[T12] Esercitazione 12

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 progammazione. 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 (ordinamento a bolle)

Tradurre in IA32 la seguente funzione C `bubble_sort` definita in `E1-bubble-sort/e1.c` che, dato un array di short `v` e la sua lunghezza `n`, ordina l'array usando l'algoritmo di ordinamento a bolle. La funzione richiama un'altra funzione `swap` già fornita in `E1-bubble-sort/e1_main.c`.

`e1.c`

```
void bubble_sort(short *v, unsigned n) {
    unsigned i, again;
    do {
        again = 0;
        for (i=1; i<n; ++i)
            if (v[i-1] > v[i]) {
                swap(&v[i-1], &v[i]);
                again = 1;
            }
    } while(again);
}
```

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

Esercizio 2 (creazione di un file archivio)

Scrivere nel file `E2-archiver/e2.c` una funzione `archiver` che crea un file archivio in cui inserisce un numero arbitrario di file in modo simile a quanto avviene per un file `tar`. La funzione deve avere il seguente prototipo:

```
void archiver(const char* archive, const char** files, int n);
```

dove:

1. `archive` è il pathname del file archivio di output (es. `archive.dat`)
2. `files` è un array di stringhe che rappresentano i pathname dei file di input da archiviare in `archive`
3. `n` è il numero di file da archiviare

La struttura del file archivio deve essere la seguente, dove ogni riga della tabella rappresenta un campo di dati:

```
dimensione in byte | contenuto del campo
-----------------------------------------------------
256                | pathname file 1
8                  | dimensione in byte n1 del file 1
n1                 | contenuto del file 1
-----------------------------------------------------
256                | pathname file 2
8                  | dimensione in byte n2 del file 2
n2                 | contenuto del file 2
-----------------------------------------------------
...
```

Come si vede, ogni file archiviato ha una header formata da: 1) 256 byte che contengono il pathname del file (come stringa C, quindi con terminatore zero alla fine del pathname - non è necessario ovviamente che il pathname effettivo usi tutti i 256 byte disponibili e i byte extra saranno padding) e 2) 8 byte che rappresentano la dimensione del file. Alla header seguono i byte del file stesso.

Se un file con il pathname `archive` esiste già, il suo contenuto deve essere inizialmente troncato a dimensione zero dalla funzione `archiver`. Il file archivio creato deve avere privilegi di lettura e scrittura per l'utente proprietario, sola lettura per il gruppo proprietario, e nessun permesso per tutti gli altri.

Compilare il programma con `make` ed eseguirlo con `./e2`. Il programma genererà un file archivio di output `archive.dat` contenente 5 file.

Suggerimenti:

1. per calcolare la dimensione `size` di un file con descrittore `fd` si può usare `size = lseek(fd, 0, SEEK_END)`. La system call si posiziona alla fine del file e restituisce l'offset corrente che coincide con la dimensione in byte del file stesso.
2. per scopi di debugging, è possibile esplorare il contenuto binario del file di output `archive.dat` creato dal programma di prova con il comando `xxd archive.dat | less`

Esercizio 3 (Domande)

Domanda 1 Si consideri questo frammento di programma:

```
#define N 10
int f(int x, int y) {
    int z = (x*x) + log(y * x * x);
    int w = 3 * N;
    while (!isprime(z)) z++;
    return w + z;
}
```

e la sua versione ottimizzata:

```
#define N 10
int f(int x, int y) {
    int x2 = (x*x);
    int z = x2 + log(y * x2);
    while (!isprime(z)) z++;
    return 30 + z;
}
```

Quali ottimizzazioni sono state applicate? Una sola delle seguenti risposta è corretta.

- **A**. loop unrolling + constant folding + constant propagation
- **B**. register allocation + function inlining + common subexpression elimination
- **C**. constant folding + loop-invariant code motion + constant propagation
- **D**. common subexpression elimination + constant folding + constant propagation
- **E**. Nessuno dei precedenti

Domanda 2 Si consideri una cache associativa a 2 vie con 4 linee da 32 byte ciascuna e politica di rimpiazzo LRU. Potendo scegliere durante un cold cache miss, selezionare sempre la linea con indice più basso. Data la seguente sequenza di accessi a indirizzi di memoria, qual è il contenuto delle linee di cache alla fine della sequenza: `362`, `422`, `261`, `228`, `45`, `200`, `258`, `78`? (l'ordine è importante)

- **A**. 2, 7, 1, 6
- **B**. 8, 2, 7, 1
- **C**. 8, 2, 1, 6
- **D**. 1, 6, 8, 2
- **E**. Nessuno dei precedenti

Domanda 3 Quale delle seguenti affermazioni è *FALSA*?

