Login xdusek27, **Dušek Vladimír**, 2. ročník BIT, prezenční, FIT
Ak. rok 2017/2018 studium č. 1: řádný zápis

Zpět na termíny

[[IPS Home]]

Navigace

- Hlavní stránka
- Poslední změny
- Nápověda

Stránka

Zobrazit Hledat

Nástroje

- Odkazuje sem
- Nahrát soubor
- Seznam souborů
- Seznam stránek
- Historie stránky

Stránka Zdroj Sledovat

IPS:Ukol1

Obsah

Zadání

Cílem úkolu je implementovat funkce pro blokovou alokaci paměti určenou pro vlákna.

1. Alokace má využívat bezzámkové struktury.
2. Pro hledání volných bloků použijte algoritmus First fit.
3. Každé vlákno má svoji alokovanou hromadu (určenou bázovým ukazatelem první zóny) a svůj seznam (pole) bloků.
4. Společným prvkem pro všechna vlákna je globální proměnná blks_table (tu je potřeba ve zdrojovém souboru deklarovat).

Odevzdání

Odevzdávat bude každý student jeden soubor se jménem *tmal.c* do termínu Úloha 2. Pracovat můžete ve dvoučlenných týmech. Na prvním řádku v odevzdaném souboru bude v komentáři

```
// xlogin99
```

napsán login vašeho kolegy/kolegyně.

Deklarace datových typů a funkcí

Hlavičkový soubor *tmal.h*:

```
/**
 * Hlavickovy soubor pro Thread Memory Allocator.
 * Demonstracni priklad pro 1. ukol IPS/2017
 * Ales Smrcka
 */
#ifndef _TMAL_H
#define _TMAL_H

#include <stddef.h> // size_t
#include <stdbool.h> // bool

/**
 * The structure blk_t encapsulates data of a single memory block.
 */
struct blk_t {

    /// base pointer of the allocated space
    void *ptr;

    /// size of the block
    size_t size;

    /**
     * Index to blk_t in the current pool which points to the left (resp.
     * right) of the block this block points to (double-linked list). Negative
     * value means there is no such block.
     */
    int prev_idx;
    int next_idx;

    /// true = block is allocated, false = block is free
    bool used;
};

/**
 * Extended block pool: base pointer of array + array capacity.
```

```
 */
struct blk_pool_t {

    /// pointer to the first block info. NULL = block pool is not used
    struct blk_t *blks;

    /// number of active blocks (allocated for array of blk_t)
    unsigned nblks;

    /// heap capacity
    size_t heap_size;
};

/**
 * Global base pointer to block tables. Thread index is the index to blk_table.
 */
extern struct blk_pool_t *blks_table;

// shorthand for reaching a given block metadata
#define BLK(tid,i) (blks_table[tid].blks[i])

/**
 * Allocate sparse table of blocks for several threads.
 * @param  nthreads     number of threads/items in the table
 * @return              pointer to the first block pool, NULL = failed
 */
struct blk_pool_t *tal_alloc_blks_table(unsigned nthreads);

/**
 * Block metadata constructor (alone, not used block).
 * @param blk pointer to block metadata.
 */
void blk_ctor(struct blk_t *blk);

/**
 * Allocates and initialize pool of blocks.
 * @param  tid      thread index.
 * @param  nblks    capacity in number of blocks in the pool.
 * @param  theap    heap capacity for a given thread.
 * @return          pointer to the first block in a pool.
 */
struct blk_t *tal_init_blks(unsigned tid, unsigned nblks, size_t theap);

/**
 * Splits one block into two.
 * @param tid       thread index
 * @param blk_idx   index of the block to be split
 * @param req_size  requested size of the block
 * @return          index of a new block created as remainder.
 */
int tal_blk_split(unsigned tid, int blk_idx, size_t req_size);

/**
 * Merge two blocks.
 * @param tid       thread index
 * @param left_idx  index of the left block
 * @param right_idx index of the right block
 */
void tal_blk_merge(unsigned tid, int left_idx, int right_idx);

/**
 * Allocate memory for a given thread. Note that allocated memory will be
 * aligned to sizeof(size_t) bytes.
 * @param  tid  thread index (in the blocks table)
 * @param  size requested allocated size
 * @return      pointer to allocated space, NULL = failed
 */
void *tal_alloc(unsigned tid, size_t size);

/**
 * Realloc memory for a given thread.
 * @param tid   thread index
 * @param ptr   pointer to allocated memory, NULL = allocate a new memory.
 * @param size  a new requested size (may be smaller than already allocated),
 *              0 = equivalent to free the allocated memory.
 * @return      pointer to reallocated space, NULL = failed.
 */
void *tal_realloc(unsigned tid, void *ptr, size_t size);

/**
 * Free memory for a given thread.
 * @param tid   thread index
 * @param ptr   pointer to memory allocated by tal_alloc or tal_realloc.
 *              NULL = do nothing.
 */
void tal_free(unsigned tid, void *ptr);
```

```
#endif
```

