C for laboratory works on the discipline of System Software (Operating Systems)

Dynamic memory

To work with dynamic memory there are functions `malloc()`, `free()`, `realloc()`, their prototypes are defined in `<stdlib.h>`.

```
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

The API is the same as defined in the first laboratory work.

Windows kernel API for memory

There are two functions `VirtualAlloc()` and `VirtualFree()` for working with anonymous memory, their prototypes are defined in `<memoryapi.h>`.

```
LPVOID VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType,
    DWORD flProtect);
BOOL VirtualFree(LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType);
```

Due to the properties of API (rounding down and up argument values), it is recommended to specify `lpAddress` at the beginning of the page and `dwSize` as the number of bytes in the total number of pages.

Use the `VirtualAlloc()` function to get anonymous memory of `size` bytes.

```
ptr = VirtualAlloc(NULL, size, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
```

If the result is `NULL`, then an error occurred.

Use the `VirtualFree()` function to free memory that was got with the `VirtualAlloc()` function.

```
flag = VirtualFree(ptr, 0, MEM_RELEASE);
```

If the result is `0`, then an error has occurred.

Use the `VirtualAlloc()` function to inform the kernel that pages starting from `ptr` in total of `size` bytes do not contain any information.

```
ptr2 = VirtualAlloc(ptr, size, MEM_RESET, PAGE_READWRITE);
```

If the result is `NULL`, then an error occurred.

For all calls in case of an error, the error code can be obtained with the `GetLastError()` function.

Unix kernel API for memory

There are functions `mmap()`, `munmap()` and `madvise()` for working with memory, their prototypes are defined in `<sys/mman.h>`.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
int madvise(void *addr, size_t length, int advice);
```

Due to properties of API (rounding of argument values), it is recommended to specify `addr` at the beginning of the page and `length` as the number of bytes in the total number of pages.

Use the `mmap()` function to obtain anonymous memory of `length` bytes.

```
ptr = mmap(NULL, length, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
```

It can be `MAP_ANON` instead of `MAP_ANONYMOUS`. If the result is `MAP_FAILED`, then an error occurred.

Use the `munmap()` function to free the memory got from the `mmap()` function.

```
rv = munmap(ptr, length);
```

If the result is `-1`, then an error occurred.

Use the `madvise()` function to inform the kernel that pages starting from `ptr` of the total `size` bytes do not contain any information.

```
rv = madvise(ptr, length, MADV_DONTNEED);
```

The `MADV_FREE` value can be used instead of `MADV_DONTNEED`. If the result is `-1`, then an error occurred.

For all calls in case of an error, the error code can be obtained from `errno`.

Choosing a compiler

Choose a compiler that supports the C11 standard or later and does not have too many standard extensions.

If you use a Unix-like system, then the C compiler should be there or it can be installed from the distributive. Choose **gcc** or **clang**.

If you use Windows, then choose **Code::Blocks** + **MinGW** (available as a single installation file), this is the IDE, **gcc** and required programs.

Specify the language standard **-std=c11** or **-std=c17** (for **Code::Blocks** this can be set in Settings / Compiler / Compiler Flags "Have gcc follow ...").

Enable warning messages with **-Wall**, different levels of optimization with **-O0** and **-O2** options (in **Code::Blocks** this can be set by Build / Select Target / Debug and Release). Also add the **-Wpedantic** and **-Wconversion** options (in **Code::Blocks** these options can be set in Settings / Compiler / Other Compiler Options).

If you use **gcc** from the **Code::Blocks** installation then in Settings / Compiler / #defines add `__USE_MINGW_ANSI_STDIO=1` so that `printf()` will conform to the modern standard.

Step-by-step implementation of the first work

1. Create a project of the following files: **main.c**, **block.c**, **block.h**, **allocator.c**, **allocator.h**, **allocator_impl.h**, **kernel.c**, **kernel.h**, **config.h**. Define macros for the size of the page, the size of the default arena in **config.h**. Define a macro for determinig alignment in **allocator_impl.h**. Create a function to get memory from the existing memory allocator (the `malloc()` function) in **kernel.c**, its prototype write in **kernel.h**. Create the structure of the block header in **block.h**: the size of the current block, the size of the previous block, the flags “block is busy”, “first block in the arena”, “last block in the arena”.

   Implement the following functions in **block.c**: `block_split()` (split a block if possible, this function is necessary for `mem_alloc()`), `block_merge()` (unite a block and its right neighbor, if both are free, this function is necessary for `mem_free()`). Create inline functions `block_to_payload()`, `payload_to_block()`, `block_next()`, `block_prev()`, setters and getters for all fields of the block header structure (all functions are single-line, now it does not make much sense, but will be required in the second step) in **block.h**.

1. (continued) Implement the `mem_alloc()` function which can work with one arena in **allocator.c**, its prototype write in **allocator.h**. Use linear scanning of blocks in the arena, first fit strategy in `mem_alloc()`. Call `mem_alloc()` manually and for each successful call fill the returned block with random numbers in **main.c**. Implement the `mem_show()` function to output the contents of the headers of all blocks from the arena in **allocator.c**. Check that `mem_alloc()` has no overflow when it aligns given size (call `mem_alloc(SIZE_MAX)`, `mem_alloc(SIZE_MAX - 1)`, etc.).