- **A**. Le system call vengono eseguite in modalità kernel
- **B**. Le system call sono identificate da un numero
- **C**. Nel kernel Linux i parametri delle system call vengono passati attraverso i registri della CPU
- **D**. L'istruzione `int $0x80` viene usata per invocare una system call, dove `$0x80` è il numero della system call
- **E**. Nessuna delle precedenti

Domanda 4 Si consideri un sistema di memoria virtuale con uno spazio di indirizzi a 20 bit, pagine da 64 KB, e la seguente tabella delle pagine: `{0x1, 0x7, 0x0, 0xF, 0xD, 0x8, 0xE, 0xC, 0x6, 0x2, 0xB, 0x3, 0x5, 0x9, 0x4, 0xA}`. A quali indirizzi fisici corrispondono i seguenti indirizzi logici: `0xFF5AB`, `0xC1A01`, `0xDB328`?

- **A**. `0xFF5A3`, `0xC1A07`, `0xDB326`
- **B**. `0xAA5AB`, `0x57A01`, `0x93328`
- **C**. `0xAF5AB`, `0x51A01`, `0x9B328`
- **D**. `0xAA8AB`, `0x57B01`, `0x93F28`
- **E**. La tabella delle pagine non è valida
- **F**. Nessuna delle precedenti

Domanda 5 Si consideri un semplice allocatore di memoria malloc/free dove, potendo scegliere, la malloc riusa il blocco libero con l'indirizzo più basso. Assumere per semplicità che non vi sia alcuna header e che i blocchi allocati vengano arrotondati a una dimensione multiplo di 4 byte. Inoltre, l'allocatore non divide blocchi liberi in più blocchi di dimensione inferiore, né fonde eventuali blocchi liberi adiacenti in un blocco di dimensioni superiore. Qual è il contenuto dell'heap alla fine della seguente sequenza di operazioni (dove `X` denota 4 byte allocati e `.` denota 4 byte liberi)?

```
p1=malloc(7)
p2=malloc(3)
p3=malloc(18)
p4=malloc(1)
free(p2)
free(p1)
p5=malloc(2)
p6=malloc(6)
free(p4)
free(p3)
p7=malloc(3)
```

- **A**. XX | X | ..... | X
- **B**. XX | . | XXXXX | . | XX
- **C**. XX | X | ..... | X | ..
- **D**. XX | . | ..... | X | XX
- **E**. XX | X | XXXXX | . | XX
- **F**. Nessuna delle precedenti

Domanda 6 Quale delle seguenti affermazioni è *VERA*?

- **A**. I segnali possono essere inviati esclusivamente dal kernel, ma possono essere catturati da programmi utente
- **B**. L'interrupt vector è una tabella che contiene i puntatori ai gestori dei segnali
- **C**. Tutti i segnali possono essere catturati e gestiti
- **D**. Il comando `kill` può essere usato solo per inviare il segnale `SIGKILL`
- **E**. Nessuna delle precedenti

Domanda 7 Si consideri una cache completamente associativa con 4 linee da 16 byte ciascuna e politica di rimpiazzo LRU. Potendo scegliere durante un cold cache miss, selezionare sempre la linea con indice più basso. Data la seguente sequenza di accessi a indirizzi di memoria, qual è il contenuto delle linee di cache alla fine della sequenza: 116, 71, 142, 162, 70, 244, 273, 132, 112 ? (l'ordine è importante)

- **A**. 10, 4, 8, 7
- **B**. 4, 15, 17, 7
- **C**. 8, 7, 17, 15
- **D**. 17, 7, 8, 15
- **E**. Nessuno dei precedenti

Domanda 8 Quale delle seguenti affermazioni è *VERA*?