Příklad použití, testovací program

Příklad použití (soubor test_tmal.c) bude překládán takto:

```
$ gcc -std=c99 -Wall -Wextra tmal.c -c
$ gcc -std=c99 -Wall -Wextra test_tmal.c -c
$ gcc -o test_tmal test_tmal.o tmal.o
```

```
#include <stdio.h>
#include <assert.h>
#include "tmal.h"
#include <unistd.h>

const unsigned int MAX_BLOCKS = 10000;
const size_t THREAD_HEAP = 200L*1024*1024; // 200MB

void debug_blkinfo(unsigned tid, unsigned i)
{
    struct blk_t *blk = &blks_table[tid].blks[i];
    printf("blks_table[%u].blks[%u] (@%p) = {\n", tid, i, blk);
    printf("  .ptr = %p,\n", blk->ptr);
    printf("  .size = %lu,\n", blk->size);
    printf("  .prev_idx = %i,\n", blk->prev_idx);
    printf("  .next_idx = %i,\n", blk->next_idx);
    printf("  .used = %s}\n", blk->used ? "true" : "false");
}

void debug6(const char *msg)
{
#ifndef NDEBUG
    printf("\n\n-- %s -----------\n", msg);
    for (int i = 0; i < 6; i++)
        debug_blkinfo(2, i);
#endif
}

int main()
{
    assert(blks_table == NULL);

    // priprava pro 4 vlakna
    tal_alloc_blks_table(4);

    assert(blks_table != NULL);

    // kazdemu vlaknu priradit/alokovat jeho heap
    for (int tid = 0; tid < 4; tid++)
        tal_init_blks(tid, MAX_BLOCKS, THREAD_HEAP);

    /**
     *   +----+----+----+----+----+-------------+
     *   | ................................... |
     *   +----+----+----+----+----+-------------+
     */
    debug6("After init");

    for (int tid = 0; tid < 4; tid++)
    {
        assert(blks_table[tid].nblks == MAX_BLOCKS);
        assert(blks_table[tid].heap_size == THREAD_HEAP);
        struct blk_t *first = blks_table[tid].blks;
        assert(first != NULL);
        assert(first->size == THREAD_HEAP);
        assert(!first->used);
        assert(first->prev_idx < 0 && first->next_idx < 0);
    }

    // vlakno 2 by rado alokovalo par bajtu
    void *b1 = tal_alloc(2, sizeof(size_t));

    /**
     *   +----+----+----+----+----+-------------+
     *   | b1 | ............................... |
     *   +----+----+----+----+----+-------------+
     */
    debug6("After the first alloc");

    assert(blks_table[2].blks[0].size == sizeof(size_t));
```

```
assert(blks_table[2].blks[0].ptr == b1);
assert(blks_table[2].blks[0].prev_idx < 0);
assert(blks_table[2].blks[0].next_idx == 1);
assert(blks_table[2].blks[0].used);
assert(blks_table[2].blks[1].size == THREAD_HEAP - sizeof(size_t));
assert(blks_table[2].blks[1].prev_idx == 0);
assert(blks_table[2].blks[1].next_idx < 0);
assert(!blks_table[2].blks[1].used);

// alokujeme vice polozek
void *  a[4];
for (int i = 0; i < 4; i++)
    a[i] = tal_alloc(2, sizeof(size_t));

/**
