[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 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 (un semplice allocatore di memoria)

L'obiettivo dell'esercizio è implementare l'operazione di deallocazione di un semplice allocatore di memoria. L'operazione `mymalloc` è fornita e si richiede di scrivere l'operazione `myfree` nel file `E1-free/e1.c`. L'operazione deve prendere il blocco deallocato e inserirlo in testa alla lista di blocchi liberi puntata dalla variabile globale `free_list` come descritto nella dispensa del corso. I blocchi liberi dell'allocatore hanno una header di 4 byte che contiene la dimensione del blocco, seguita da un campo `next` come definito nel file `e1.h`:

`e1.h` (estratto)

```
#pragma pack(1)
typedef struct header_t {
    unsigned size;
    struct header_t* next;
} header_t;
```

Si ricorda che la direttiva del compilatore `#pragma pack(1)` evita l'inserimento di padding tra il campo `size` e il campo `next`, che quindi sono contigui in memoria. L'implementazione della `mymalloc` (che non deve essere modificata) è riportata di seguito:

`e1.c`

```
header_t* free_list = NULL;

void* mymalloc(size_t m) {                          // prende size del payload da allo
    header_t *p, *q = NULL;
```

```
    m = ((m+3)/4)*4;                                  // arrotonda al più piccolo multip
    if (m < 8) m = 8;                                 // garantisce spazio per il puntat
    for (p=free_list; p != NULL; q = p, p = p->next) // cerca blocco libero first-fit
        if (p->size >= m) break;
    if (p == NULL) p = sbrk(4 + m);                   // nessun blocco libero, si espand
    else                                              // toglie nodo dalla lista dei blo
        if (q == NULL) free_list = free_list->next;
        else q->next = q->next->next;
    ((header_t*)p)->size = m;                         // size misura il blocco a meno de
    return (char*)p+4;                                // restituisce puntatore al payloa
}

void myfree(void* p) {                                // prende puntatore al payload
    // completare la funzione...
}
```

Si compili il programma con `make`. Se `myfree` è realizzata correttamente, l'output del programma di prova `E1-free/e1_main.c` deve essere il seguente (ovviamente gli indirizzi saranno diversi):

```
--- heap size: 76 byte
1   [0x10289f000][size=16  ][free]
2   [0x10289f014][size=24  ][used]
3   [0x10289f030][size=8   ][free]
4   [0x10289f03c][size=12  ][used]
=== free_list: 0x10289f000
*   [0x10289f000][size=16  ][next=0x10289f030]
*   [0x10289f030][size=8   ][next=0x0]
```

Se l'output è questo (a parte i valori degli indirizzi) inserire 1 nel form di consegna, e 0 altrimenti.

L'output elenca dapprima i blocchi presenti in heap e poi mostra il contenuto della lista di blocchi liberi dopo aver eseguito la seguente sequenza di operazioni:

```
    void* p1 = mymalloc(16);
    void* p2 = mymalloc(22);
    void* p3 = mymalloc(8);
    void* p4 = mymalloc(12);
    myfree(p1);
    myfree(p3);
    void* p5 = mymalloc(4);
    void* p6 = mymalloc(14);
    myfree(p5);
    myfree(p6);
```

Esercizio 2 (ottimizzazioni per la cache)

Si consideri la seguente funzione in `E2-cache/e2.c`:

```
#define STRIDE 64

long f(const short *v, unsigned n){
    long x = 0;
    unsigned i, j;
    for (i=0; i<STRIDE; ++i)
        for (j=0; j<n; j+=STRIDE) x += v[i+j];
    return x;
}
```

Scrivere nel file `E2-cache/e2_opt.c` una versione della funzione semanticamente equivalente che faccia un migliore uso delle cache. Compilare il programma con `make` ed eseguirlo con `./e2`. Il programma riporterà il tempo di esecuzione in millisecondi della versione non ottimizzata e di quella ottimizzata insieme allo speedup ottenuto. Verificare che la versione ottimizzata calcoli lo stesso valore di quella non ottimizzata.

Esercizio 3 (Domande)

Domanda 1 Quale delle seguenti affermazioni è *VERA*?

- **A**. `malloc` è una system call
- **B**. `malloc` alloca tutti blocchi della stessa dimensione
- **C**. `malloc` restituisce indirizzi fisici
- **D**. L'indirizzo passato alla `free` punta al *payload* del blocco allocato da `malloc`
- **E**. Nessuna delle precedenti

Domanda 2 Quale delle seguenti affermazioni è *VERA*?

- **A**. La frammentazione esterna si verifica a causa del padding
- **B**. L'allocatore della memoria dinamica (`malloc`/`free`) non soffre di frammentazione esterna
- **C**. Il meccanismo di paginazione della memoria virtuale non soffre di frammentazione interna, poiché tutte le pagine hanno la stessa dimensione
- **D**. L'area di swap è necessaria per contrastare la frammentazione interna
- **E**. Nessuna delle precedenti

Domanda 3 Quale delle seguenti affermazioni è *FALSA*?