- **A**. Una cache completamente associativa è una cache a k vie con k = 1
- **B**. Un cache miss di tipo "cold" prevede il "rimpiazzo" di una linea di cache
- **C**. Con una cache completamente associativa possono verificarsi cache miss di tipo "conflict"
- **D**. Con una cache ad accesso diretto non è possibile usare la politica di rimpiazzo LRU
- **E**. Nessuna delle precedenti

Soluzioni

Esercizio 1 (ordinamento a bolle)

`e1_eq.c`

```c
#include "e1.h"

void bubble_sort(short *v, unsigned n) {
```

```
    // edi <-> v, esi <-> n, ebx <-> i, ebp <-> again
    unsigned i, again;
L1: again = 0;
    i = 1;
L2: if (i>=n) goto E;
    short tmp1 = v[i-1];
    if (tmp1 <= v[i]) goto F;
    swap(&v[i-1], &v[i]);
    again = 1;
F:  i++;
    goto L2;
E:  if (again) goto L1;
}
```

e1.s

```
.globl bubble_sort

bubble_sort: # void bubble_sort(short *v, unsigned n) {
    # edi <-> v, esi <-> n, ebx <-> i, ecx <-> again
    pushl %edi
    pushl %esi
    pushl %ebx
    subl $8, %esp
    movl 24(%esp), %edi
    movl 28(%esp), %esi
L1: xorl %ecx, %ecx                 # again = 0;
    movl $1, %ebx                   # i = 1;
L2: cmpl %esi, %ebx                 # if (i>=n)
    jae E                           #     goto E;
    movw -2(%edi,%ebx,2), %ax       # short tmp1 = v[i-1];
    cmpw (%edi,%ebx,2), %ax         # if (tmp1 <= v[i])
    jle F                           #     goto F;
    leal -2(%edi,%ebx,2), %eax
    movl %eax, (%esp)
    leal (%edi,%ebx,2), %eax
    movl %eax, 4(%esp)
    call swap                       # swap(&v[i-1], &v[i]);
    movl $1, %ecx                   # again = 1;
F:  incl %ebx                       # i++;
    jmp L2                          # goto L2;
E:  testl %ecx, %ecx                # if (again)
    jne L1                          #     goto L1;
    addl $8, %esp
    popl %ebx
    popl %esi
    popl %edi
    ret
```

Esercizio 2 (creazione di un file archivio)

```
#include "e2.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void check_error(long res, char* msg) {
```

```
    if (res != -1) return;
    perror(msg);
    exit(EXIT_FAILURE);
}

#define FILENAME_LEN 256
#define BUF_SIZE 4096

void copy_file(int fd_dest, int fd_src) {
    ssize_t res;
    char buf[BUF_SIZE];
    for (;;) {
        res = read(fd_src, buf, BUF_SIZE);
        check_error(res, "read");
        if (res == 0) break;
        res = write(fd_dest, buf, res);
        check_error(res, "write");
    }
}

void archiver(const char* archive, const char** files, int n) {
    char filename[FILENAME_LEN];
    int fd_archive, fd, i;
    long res;

    fd_archive = open(archive, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    check_error(fd_archive, "open");

    for (i=0; i<n; ++i) {
        fd = open(files[i], O_RDONLY);
        check_error(fd, "open");

        long size = lseek(fd, 0, SEEK_END);
        check_error(size, "lseek");

        res = lseek(fd, 0, SEEK_SET);
        check_error(res, "lseek");

        strcpy(filename, files[i]);

        res = write(fd_archive, filename, FILENAME_LEN);
        check_error(res, "write");

        res = write(fd_archive, &size, sizeof(size));
        check_error(res, "write");

        copy_file(fd_archive, fd);

        res = close(fd);
        check_error(res, "close");
    }

    res = close(fd_archive);
    check_error(res, "close");
}
```

Esercizio 3 (Domande)

1. **D**. common subexpression elimination + constant folding + constant propagation

   Spiegazione:

- common subexpression elimination: viene precalcolato `x2 = x * x` e poi viene usata `x2` ogni volta che nel codice originale viene usata la sottoespressione `x * x`
- constant folding + constant propagation: l'espressione costante `3 * N` viene precalcolata e sostituita con la costante `30`. A questo punto la variabile `w` contiene una costante nota a tempo di compilazione e il suo valore non viene mai cambiato. Perciò la variabile `w` viene eliminata e il suo valore costante noto propagato in ogni punto in cui `w` compare (in questo caso solo nella return)

