Exam in EDAF15 Algorithm Implementation

May 31, 2014, 8-13

Inga hjälpmedel!

Examiner: Jonas Skeppstedt

30 out of 60p are needed to pass the exam.

1. (10p) Pipelining

   (a) (1p) Why is it important to design a pipeline so that the different pipeline stages need approximately the same time to perform their work?

   (b) (3p) Consider a simple 5-stage pipelined RISC processor. Which pipeline stages are useful and what does each do?

   (c) (5p) Explain what is needed in a superscalar processor to be able to efficiently execute multiple instructions each clock cycle. Why?

   Answer *See book.*

2. (10p) Cache Memories

   (a) (4p) Explain what temporal and spatial locality means. Give examples of when this can be exploited when executing a program.

   (b) (2p) What is the purpose of having an N-way associative cache instead of a direct mapped cache? What is the additional hardware cost?

   (c) (4p) In C programs with nested for-loops and matrices it is sometimes useful to try to order the loops to improve performance. Why and what is the goal?

   Answer *See book.*

3. (10p) Implement a circular double linked list where an empty list is represented by `NULL` and the `data` pointers in multiple nodes can have the same value, and the functions declared below.

```
#include <stdlib.h>
typedef struct list_t   list_t;
struct list_t {
        void            data;
        list_t*         succ;
        list_t*         pred;
};

/* create a new list node with this data. */
list_t* new_list(void* data);

/* deallocate entire list but not any data pointer. */
void free_list(list_t* list);

/* return the number of nodes in the list. */
size_t length(list_t* list);

/* insert data first in the list. */
void insert_first(list_t** list, void* data);
```

```
/* insert data last in the list. */
void insert_last(list_t** list, void* data);

/* if the list is empty, return NULL, otherwise
 * remove (and free) the first node in the list
 * and return its data pointer.
 */
void* remove_first(list_t** list);

/* Allocate and return a pointer to an array with the
 * contents (data pointers) of the list, and write
 * the length of the list in the variable pointed
 * to by size. If the size is zero, NULL should be
 * returned.
 *
 * Note: the word array above is used in the sense that
 * memory should be allocated for a number of elements
 * in contiguous memory locations and not as in array
 * declaration.
 *
 */
void** list_to_array(list_t* list, size_t* size);
```

Answer *See code below.*

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

typedef struct list_t   list_t;
struct list_t {
        void*           data;
        list_t*         succ;
        list_t*         pred;
};

void* xmalloc(size_t size)
{
        void*   p = malloc(size);

        if (p == NULL) {
                fprintf(stderr, "out of memory\n");
                exit(1);
        }
        return p;
}

/* create a new list node with this data. */
list_t* new_list(void* data)
{
        list_t*         p;

        p = xmalloc(sizeof(list_t));
        p->succ = p->pred = p;
        p->data = data;
```

```
        return p;
}

/* deallocate entire list but not any data pointer. */
void free_list(list_t* list)
{
        list_t*         p;
        list_t*         q;

        if (list == NULL)
                return;

        list->pred->succ = NULL;
        p = list;
        while (p != NULL) {
                q = p->succ;
                free(p);
                p = q;
        }
}

/* return the number of nodes in the list. */
size_t length(list_t* list)
{
        size_t          n;
        list_t*         p;

        if (list == NULL)
                return 0;
        n = 0;
        p = list;
        do {
                n += 1;
                p = p->succ;
        } while (p != list);

        return n;
}

static void insert(list_t** list, void* data)
{
        list_t*         h;
        list_t*         p;

        p = new_list(data);

        h = *list;

        if (h != NULL) {
                h->pred->succ = p;
                p->pred = h->pred;
                p->succ = h;
                h->pred = p;
```

```
        } else
                *list = p;
}

/* insert data first in the list. */
void insert_first(list_t** list, void* data)
{
        insert(list, data);

        *list = (*list)->pred;
}

/* insert data last in the list. */
void insert_last(list_t** list, void* data)
{
        insert(list, data);
}

/* if the list is empty, return NULL, otherwise
 * remove (and free) the first node in the list
 * and return its data pointer.
 */
void* remove_first(list_t** list)
{
        list_t*         h;
        void*           data;

        h = *list;

        if (h == NULL)
                return NULL;

        data = h->data;

        if (h == h->succ)
                *list = NULL;
        else {
                *list = h->succ;

                h->pred->succ = h->succ;
                h->succ->pred = h->pred;

                free(h);
        }

        return data;
}

/* Allocate and return a pointer to an array with the
 * contents (data pointers) of the list, and write
 * the length of the list in the variable pointed
 * to by size. If the size is zero, NULL should be
 * returned.
 *
```

```
 * Note: the word array above is used in the sense that
 * memory should be allocated for a number of elements
 * in contiguous memory locations and not as in array
 * declaration.
 *
 */
void** list_to_array(list_t* list, size_t* size)
{
        size_t          n;
        size_t          i;
        void**          a;

        n = *size = length(list);

        if (n == 0)
                return NULL;

        a = xmalloc(n * sizeof(void*));

        for (i = 0; i < n; ++i) {
                a[i] = list->data;
                list = list->succ;
        }

        return a;
}
```

4. (10p) Explain briefly (= in one sentence and at most three lines) the following:

   (a) (1p) `alloca`

   (b) (1p) `gprof`

   (c) (1p) `oprofile`

   (d) (1p) `gcov`

   (e) (1p) `cachegrind`

   (f) (1p) `valgrind`

   (g) (1p) compiler SIMD vectorization

   (h) (1p) memory leak

   (i) (1p) free-list (to avoid malloc/free)

   (j) (1p) reference implementation (for code tuning)

   Answer *See book.*

5. (5p) Explain a method for writing fast C programs. Your answer should **not** be a catalogue of clever tricks but an explanation of what you actually would do in the general case.

6. (3p) Assume we have a negative number in two's complement form and it is represented with 8 bits. How is the same value represented with 16 bits, and why?

   Answer $2^{16} - Y$*, 8 one bits followed by the original bits. See book why this is correct.*

7. (2p) What is the difference between arithmetic and logical right shift? Whether the C operator >> is arithmetic or logical is implementation defined, i.e. the compiler writer decides which to use and must document the choice. Instead of relying on finding the documentation, write a C program that determines which shift is used and prints what it finds.

   Answer *See book and code below.*

```c
#include <stdio.h>

int main(void)
{
        if ((-1 >> 1) < 0)
                puts("arithmetic");
        else
                puts("logical");
}
```

8. (5p) Write an efficient function to compute $a/2^b$ for the type `int`.

   Answer *See code below.*

```c
int divide(int a, int b)
{
        int     q;

        q = a >> b;

        if (a < 0 && (a & ((1 << b) - 1)) != 0)
                q += 1;

        return q;
}
```

9. (5p) Write an efficient function to determine if the integer a is a power of 2.

   Answer *Note that the test* $a > 0$ *also must be performed otherwise both zero and* `INT_MIN` *would falsely be regarded as powers of 2.* `INT_MIN` *is represented as* 1000...000*.*

```c
int is_power_of_2(int a)
{
        return a > 0 && (a & (a - 1)) == 0;
}
```