

    while (*s1_aux++);
    while (*s2_aux++);
    while (s1 != s1_aux && s2 != s2_aux && *(--s1_aux) == *(--s2_aux));
    return s1 == s1_aux && *s1_aux == *s2_aux;
}
```

Scrivere la soluzione nel file `E1-is-suffix/e1.s`. Usare il file `E1-is-suffix/e1_eq.c` per sviluppare la versione C equivalente e `E1-is-suffix/e1_main.c` come programma di prova.

Esercizio 2 (indicatore di progresso)

Si vogliono usare i segnali per creare un indicatore di progresso per la funzione `do_sort`, che implementa un semplice algoritmo di ordinamento a bolle. L'indicatore di progresso è la percentuale di n coperta da `i`, vale a dire `100.0*i/n`.

Si scriva la soluzione nel file `E2-sort-timer/e2.c` modificandolo e aggiungendo quanto necessario al raggiungimento dell'obiettivo.

`e2.c`

```c
#include "e2.h"

static void do_sort(int *v, int n) {
    int i, j;
    for (i=0; i<n; ++i)
        for (j=1; j<n; ++j)
            if (v[j-1] > v[j]) {
                int tmp = v[j-1];
                v[j-1] = v[j];
                v[j] = tmp;
            }
}

void sort(int *v, int n) {

    // completare con gestione segnali...

    do_sort(v, n);
}
```

Il risultato atteso deve essere come segue (ovviamente i numeri esatti delle percentuali possono variare):

```
start sorting...
------------------------------
5.4%
11.2%    <---- percentuali stampate dal gestore del segnale
17.1%
23.3%
29.7%
36.5%
43.7%
51.5%
59.7%
68.7%
79.0%
90.7%
------------------------------
v[0]=0
v[1]=1
v[2]=2
v[3]=3
v[4]=4
v[5]=5
v[6]=6
v[7]=7
v[8]=8
v[9]=9
```

Suggerimento: rendere la variabile `i` di `do_sort` globale (dichiarata fuori dalla funzione) e tenere in un'altra variabile globale `max` il valore di `n`. In questo modo è possibile accedervi da un handler di un segnale che può stampare il rapporto tra `i` e `max`. Fare riferimento alla dispensa del corso sul segnali.

Esercizio 3 (Domande)

Domanda 1 Quale tra le seguenti affermazioni è *VERA*?

- **A**. Una “trap” è un tipo di interruzione asincrona
- **B**. Una “trap” viene generata, ad esempio, quando si muove il mouse
- **C**. L’istruzione `int` può essere usata esclusivamente per invocare le system call
- **D**. L’interrupt generata dal timer è asincrona
- **E**. Nessuna delle precedenti

Domanda 2 Quale tra le seguenti affermazioni è *VERA*?

- **A**. Una system call si invoca con l’istruzione `int` seguita da un numero che identifica la system call da eseguire
- **B**. Il valore di ritorno di una system call viene scritto nel registro EAX.
- **C**. Un programma utente può passare i parametri ad una system call anche copiandoli sullo stack del kernel
- **D**. Se una system call viene invocata da un programma utente, viene eseguita interamente in user mode
- **E**. Nessuna delle precedenti

Domanda 3 Dato il seguente codice, quale tra le seguenti affermazione è *VERA*?

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

int main(int argc, char *argv[]) {
    int status;
    pid_t pid = fork();
    if (pid == -1) return EXIT_FAILURE;

    if (pid == 0) {
        printf("bye!\n");
        _exit(0);
    }

    sleep(10);
    wait(&status);
    return EXIT_SUCCESS;
}
```

- **A**. Il processo figlio non entra mai nello stato *zombie* poiché il padre esegue la `wait`
- **B**. Il processo figlio diventa *orfano* non appena il processo padre termina, e viene quindi *adottato* (cioè diventa figlio di un altro processo) da un processo *antenato* (nell’albero dei processi) del processo padre
- **C**. Il processo figlio entra nello stato *zombie* dopo aver terminato la propria esecuzione e fino al momento in cui il processo padre raccoglie il suo exit status
- **D**. Il processo padre termina sicuramente prima del figlio in virtù della chiamata a `wait`
- **E**. Nessuna delle precedenti

Domanda 4 Dato il seguente codice, selezionare la risposta corretta:

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

void handler(int signo) {
    printf("Catturato segnale\n");
}

int main(int argc, char *argv[]) {
    struct sigaction act = { 0 };
    act.sa_handler = handler;
    if (sigaction(SIGFPE, &act, NULL) == -1){
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    printf("%d\n", 3/0);

    return 0;
}
```

- **A**. Il programma non termina (o termina dopo molto)
- **B**. Il programma termina con un core dump
- **C**. Il programma stampa “Catturato segnale” e poi termina
- **D**. Il programma stampa “inf” e termina
- **E**. Nessuna delle precedenti

Domanda 5 Dato il seguente codice, selezionare la risposta corretta:

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

int main(int argc, char *argv[]) {
    alarm(3);
    pause();
    printf("bye!\n");
    return 0;
}
```

- **A**. Il programma non termina
- **B**. Il programma termina dopo circa 3 secondi stampando “bye!”
- **C**. Il programma termina dopo circa 3 secondi ma non stampa “bye!”
- **D**. Il programma stampa “bye!” ogni 3 secondi
- **E**. Nessuna delle precedenti

Domanda 6 Quale delle seguenti affermazioni è *FALSA*?

