Exam in Algorithm Implementation 2009-01-13

Inga hjälpmedel!

Examinator: Jonas Skeppstedt, tel 0767 888 124

Approximately 30 out of 60 points are needed to pass the exam.

1. (10p) Describe instruction execution in a five-stage pipelined RISC processor. The RISC acronym stands for reduced instruction set computer. What is meant by ”reduced” and why is that a good idea?

   Answer *Five stage pipeline — see slides. reduced implementation complexity as opposed to number of instructions which may be large without any problem. With simple instructions, pipelining is easier to implement.*

2. (10p) Assume a friend has an array of structs where each struct has two struct members:

```
#define    N     (10000000)

struct {
        double     a;
        double     b;
}       data[N];
```

   and processes this data many times on a computer with a 32 byte cache block size. Due to the nature of the program, it turns out that every time each struct is processed only one of `a` or `b` is accessed, for instance as:

```
void process_a(double x)
{
        int     i;

        for (i = 0; i < N; ++i)
                data[i].a = data[i].a + x;
}
```

   Your friend asks you if you think there might be many cache misses and if there is a simple way to change the source code so the program becomes faster? What do you say?

   Answer *By making two separate arrays of doubles, the locality is significantly increased, since only data that will be used is actually fetched.*

3. (2p) What is bad with the following program?

```
#include <assert.h>
#include <stdio.h>

FILE* openfile(char* name)
{
        FILE*   fp;

        assert((fp = fopen(name, "r")) != NULL);

        return fp;
}
```

Answer *When the macro* `NDEBUG` *is defined (in order to avoid runtime-checking overhead) the expression in the argument to* `assert` *will not be evaluated. Therefore the file will not be opened and the program will encounter undefined behaviour and probably crash.*

4. (8p) Memory allocation. What is an arena ? Compare it with using

   (a) `alloca`,

   (b) `malloc`,

   Which are the important differences in terms of

   (a) C99 standard support,

   (b) execution time for allocation,

   (c) execution time for deallocation, and

   (d) limitations in how the allocated memory can be used?

   Answer *An arena is a region memory from which smaller blocks are allocated. When the programmer knows that no such small block is needed any longer, the entire arena can be deallocated. Allocating memory from an arena is very fast, essentially just checking the arena has enough memory left for the allocation, checking alignment, and then moving a pointer. Since deallocation is done for the whole arena, no time is wasted for deallocating individual blocks. There is no restriction on how the data may be used.*

   `alloca` *is not Standard C, but available on all UNIX systems and other systems as well. The compiler allocates memory by moving the stack pointer which is very fast. No time is needed for deallocation since that is done when the function returns and deallocates its stack frame. Since the memory is allocated from the stack frame, it cannot be returned from the allocating function since*

the lifetime of the data ends with that return. No runtime check is performed to test whether the allocation is possible or not.

`malloc` *is Standard C and both allocation and deallocation takes time. There is no restriction on how the data may be used.*

5. *(10p) Declare an appropriate type and implement functions to create, destruct, print, and add two-dimensional matrices of type int. The function* `init_matrix` *initialises a matrix with random values but you don't have to write it. Your* `new_matrix` *function should set the values to zero.*

```
#include <stdio.h>

int main(int argc, char** argv)
{
        matrix_t*       a;
        matrix_t*       b;
        matrix_t*       c;
        int             rows;
        int             cols;

        sscanf(argv[1], "%d", &rows);
        sscanf(argv[2], "%d", &cols);

        a = new_matrix(rows, cols);
        b = new_matrix(rows, cols);

        init_matrix(a);     // you don't have to write init_matrix
        init_matrix(b);     // you don't have to write init_matrix

        print_matrix(a);
        print_matrix(b);

        c = add(a, b);

        print_matrix(c);

        free_matrix(a);
        free_matrix(b);
        free_matrix(c);
}
```

Answer *A sample solution is:*

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
        size_t          rows;
        size_t          cols;
        int**           matrix;
} matrix_t;

void error(const char* msg)
{
        fprintf(stderr, "fatal error: %s\n", msg);
        exit(1);
}

matrix_t* new_matrix(size_t rows, size_t cols)
{
        matrix_t*       m;
        size_t          i;

        m = malloc(sizeof *m);
        if (m == NULL)
                error("out of memory");
        m->rows = rows;
        m->cols = cols;
        m->matrix = malloc(rows * sizeof(int*));
        if (m->matrix == NULL)
                error("out of memory");
        for (i = 0; i < rows; ++i) {
                m->matrix[i] = calloc(cols, sizeof(int));
                if (m->matrix[i] == NULL)
                        error("out of memory");
        }

        return m;
}

void free_matrix(matrix_t* m)
{
        size_t          i;

        for (i = 0; i < m->rows; ++i)
                free(m->matrix[i]);
        free(m->matrix);
        free(m);
}
```

```
void init_matrix(matrix_t* m)
{
        size_t          i;
        size_t          j;

        for (i = 0; i < m->rows; ++i)
                for (j = 0; j < m->cols; ++j)
                        m->matrix[i][j] = rand();
}

void print_matrix(matrix_t* m)
{
        size_t          i;
        size_t          j;

        for (i = 0; i < m->rows; ++i)
                for (j = 0; j < m->cols; ++j)
                        printf("m[%8zu][%8zu] = %d\n", i, j, m->matrix[i][j]);
}

matrix_t* add(matrix_t* a, matrix_t* b)
{
        size_t          i;
        size_t          j;
        matrix_t*       c;

        c = new_matrix(a->rows, a->cols);
        for (i = 0; i < c->rows; ++i)
                for (j = 0; j < c->cols; ++j)
                        c->matrix[i][j] = a->matrix[i][j] + b->matrix[i][j];
        return c;
}
```

6. (10p) Rewrite the following program so that it becomes as fast as possible. Assume that your compiler cannot perform any optimisation except register allocation and that the cost of multiplication and divide is 4 and 30 clock cycles, respectively, and that your machine is a simple pipelined five-stage RISC processor.

```
void work(int s, int n, unsigned int a[], unsigned int b[], unsigned int c[])
{
        int     i;

        for (i = 1; i < n; i++)
                a[i] = b[i] * 16 + b[i+1] * 15 + c[s * i] / 8;
```

```
}
```

Answer *Below we have used the* **register** *storage class specifier but that is not really needed, since the compiler was supposed to do register allocation itself. Measuring performance is easier this way, however, and was done on a 2.5 GHz Power Quad G5 machine with GCC 4.4.4. Although GCC uses shift instead of divide, you should do that anyway in this question. The timings below refer to the different versions of the function distributed in the file* `work.c`.

version	`-O0`	`-O5`
f0	*0.97 s*	*0.32 s*
f1	*0.70 s*	*0.33 s*
f2	*0.66 s*	*0.24 s*
f3	*0.67 s*	*0.24 s*
f4	*0.42 s*	*0.24 s*
f5	*0.39 s*	*0.23 s*
f6	*0.37 s*	*0.24 s*
f7	*0.35 s*	*0.21 s*