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

Secondo appello (13/07/2012) – 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]

Si discutano le convenzioni di chiamata a funzione del System V AMD64 ABI. Illustrare in particolare:

1) il meccanismo di passaggio dei parametri;
2) il layout di uno stack frame;
3) la creazione e la distruzione dello stack frame.

Domanda 2 [6 punti]

Scrivere una funzione C che crea una matrice di interi di dimensione m x n allocata dinamicamente. La funzione deve avere il seguente prototipo:

```
int** alloc_matrix(int m, int n);
```

Domanda 3 [7 punti]

Si consideri un'operazione `add(x)` che inserisce un elemento `x` in un array allocato dinamicamente. Usando il metodo dei crediti, calcolare il costo ammortizzato $T_{am}^{add}(n)$ dell'operazione `add()`, assumento che l'operazione incrementa la dimensione dell'array espandendolo mediante riallocazione quando non c'è più spazio. Analizzare i seguenti due possibili scenari:

1) l'array triplica la propria dimensione quando viene riallocato;
2) l'array diventa di 1000 celle più grande quando viene riallocato.

Domanda 4 [6 punti]

Tradurre in C il seguente frammento di codice x86-64 (si consulti la tabella alla fine del testo di esame):

```
_f:
     pushq %rbp
     movq  %rsp, %rbp
     cmpq  %rsi, %rdi
     jge   L1
     movq  %rdi, %rax
     jmp   L2
L1:
     movq  %rsi, %rax
L2:
     popq  %rbp
     ret
```

Domanda 5 [6 punti]

Si consideri un programma che richiede di allocare solo ed esclusivamente blocchi di dimensione pari a una potenza di due. Che allocatore usereste? E se invece i blocchi fossero tutti di piccole dimensioni, ad esempio inferiori a 64 byte? Motivare la risposta descrivendo la struttura e il funzionamento degli allocatori in questione.

Domanda 6 (facoltativa) [3 punti]

Si illustri il paradigma di calcolo `MapReduce`, discutendone vantaggi e motivazioni.

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.