 *   +----+----+----+----+----+-------------+
 *   | b1 | a0 | a1 | a2 | a3 | ........... |
 *   +----+----+----+----+----+-------------+
 */
debug6("After the next 4 allocs");

tal_free(2, a[0]);

/**
 *   +----+----+----+----+----+-------------+
 *   | b1 | .. | a1 | a2 | a3 | ........... |
 *   +----+----+----+----+----+-------------+
 */
debug6("After a[0] free");

tal_free(2, a[2]);

/**
 *   +----+----+----+----+----+-------------+
 *   | b1 | .. | a1 | .. | a3 | ........... |
 *   +----+----+----+----+----+-------------+
 */
debug6("After a[2] free");

void *c1 = tal_alloc(2, 1); // alokuj pouze 1 bajt, ale zarovnej

/**
 *   +----+----+----+----+----+-------------+
 *   | b1 | c1 | a1 | .. | a3 | ........... |
 *   +----+----+----+----+----+-------------+
 */
debug6("After alloc in the middle");

assert(blks_table[2].blks[0].next_idx == 1);
assert(blks_table[2].blks[1].prev_idx == 0);
assert(blks_table[2].blks[1].ptr == c1);
assert(blks_table[2].blks[1].size == sizeof(size_t));

tal_free(2, a[1]);

/**
 *   +----+----+----+----+----+-------------+
 *   | b1 | c1 | ...... | a3 | ........... |
 *   +----+----+----+----+----+-------------+
 */
debug6("After free in the middle");

unsigned b1_idx = 0;
unsigned c1_idx = blks_table[2].blks[b1_idx].next_idx;
unsigned blank_idx = blks_table[2].blks[c1_idx].next_idx;
unsigned a3_idx = blks_table[2].blks[blank_idx].next_idx;
unsigned rest_idx = blks_table[2].blks[a3_idx].next_idx;
assert(blks_table[2].blks[b1_idx].ptr == b1);
assert(blks_table[2].blks[c1_idx].ptr == c1);
assert(blks_table[2].blks[blank_idx].ptr == a[1]);
assert(blks_table[2].blks[a3_idx].ptr == a[3]);
assert(blks_table[2].blks[rest_idx].size ==
    THREAD_HEAP - 5*sizeof(size_t));

tal_free(2, b1);

/**
 *   +----+----+----+----+----+-------------+
 *   | .. | c1 | ...... | a3 | ........... |
 *   +----+----+----+----+----+-------------+
 */
debug6("After free in the beginning");

tal_free(2, c1);

/**
```

```
     *   +----+----+----+----+----+-------------+
     *   | .................. | a3 | ........... |
     *   +----+----+----+----+----+-------------+
     */
    debug6("After free in between");

    unsigned first_idx = 0;
    unsigned second_idx = blks_table[2].blks[first_idx].next_idx;
    unsigned third_idx = blks_table[2].blks[second_idx].next_idx;
    assert(! blks_table[2].blks[first_idx].used);
    assert(blks_table[2].blks[second_idx].used);
    assert(! blks_table[2].blks[third_idx].used);

    assert(blks_table[2].blks[first_idx].size == 4*sizeof(size_t));
    assert(blks_table[2].blks[second_idx].size == sizeof(size_t));

    return 0;
}
```

Id stránky: 377, verze: 8438, dne: 2017-11-13 15:01:39 uložil: smrcka Nahoru