- **A**. Le memorie cache sfruttano i principi di località spaziale e temporale per ridurre i tempi di accesso ai dati.
- **B**. I principi di località spaziale e temporale emergono spesso spontaneamente nel codice dei programmi, ma in alcuni casi il programmatore può riscrivere il codice in modo da favorirli
- **C**. La politica di rimpiazzo LRU prevede di selezionare per il rimpiazzo il blocco nella linea di cache che è stato acceduto meno recentemente
- **D**. In alcune architetture è presente una cache L1 dedicata alle istruzioni e una dedicata ai dati
- **E**. La dimensione di una linea di cache è pari alla dimensione di una pagina della memoria virtuale

Domanda 4 Data la seguente porzione di codice. Quanti cicli di clock vengono richiesti da una semplice pipeline a 5 stadi (Fetch, Decode, Execute, Memory, Write-Back) per completare tutte le istruzioni assumendo che gli hazard vengano risolti con stalli?

```
movl $1, %eax
addl %eax, %ecx
addl $2, %edi
incl %esi
addl %esi, %edx
movl $3, %ebx
```

- **A**. 16
- **B**. 10
- **C**. 13
- **D**. 18
- **E**. 25
- **F**. Nessuno dei precedenti

Domanda 5 Con riferimento alla domanda 4, è possibile ottimizzare il codice riordinando le istruzioni (senza cambiare la semantica) in modo da ridurre il numero di cicli di clock richiesti?

- **A**. No, non è possibile
- **B**. Sì è possibile ridurre il numero di cicli di clock a 12, ma non a meno
- **C**. Sì è possibile ridurre il numero di cicli di clock a 10, ma non a meno
- **D**. Sì è possibile ridurre il numero di cicli di clock a 16, ma non a meno
- **E**. Sì è possibile, ma le risposte B, C e D non sono corrette

Soluzioni

Esercizio 1 (un semplice allocatore di memoria)

e1.c

```c
#include <unistd.h>
#include <stdlib.h>
#include "e1.h"

header_t* free_list = NULL;

void* mymalloc(size_t m) {
    header_t *p, *q = NULL;
    m = ((m+3)/4)*4;                // arrotonda al più piccolo multiplo di 4 maggiore
    if (m < 8) m = 8;               // garantisce spazio per il puntatore next
    for (p=free_list; p != NULL; q = p, p = p->next)    // cerca blocco libero first-fi
        if (p->size >= m) break;
    if (p == NULL) p = sbrk(4 + m); // nessun blocco libero, si espande l'heap
    else                            // toglie nodo dalla lista dei blocchi liberi
        if (q == NULL) free_list = free_list->next;
        else q->next = q->next->next;
    ((header_t*)p)->size = m;       // size misura il blocco a meno del campo size stes
    return (char*)p+4;
}

void myfree(void* p) {
    header_t* q = (header_t*)((char*)p-4);
    q->next = free_list;
    free_list = q;
}
```

Esercizio 2 (ottimizzazioni per la cache)

e2_opt.c

```c
#include "e2.h"

long f_opt(const short *v, unsigned n){
    long x = 0;
    unsigned i;
    for (i=0; i<n; i+=1) x += v[i];
    return x;
}
```

Esercizio 3 (Domande)

1. **D**. L'indirizzo passato alla `free` punta al *payload* del blocco allocato da `malloc`

Spiegazione: alla funzione `free` occorre passare l'indirizzo restituito dalla corrispondente `malloc`, che punta al *payload* del blocco (cioè la parte effettivamente utilizzabile per i dati) e non al header del blocco (che è gestito internamente da `malloc`/`free` in modo totalmente trasparente per il programmatore).

2. **E**. Nessuna delle precedenti

   Spiegazione: Tutte le altre risposte sono false. Infatti:

   - La frammentazione esterna si ha quando si avrebbe spazio libero sufficiente ad accogliere una richiesta di allocazione, ma poiché i blocchi liberi sono frammentati non è possibile accoglierla (ciascun blocco libero non è sufficientemente grande da soddisfare la richiesta di allocazione). Questa non ha niente a che vedere con il padding, perciò l'affermazione A è falsa.
   - L'allocatore `malloc`/`free` soffre di frammentazione esterna. Infatti, la `malloc` deve poter allocare blocchi di dimensioni diverse (dunque la `free` può lasciare buchi di dimensioni diverse) e, per ovvie ragioni, può restituire solo blocchi contigui di memoria. Dunque anche la risposta B è falsa.
   - Il meccanismo di paginazione della memoria, poiché alloca solo blocchi di dimensione fissa, non soffre del problema della frammentazione esterna, tuttavia soffre del problema della frammentazione interna (come tutti gli allocatori che possono allocare blocchi di memoria più grandi di quelli realmente richiesti). Dunque la risposta C è falsa.
   - Infine, l'area di swap permette di allocare più memoria virtuale di quanta ne sia disponibile in memoria fisica (in RAM). Non rappresenta un mezzo per contrastare la frammentazione interna, che invece rimane indipendentemente dall'area di swap. Dunque anche la risposta D è falsa.

3. **E**. La dimensione di una linea di cache è pari alla dimensione di una pagina della memoria virtuale

   Spiegazione: Non c'è alcuna relazione tra la dimensione delle linee di cache e la dimensione delle pagine della memoria virtuale.

4. **A**. 16

   Spiegazione:

```
movl $1, %eax      F D E M W
stallo               * * * * *
stallo                 * * * * *
stallo                   * * * * *
addl %eax, %ecx            F D E M W
addl $2, %edi                F D E M W
incl %esi                      F D E M W
stallo                           * * * * *
stallo                             * * * * *
stallo                               * * * * *
addl %esi, %edx                        F D E M W
movl $3, %ebx                            F D E M W
                   |<===      16 cicli       ===>|
```

5. **C**. Sì è possibile ridurre il numero di cicli di clock a 10, ma non a meno

   Spiegazione:

```
movl $1, %eax      F D E M W
incl %esi            F D E M W
addl $2, %edi          F D E M W
movl $3, %ebx            F D E M W
addl %eax, %ecx            F D E M W
addl %esi, %edx              F D E M W
                   |<=  10 cicli   =>|
```

Si noti che lo stadio M dell'istruzione `movl $1, %eax` non crea hazard strutturale con lo stadio F dell'istruzione `movl %3, %ebx` poiché la prima non impegna attivamente la memoria.