Sistemi di Calcolo (A.A. 2015-2016)
Corso di Laurea in Ingegneria Informatica e Automatica
Sapienza Università di Roma

B

Esame del 18/2/2016 (esonerati) – Durata 1h 30'

Esercizio 1 (allocazione dinamica della memoria)

Si consideri il seguente frammento di programma C:

```
short* alloc(short val, unsigned size) {
    short* v = malloc(size*sizeof(short));
    while (size--) v[size] = val;
    return v;
}
double *x, *y, *z, *w;
x = alloc(3,  2);
y = alloc(7,  3);
z = alloc(10, 1);
w = alloc(10, 1);
free(y);
free(w);
y = alloc(6, 2);
w = alloc(6, 2);
```

Si assuma che l'allocatore parta da un heap inizialmente vuoto e si ricordi che `sizeof(short)==2`. L'allocatore cercherà di minimizzare la dimensione dell'heap tentando di usare lo spazio libero con gli indirizzi più bassi. Se non si riesce a soddisfare una richiesta di allocazione, l'heap verrà espanso del minimo indispensabile. Rispondere alle seguenti domande, motivando le risposte:

1. Come è partizionato l'heap in blocchi liberi/in uso dopo ogni `malloc`/`free`?
2. Si genera frammentazione durante l'esecuzione del programma? Se sì, di che tipo?
3. Quanto è grande l'heap alla fine?

Inserire le risposte nel file `es1.txt`.

Esercizio 2 (memorie cache)

Si consideri la seguente funzione che calcola la somma degli elementi di una lista collegata eseguita su una piattaforma a 64 bit equipaggiata con una cache contenente una singola linea da 64 byte:

```
typedef struct nodo nodo;
struct nodo {
    int   info;
    nodo* next;
};
int somma(const nodo* p) {
    int sum = 0;
    for (; p != NULL; p = p->next) sum += p->info;
    return sum;
}
```

Assumere che le variabili del programma siano tutte tenute in registri della CPU, la lista contenga un milione di nodi e i nodi siano allineati a indirizzi multipli di 16. La posizione dei nodi in memoria non è nota a priori.

1. Quanti operazioni di accesso al sistema di memoria (cache/DRAM) vengono

effettuate dalla funzione?

2. Descrivere uno scenario di caso peggiore in cui la funzione incorre nel **maggior** numero possibile di cache miss. Quanti cache miss e quanti cache hit vengono generati in quello scenario?
3. Descrivere uno scenario di caso migliore in cui la funzione incorre nel **minor** numero possibile di cache miss. Quanti cache miss e quanti cache hit vengono generati in quello scenario?

Inserire le risposte nel file `es2.txt`.

Esercizio 3 (memoria virtuale)

1. Si consideri il seguente programma C eseguito su una piattaforma con memoria virtuale gestita con pagine da 2 KB:

```
int x[1024];
int main() {
    double y[4096];
    int* z = malloc(1000000*sizeof(int));
    return 0;
}
```

Quante pagine occupano, approssimativamente, le sezioni DATA, HEAP e STACK del programma subito prima dell'istruzione `return`? Si assuma che `short` occupi 2 byte, `int` 4 byte e `double` 8 byte.

2. Supponiamo di avere uno spazio logico di 16 GB suddiviso in pagine di 8 KB. Quanti bit occupa l'offset all'interno della pagina e quanti bit occupa il numero di pagina in un indirizzo di memoria logico?
3. Si supponga di avere una tabella delle pagine con celle a 32 bit che occupa 16 MB. Se il sistema di memoria usa pagine da 4 KB, quanto è grande lo spazio virtuale? Mostrare i calcoli effettuati.

Inserire le risposte nel file `es3.txt`.

Esercizio 4 (ottimizzazione di programmi)

Si crei nel file `es4-opt.c` una versione ottimizzata manualmente del seguente modulo `es4.c`, dove la funzione `sum` calcola la somma degli elementi di una matrice rappresentata in formato row-major[1]:

```
#include "es4.h"

int idx(int i, int j, int n) {
    return i+j*n;
}

int sum(unsigned char* v, int n) {
    int s = 0, i, j;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            s += v[idx(i,j,n)];
    return s;
}
```

---

[1] Una matrice in formato **row-major** è un array monodimensionale in cui le righe sono disposte consecutivamente in memoria.

Compilare due versioni del programma, **usando `gcc` a 32 bit con livello di ottimizzazione O1** e lo stesso modulo `es4-main.c`:

1. Non ottimizzata: eseguibile `es4`.
2. Ottimizzata: eseguibile `es4-opt`.

Ai fini dell'ottimizzazione:

1. Usare `gprof` per identificare le porzioni più onerose computazionalmente. Chiamare gli eseguibili usati per la profilazione `es4-pg` e `es4-opt-pg`. Salvare i report di `gprof` nei file `es4.txt` e `es4-opt.txt`, rispettivamente
2. Esaminare il modulo assembly `es4.s` fornito (generato a partire da `es4.c` con `gcc –m32 –S -O1`) per capire quali ottimizzazioni siano già state effettuate dal compilatore.

Rispondere alle seguenti domande:

1. Descrivere le ottimizzazioni applicate e dire perché si ritiene che siano efficaci.
2. Riportare il tempo di esecuzione di `es4` e di `es4-opt` usando il comando `time`.
3. Riportare i flat profile delle due versioni usando `gprof`.
4. Di quante volte è più veloce l'eseguibile `es4-opt` rispetto a `es4`?
5. Usando i dati del profilo `es4.txt`, calcolare lo speedup massimo che si può ottenere ottimizzando la funzione `sum`. Motivare la risposta.

Inserire le risposte nel file `es4.txt`. Alla fine del compito, **non eliminare i seguenti file**:

- `es4`
- `es4-pg`
- `es4.txt`
- `es4-opt`
- `es4-opt-pg`
- `es4-opt.txt`
- `gmon.out`