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

C

Esame del 20/1/2016 (esonerati) – Durata 1h 30'

Esercizio 1 (gerarchie di memoria)

Si consideri la seguente funzione che opera su un vettore di `n` interi:

```
int vec_sum(int* v, int n) {
    int sum = 0;
    while (n-- > 0) sum += v[n];
    return sum;
}
```

Assumere che: 1) le variabili del programma siano tutte tenute in registri della CPU; 2) le funzioni siano eseguite su un sistema con una cache da 256 KB avente linee da 64 byte; 3) l'array sia allineato a un indirizzo multiplo di 32; 4) `n` = 1000000; 4) `sizeof(int)==4`.

1. Quanti accessi a memoria vengono effettuati dalla funzione?
2. Quanti cache miss vengono generati? Illustrare il procedimento usato per il calcolo.
3. La dimensione della linea conta? E quella della cache?

Inserire le risposte nel file `es1.txt`.

Esercizio 2 (allocazione dinamica della memoria)

Si consideri il seguente frammento di programma C eseguito su una piattaforma a 64 bit:

```
typedef struct nodo nodo;
struct nodo {
    int   info;
    nodo* next;
};

nodo *l = NULL, *p = NULL;
l = add(l, 10);             // l = malloc(sizeof(nodo))
l = add(l, 20);             // l = malloc(sizeof(nodo))
l = add(l, 30);             // l = malloc(sizeof(nodo))
while (l != NULL) {
    p = add(p, l->info);    // p = malloc(sizeof(nodo))
    l = remove(l);          // free(l)
}
```

Dove:

- `add(l,x)` crea un nuovo nodo con info `x` usando `malloc(sizeof(node))`, lo mette come primo nodo della lista `l`, e restituisce la lista risultante.
- `remove(l)` toglie il primo nodo della lista `l`, lo dealloca con `free`, e restituisce la lista risultante.

Si assuma che l'allocatore parta da un heap inizialmente vuoto. 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:

1. Come è partizionato l'heap in blocchi liberi/in uso dopo ogni `malloc`/`free`?

2. Si genera frammentazione durante l'allocazione? Se sì, di che tipo?
3. Quanto è grande l'heap alla fine?

Inserire le risposte nel file `es2.txt`.

Esercizio 3 (analisi e ottimizzazione del costo dei programmi)

1. Si consideri il seguente modulo C:

```
int f(int v[], int n, int q) {
    int i, s = 0;
    for (i=0; i<n; i++) s += v[q*i];
    return s;
}
```

Il file `es3.s` riporta il codice x86 generato da `gcc 4.8.4` con livello di ottimizzazione 1. Che ottimizzazioni sono state effettuate dal compilatore? Riportare frammenti di codice IA32 a giustificazione delle affermazioni fatte.

2. Come vi aspettate che un compilatore ottimizzato traduca la seguente istruzione in codice IA32 in modo da ridurre al minimo il tempo di esecuzione? Assumere che le variabili siano tenute in registri.

```
a = 4*b+c-19;
```

3. Si considerino i seguenti due frammenti di programma C, assumendo che la matrice `c` sia inizializzata a zero:

Versione 1	**Versione 2**

`for (i=0; i<n; i++)`
`  for (j=0; j<n; j++)`
`  for (k=0; k<n; k++)`
`  c[i][j] += a[i][k]*b[k][j];`
	`for (k=0; k<n; k++)`
	`  for (i=0; i<n; i++)`
	`  for (j=0; j<n; j++)`
	`  c[i][j] += a[i][k]*b[k][j];`

Quale versione ci aspettiamo che sia più veloce? Quale tecnica di ottimizzazione è stata applicata per ottenere una versione dall'altra?

Inserire le risposte nel file `es3.txt`.

Esercizio 4 (ottimizzazione di programmi)

Si crei nel file `es4-opt.c` una versione ottimizzata del seguente modulo `es4.c`:

```
#include "es4.h"

void filtro4(unsigned* in, unsigned* out, int n) {
    int i;
    for (i=0; i+3 < n; i++) out[i] = media(in+i, 4);
    for (;i<n; i++) out[i] = in[i];
}
```

Compilare due versioni del programma, usando `gcc` **a 32 bit** con livello di ottimizzazione 1 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 nelle due versioni. Chiamare gli eseguibili usati per la profilazione `es4-pg` e `es4-opt-pg`.

Salvare i report di `gprof` nei file `es4.txt` ed `es4-opt.txt`, rispettivamente.
2. Esaminare il modulo assembly `es4.s` generato a partire da `es4.c` con `gcc –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 che bisognerebbe ottenere per la funzione `filtro4` (considerandone il tempo complessivo self+children) per ottenere uno speedup totale per l'intero programma pari a 10x, se questo è possibile. 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`