5/6 cfu

Ingegneria degli Algoritmi (A.A. 2011-2012)

Corsi di Laurea in Ingegneria Informatica e Automatica, Ingegneria dei Sistemi Informatici, e Laurea Magistrale in Ingegneria Informatica

Sapienza Università di Roma

Quarto appello [straordinario] (14/11/2012) – Durata 2 ore – 5/6 cfu

Cognome: _____ Nome: _____ Matricola: _____	Autorizzo la pubblicazione del voto di questo esame sul sito web http://www.dis.uniroma1.it/~demetres/didattica/ae, secondo quanto prevede il decreto legislativo 196/2003 (codice in materia di protezione dei dati personali) che dichiaro di conoscere. In fede, _____

Rispondere alle seguenti domande, **motivando le risposte** (risposte non motivate saranno considerate nulle).

Domanda 1 [7 punti]

Una *coda con priorità* è un tipo di dato che permette di mantenere una collezione di chiavi soggetta alle seguenti operazioni:

- `insert(x)`: aggiunge la chiave `x` alla collezione;
- `findMin()`: restituisce la chiave più piccola della collezione;
- `deleteMin()`: elimina la chiave più piccola dalla collezione.

Si consideri la seguente implementazione di una coda con priorità. La struttura dati proposta usa una lista collegata semplice `l` e un heap binario `h`, inizialmente vuoti. Le operazioni sono realizzate come segue:

- `insert(x)`: aggiunge la chiave `x` come primo elemento della lista `l`;
- `findMin()`: elimina una alla volta tutte le chiavi dalla lista `l` e le inserisce nell'heap `h`, poi restituisce il minimo nell'heap `h`;
- `deleteMin()`: elimina una alla volta tutte le chiavi dalla lista `l` e le inserisce nell'heap `h`, poi elimina il minimo dall'heap `h`.

Ricordando che, nel caso peggiore, inserire/togliere una chiave da un heap binario con n nodi richiede tempo O(log n) e trovare il minimo richiede tempo O(1), studiare per ciascuna operazione della struttura dati proposta:

1. il tempo richiesto nel caso peggiore dall'operazione;
2. il tempo ammortizzato richiesto dall'operazione.

Per calcolare il tempo ammortizzato, usare il metodo dei crediti.

Domanda 2 [6 punti]

Scrivere una funzione C che, date due stringhe `a` e `b`, restituisce in `c` una nuova stringa allocata dinamicamente ottenuta prendendo alternativamente un carattere da `a` e uno da `b`. Se una delle due stringhe finisce prima dell'altra, si prenderanno i caratteri da quella con caratteri rimanenti. La funzione deve avere il seguente prototipo:

```
void interleave(const char* a, const char* b, char** c);
```

Ad esempio, il seguente frammento di programma dovrebbe stampare `"hbaumrger"`:

```
char* c;
interleave("ham", "burger", &c);
prinf("%s\n", c);
free(c);
```

Domanda 3 [7 punti]

Si discutano i seguenti meccanismi di base degli allocatori di memoria: allineamento e padding, header di blocco, bit stealing, liste di blocchi, boundary tag, lookup table.

Domanda 4 [6 punti]

Tradurre in C il seguente frammento di codice x86-64:

```
_f:
      pushq %rbp
      movq  %rsp, %rbp
      movq  %rsi, %rcx
      subq  %rdi, %rcx
      movq  %rdi, %rax
      subq  %rsi, %rax
      cmpq  %rsi, %rdi
      cmovleq %rcx, %rax
      popq  %rbp
      ret
```

Si consulti la tabella alla fine del compito tenendo presente che l'assegnamento condizionale `cmovleq C,D` (move if less or equal) preceduto dal test `cmpq A,B` assegna `C` a `D` se `A ≤ B`.

Domanda 5 [6 punti]

Si descriva la legge di Amdahl per programmi non paralleli, mostrando come ricavarla e applicandola a un esempio a scelta, e discutendo la sua importanza nell'ambito del performance profiling dei programmi.

prefix	description	example	C analog
`add`	add source to destination	`addl $5,%ecx`	`ecx += 5`
`call`	procedure call	`call _foo`	`foo()`
`cltq`	sign-extend `eax` to `rax`	–	–
`dec`	decrement destination	`decq %rcx`	`rcx--`
`imul`	multiply destination with source	`imull %esi,%eax`	`eax *= esi`
`inc`	increment destination	`incl %ecx`	`ecx++`
`ja`	jump if above *(unsigned comparison)*	`cmpl %eax,%ebx` `ja L2`	`if ((unsigned)eax > (unsigned)ebx) goto L2`
`jae`	jump if above or equal *(unsigned comparison)*	`cmpl %eax,%ebx` `jae L2`	`if ((unsigned)eax >= (unsigned)ebx) goto L2`
`jb`	jump if below *(unsigned comparison)*	`cmpl %eax,%ebx` `jb L2`	`if ((unsigned)eax < (unsigned)ebx) goto L2`
`jbe`	jump if below or equal *(unsigned comparison)*	`cmpl %eax,%ebx` `jbe L2`	`if ((unsigned)eax <= (unsigned)ebx) goto L2`
`je`	jump if equal	`cmpq %rax,%rbx` `je L2`	`if (rax == rbx)` `goto L2`
`jg`	jump if greater *(signed comparison)*	`cmpq %rax,%rbx` `jg L2`	`if (rax > rbx)` `goto L2`
`jge`	jump if greater or equal *(signed comparison)*	`cmpq %rax,%rbx` `jge L2`	`if (rax >= rbx)` `goto L2`
`jl`	jump if less *(signed comparison)*	`cmpq %rax,%rbx` `jl L2`	`if (rax < rbx)` `goto L2`
`jle`	jump if less or equal *(signed comparison)*	`cmpq %rax,%rbx` `jle L2`	`if (rax <= rbx)` `goto L2`
`jmp`	unconditional jump	`jmp L2`	`goto L2`
`jne`	jump if not equal	`cmpq %rax,%rbx` `jne L2`	`if (rax != rbx)` `goto L2`
`jnz`	identical to `jne`	–	–
`jz`	identical to `je`	–	–
`lea`	copy address to destination (load effective address)	`leaq -12(%rax),%rcx`	`rcx=rax-12`
`leave`	pop the current stack frame, and restore the caller's frame	–	–
`mov`	copy data from source to destination	`movq $7,(%rax)`	`*(long*)rax=7`
`movabs`	copy 64-bit immediate to destination register	`movabsq $-7,%rax`	`rax=-7`
`movsl`	copy sign-extended word to destination register	`movslq %bx,%rax`	`rax=bx`
`movzl`	copy zero-extended word to destination register	`movzlw %bx,%eax`	`eax=(unsigned)bx`
`movsb`	copy sign-extended byte to destination register	`movsbq %bl,%rax`	`rax=bl`
`movzb`	copy zero-extended byte to destination register	`movzbq %bl,%rax`	`rax=(unsigned)bl`
`pop`	pop value from stack and write it to destination	`popq %rbx`	–
`push`	push value on stack	`pushq %rbx`	–
`ret`	return from procedure call	`ret`	`return`
`sub`	subtract source from destination	`subl $5,%ecx`	`ecx -= 5`

Tabella 1: Istruzioni x86-64 più comunemente utilizzate.