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

A

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

Esercizio 1 (gerarchie di memoria)

Si considerino le seguenti due varianti di una funzione che calcola il prodotto scalare di due vettori di `n double` eseguito su una cache senza vincoli di associativà con almeno 2 linee da 64 byte:

```
typedef struct {
    double x, y;
} coppia;

double scalare1(double* A, double* B, int n) {
    double sum = 0.0; int i;
    for (i=0; i<n; i++) sum += A[i]*B[i];
    return sum;
}

double scalare2(coppia* AB, int n) {
    double sum = 0.0; int i;
    for (i=0; i<n; i++) sum += AB[i].x * AB[i].y;
    return sum;
}
```

Assumere che le variabili del programma siano tutte tenute in registri della CPU, gli array partano da indirizzi multipli di 64, `n` = 1000000 e `sizeof(double)==8`.

1. Quanti accessi a memoria vengono effettuati nel ciclo da ciascuna versione?
2. Mostrare come calcolare i cache miss generati dalle due versioni della funzione. Che numeri si ottengono?
3. Se la cache contenesse una sola linea i risultati cambierebbero? Perché?

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 32 bit:

```
typedef struct {
    double x, y;
} punto;
int i, n = 3;
punto** v = malloc(n*sizeof(punto*));
for (i=0; i<n; i++) v[i] = malloc(sizeof(punto));
for (i=0; i<n; i++) {
    punto* p = malloc(2*sizeof(punto));
    memcpy(p, v[i], sizeof(punto));
    free(v[i]);
    v[i] = p;
}
```

Si assuma che l'allocatore parta da un heap inizialmente vuoto e si ricordi che `sizeof(double)==8`. 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 (memoria virtuale)

1. Supponiamo di avere uno spazio logico di 1 GB suddiviso in pagine di 256 KB. Quanti byte occuperebbe una tabella delle pagine che dovesse indicizzare uno spazio fisico di 4 GB? Per motivi di allineamento, assumere che le entry della tabella delle pagine siano di dimensione multipla di 32 bit.
2. Si consideri un processo 1 che esegue la funzione `produci`, e un processo 2 che successivamente esegue la funzione `consuma`:

Processo 1	**Processo 2**

`void produce(char* s) {`
`  strcpy(s, "Obi-Wan Kenobi");`
`}`
	`void consuma(const char* s) {`
	`  printf("%s\n", s);`
	`}`

Vorremmo che il processo 2 stampasse la stringa precedentemente scritta dal processo 1: questo può succedere usando un sistema di memoria virtuale paginato? Se sì, come?
3. Il seguente testo contiene vari errori: "*In un sistema di memoria paginato, i processi vedono indirizzi nello spazio della memoria fisica, che è suddivista in frame. Un array chiamato tabella delle pagine, globale a tutti i processi, permette di mappare gli indici dei frame negli indici delle pagine corrispondenti. Uno dei vantaggio principali di un sistema paginato è quello di risolvere completamente il problema della frammentazione esterna nell'allocazione della memoria ai processi, facendo in modo che tutto lo spazio allocato venga effettivamente utilizzato*". Correggere gli errori scrivendo il testo corretto.

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"

// conta punti inclusi nella circonferenza passante per l'origine
// centrata nel punto p
int count(int* x, int* y, int p, int n) {
    int i, c;
    for (i = c = 0; i < n; i++)
        if (distsqr(x[p], y[p], x[i], y[i]) <
            distsqr(x[p], y[p], 0, 0)) c++;
    return c;
}

// conta il massimo numero di punti che sono inclusi nella
// circonferenza passante per l'origine centrata in uno dei
// punti dell'insieme
int count_max(int* x, int* y, int n) {
    int i, max = 0;
    for (i=0; i<n; i++)
        if (count(x, y, i, n) > max) max = count(x, y, i, n);
    return max;
}
```

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. 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` 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 massimo che si può ottenere ottimizzando la funzione `count`. 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`