   Next implement the `mem_free()` function in **allocator.c**. Check all cases in `mem_alloc()` and `mem_free()` functions: the first block, the last block, the block with neighbors, four combinations with neighbors (busy curr busy, free curr busy, busy curr free, free curr free). Call `mem_show()` for each test.

2. Encode all flags in the block header in two size fields. Because alignment is a power of two, these flags are encoded in lower bits of sizes. Define the flags in `#define`s in **block.h**, change all the setters and getters for the block header in **block.h**. Check the correctness of `mem_alloc()` and `mem_free()` functions.

3. Change the call to the `malloc()` function in **kernel.c** with the call to the kernel memory allocation function. Create a function to free memory (now it is not used, the function must have an argument with the size of memory, this is required for Unix). Check the correctness of `mem_alloc()` and `mem_free()` functions.

4. Create the `mem_realloc()` function in **allocator.c**, which can decrease the block in-place and can increase the block in-place to the right, if the right neighboring block is free and has sufficient size (do not forget about the size of the header of the right neighboring block). Otherwise, the function calls `mem_alloc()` → `memcpy()` → `mem_free()`. To implement this function use `block_split()` and `block_merge()` functions. Check all cases in the logic of `mem_realloc()`.

5. Develop code or take existent code for RB tree or AVL tree. Check whether this code works in a separate program. Simplify the code for the tree: functions should get pointers to the tree node structures and should return pointers to these structures, encode the key as `size_t`, check that the code for the tree does not get and does not free dynamic memory. Check whether the modified code works in a separate program.

5. (continued) Add two pointers to the node structure to create doubly linked lists of nodes with the same key values. Modify the functions of adding nodes to and removing nodes from the tree so that they support the same key values. If it is necessary to add a node to or remove a node from the tree by key, then add or remove a node from the list. If a node is removed and it is the node of the tree, then the next node from its list should be emplaced in its place. Create a function to walk through all nodes of the tree and call the function to output the contents of the node (do not forget to walk through the list in each node).

   This optimization with the tree cannot be done and the tree code without modifications can be used. In this case, the program will run slower.

6. Create the **tree.h** file in the allocator project, write synonyms for tree structures and tree nodes in `typedef`s and `#define`s for calling tree functions there (for example, `typedef struct avl_tree block_tree;` and `#define block_tree_add (t, n) avl_tree_add((t), (n))`). This will allow to change the tree code if needed. The tree node structure will be located in the free block payload. Create inline functions `block_to_node()` and `node_to_block()` in **block.h**

6. (continued) (it makes sense to call `block_to_payload()` and `payload_to_block()` in these functions).

Then there are two variants: do not create blocks that are smaller than the size of the tree node structure, create free blocks that are smaller than the size of the tree node structure, but enough to place structures in their payloads to create double linked segregated lists.

Add tree code files to the allocator project. Check that everything is compiled. Change the first fit algorithm with the best fit algorithm that searches in a tree in `mem_alloc()`. Change the corresponding code in `mem_free()`. Change the corresponding code in `mem_realloc()`. Take the code that outputs the contents of the block header from `mem_show()` and create a separate function from this code. Call the function to walk through the tree and pass it the pointer to the function that outputs the contents of the block header. Check correctness of `mem_alloc()`, `mem_free()` and `mem_realloc()` manually in **main.c**.

7. Create an automatic tester in **main.c**, as specified in the task for the laboratory work or better. Store 100 results of the allocator API fuction calls and perform 10 000 calls to the allocator API functions in the tester (the number of values is conditional). Test the program.

8. Add the ability to create additional default arenas in **allocator.c**. Add the ability to free the arena in `mem_free()` (if there is one free block left in the arena, i.e. it is the first and last one, then the arena is free). Test the correctness of the allocator manually and by the automatic tester.

9. Add the ability to create arenas with one block larger than the maximum block size in the default arena in **allocator.c**. Modify `mem_realloc()`. It is necessary to add the following variants: large block → large block of another size, large block → block in the default arena. It is possible not to decrease the block until its new size is not less than N% of its actual size (for example, there is a block of 1 GB and then it is decrease by 4 KB). Some kernels allow to free any memory, this allows to shrink the size of a large block without reallocaction. Test the correctness of the allocator manually and by the automatic tester.

10. Add a function to inform the kernel that some pages do not contain any information in **kernel.c**. In order to use this function, it is necessary to specify the beginning of the page. Add the `size_t` offset field to the block header. Store the offset of the block payload from the beginning of its arena in this field. This field is not required for non-paged memory.

    For paged memory, the beginning of the arena is the beginning of the page. Using this field and page size, it is possible to determine where a page starts in a block and the number of pages in a block. Modify `mem_alloc()`, `mem_free()` and `mem_realloc()` functions and call this function if some memory is freed and there is the beginning of the page in it and an integer number of pages after this beginning.

    Checking the implementation this algorithm is difficult, because the kernel may not immediately “free” the specified pages, so before invoking the kernel function in **kernel.c**, fill the pages to be freed by random data, and then “free” it. Test the correctness of the allocator manually and by the automatic tester.