2. **B**. 8, 2, 7, 1

   Spiegazione:

```
Indirizzi:              362, 422, 261, 228, 45, 200, 258, 78
Blocchi (indirizzo/32):  11,  13,   8,   7,  1,   6,   8,  2
Cache: (tra parentesi il tempo dall'ultimo accesso a un blocco in cache)
        0       1       2       3
11 -> |       |       | 11(0) |       | (cold miss)
13 -> |       |       | 11(1) | 13(0) | (cold miss)
 8 -> |  8(0) |       | 11(2) | 13(1) | (cold miss)
 7 -> |  8(1) |       |  7(0) | 13(2) | (conflict miss, rimpiazzo LRU)
 1 -> |  8(2) |       |  7(1) |  1(0) | (conflict miss, rimpiazzo LRU)
 6 -> |  8(3) |  6(0) |  7(2) |  1(1) | (cold miss)
 8 -> |  8(0) |  6(1) |  7(3) |  1(2) | (hit)
 2 -> |  8(1) |  2(0) |  7(4) |  1(3) | (capacity miss, rimpiazzo LRU)
```

3. **D**. L'istruzione `int $0x80` viene usata per invocare una system call, dove `$0x80` è il numero della system call

   Spiegazione: `$0x80` non è il numero che identifica la system call, ma il numero che identifica il gestore dell'interrupt numero `$0x80` (quello che invoca le system call). Il numero che identifica la system call viene invece scritto nel registro `%eax`

4. **C**. `0xAF5AB`, `0x51A01`, `0x9B328`

   Spiegazione:

   Con indirizzi di 20 bit e pagine da 64 KB (2^16 byte) si hanno i 4 bit più significativi per il numero di pagina e i 16 bit meno significativi per l'offset. Gli indirizzi possono quindi essere rappresentati con 5 cifre esadecimali in cui la più significativa è il numero di pagina e le restanti 4 l'offset. L'indirizzo fisico si costruisce sostituendo ai bit del numero di pagina i corrispondenti bit del numero di frame. Perciò la cifra esadecimale più significativa degli indirizzi logici verrà usata come indice nella tabella delle pagine per risalire al numero di frame corrispondente. Per ottenere gli indirizzi fisici è sufficiente il numero di frame al numero di pagina.

5. **F**. Nessuna delle precedenti

   Spiegazione:

```
p1 = malloc(7)
XX
p2 = malloc(3)
XX | X
p3 = malloc(18)
XX | X | XXXXX
p4 = malloc(1)
XX | X | XXXXX | X
free(p2)
XX | . | XXXXX | X
free(p1)
```

```
.. | . | XXXXX | X
p5 = malloc(2)
XX | . | XXXXX | X
p6 = malloc(6)
XX | . | XXXXX | X | XX
free(p4)
XX | . | XXXXX | . | XX
free(p3)
XX | . | ..... | . | XX
p7 = malloc(3)
XX | X | ..... | . | XX
```

6. **E.** Nessuna delle precedenti

7. **E**. Nessuno dei precedenti

   Spiegazione:

```
Indirizzi:              116, 71, 142, 162, 70, 244, 273, 132, 112
Blocchi (indirizzo/16):   7,  4,   8,  10,  4,  15,  17,   8,   7
Cache: (tra parentesi il tempo dall'ultimo accesso a un blocco in cache)
        0       1       2       3
 7 -> |  7(0) |       |       |       | (cold miss)
 4 -> |  7(1) |  4(0) |       |       | (cold miss)
 8 -> |  7(2) |  4(1) |  8(0) |       | (cold miss)
10 -> |  7(3) |  4(2) |  8(1) | 10(0) | (cold miss)
 4 -> |  7(4) |  4(0) |  8(2) | 10(1) | (hit)
15 -> | 15(0) |  4(1) |  8(3) | 10(2) | (capacity miss, rimpiazzo LRU)
17 -> | 15(1) |  4(2) | 17(0) | 10(3) | (capacity miss, rimpiazzo LRU)
 8 -> | 15(2) |  4(3) | 17(1) |  8(0) | (capacity miss, rimpiazzo LRU)
 7 -> | 15(3) |  7(0) | 17(2) |  8(1) | (capacity miss, rimpiazzo LRU)
```

8. **D**. Con una cache ad accesso diretto non è possibile usare la politica di rimpiazzo LRU

   Spiegazione:

   Con una cache ad accesso diretta ogni blocco può essere caricato soltanto in una specifica linea di cache. Per tanto ogni volta che avviene un cache miss non è possibile scegliere quale linea di cache rimpiazzare, poiché questa sarà necessariamente l’unica in cui il blocco può essere caricato.