- **A**. Il segnale `SIGKILL` non può essere ignorato né catturato
- **B**. Il segnale `SIGSEGV` non può essere catturato
- **C**. Il segnale `SIGCHLD` viene ignorato per default
- **D**. Il segnale `SIGINT`, se non catturato o ignorato, causa la terminazione del processo che lo riceve
- **E**. Il segnale `SIGQUIT` , se non catturato o ignorato, causa la terminazione del processo che lo riceve

Domanda 7 Quale delle seguenti affermazioni è *VERA*?

- **A**. Il comando `kill -9 <pid>` invia il segnale `SIGINT` al processo con pid `<pid>`
- **B**. Il segnale inviato col comando `kill -SIGINT <pid>` non può essere catturato dal processo con pid `<pid>`
- **C**. `alarm(s)` invia il segnale `SIGALRM` ogni `s` secondi
- **D**. `ualarm(t, n)` invia un segnale dopo `t` microsecondi e successivamente ogni `n` microsecondi
- **E**. Nessuna delle precedenti

Soluzioni

Esercizio 1 (prefisso di stringa)

`e1_eq.c`

```c
#include "e1.h"
#include <stdio.h>

int is_suffix(const char* s1, const char* s2){

    const char *a = s1;
    const char *c = s2;

L1:
    if (*a == 0) goto L2;
    a++;
    goto L1;

L2:
    if (*c == 0) goto L3;
    c++;
    goto L2;

L3:
    if (a - s1 == 0) goto E;
    if (c - s2 == 0) goto E;
    a--;
    c--;
    char dl = *c;
    if (dl - *a != 0) goto E;
    goto L3;

E:
    if (s1 != a) goto R;
    if (*a == *c) return 1;

R:
    return 0;
}
```

`e1.s`

```
 .globl is_suffix

 is_suffix:
    movl 4(%esp), %eax
    movl 8(%esp), %ecx

 L1:
    cmpb $0, (%eax)
    je   L2
    incl %eax
    jmp  L1
```

```
L2:
    cmpb $0, (%ecx)
    je   L3
    incl %ecx
    jmp  L2

L3:
    cmpl 4(%esp), %eax
    je   E
    cmpl 8(%esp), %ecx
    je   E
    decl %eax
    decl %ecx
    movb (%ecx), %dl
    cmpb (%eax), %dl
    jne   E
    jmp  L3

E:
    cmpl 4(%esp), %eax
    jne   R0
    movb (%ecx), %dl
    cmpb (%eax), %dl
    jne  R0
    movl $1, %eax
    jmp  R

R0:
    xorl %eax, %eax

R:
    ret
```

Esercizio 2 (indicatore di progresso)

e2.c

```c
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include "e2.h"

int i, max;

static void do_sort(int *v, int n) {
    int j;
    for (i=0; i<n; ++i)
        for (j=1; j<n; ++j)
            if (v[j-1] > v[j]) {
                int tmp = v[j-1];
                v[j-1] = v[j];
                v[j] = tmp;
            }
}

void handler(int sig) {
    printf("%3.1f%%\n", 100.0*i/max);
    ualarm(500000,0);
```

```
}

void sort(int *v, int n) {

    max = n;

    struct sigaction act = { 0 };               // preparazione struttura
    act.sa_handler = handler;                   // gestore segnale
    int ret = sigaction(SIGALRM, &act, NULL);   // gestore installato
    if (ret == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    ualarm(500000,0);

    do_sort(v, n);

    act.sa_handler = SIG_IGN;                   // segnale ignorato
    ret = sigaction(SIGALRM, &act, NULL);       // gestore installato
    if (ret == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
}
```

Esercizio 3 (Domande)

1. **D**. L'interrupt generata dal timer è asincrona
2. **B**. Il valore di ritorno di una system call viene scritto nel registro EAX.
3. **C**. Il processo figlio entra nello stato *zombie* dopo aver terminato la propria esecuzione e fino al momento in cui il processo padre raccoglie il suo exit status

   Spiegazione: quando un processo termina, il kernel libera le risorse associate, ma deve conservarne alcune, come il PCB, per permettere al processo padre di raccogliere l'exit status tramite `wait`/`waitpid`. In questo intervallo di tempo il processo entra in uno stato denominato *zombie*. Quando il padre raccoglie l'exit status il kernel può rilasciare tutte le risorse associate al figlio, che esce dallo stato *zombie* e non compare più nella lista dei processi. Se il padre termina senza raccogliere l'exit status del figlio (con `wait`/`waitpid`) il processo figlio viene adottato da un processo antenato del processo padre (storicamente init, nei sistemi moderni un discendente di init) che raccoglie l'exit status del figlio permettendo così al kernel di liberare tutte le risorse associate (questi processi vengono detti anche *reaper*).
4. **A**. Il programma non termina (o termina dopo molto)

   Spiegazione: la divisione per zero fa generare alla CPU un FAULT. Il kernel invoca il gestore associato alla divisione per zero e manda il segnale SIGFPE al processo che ha generato il fault per terminarlo (il segnale SIGFPE ha come default la terminazione e la generazione di un core dump). Quando c'è un fault il programma riprende l'esecuzione ripetendo l'istruzione in cui si è generato il fault. Per questo motivo, catturando il segnale SIGFPE il programma non termina.
5. **C**. Il programma termina dopo circa 3 secondi ma non stampa "bye!" [il segnale SIGALRM ha come default la terminazione]
6. **B**. Il segnale SIGSEGV non può essere catturato [gli unici segnali a non poter essere catturati sono SIGKILL e SIGSTOP]

7. **D**. `ualarm(t, n)` invia un segnale dopo `t` microsecondi e successivamente ogni `n` microsecondi [vedi man ualarm]