System Software (Operating Systems)
Laboratory work №1
General-purpose memory allocator using tags

A memory allocator is a part of a program that is used for memory allocation according to the requests from other parts of a program. Memory allocators can be different. A general-purpose memory allocator can be used by any program with almost any memory allocator requirements. Memory allocators can call each other. For example, a user program memory allocator can call the kernel memory allocator.

A general-purpose memory allocator must perform at least three tasks: allocating a memory block of the specified size, freeing a previously allocated memory block, and resizing a previously allocated memory block.

The general-purpose memory allocator API is defined by the following functions (semantics was taken from the C programming language standard, such an API is common in both user programs and the kernel):

- `void *mem_alloc(size_t size);`

  The `mem_alloc()` function allocates space for an object whose size is specified by `size` bytes and whose contents is undefined. The `mem_alloc()` function returns either a null pointer or a pointer to the allocated space.

- `void mem_free(void *ptr);`

  The `mem_free()` function causes the space pointed to by the `ptr` pointer to be freed, i.e. it becomes available for further allocations. If `ptr` is a null pointer, then no action is taken. Otherwise, if the argument does not correspond to a pointer that was previously returned by the allocator API function, or if the corresponding space was freed by calling the allocator API functions, then the behavior is undefined. The `mem_free()` function does not return a value.

- `void *mem_realloc(void *ptr, size_t size);`

  The `mem_realloc()` function reallocate the memory of the object pointed to by the `ptr` pointer and returns a pointer to a new object whose size is specified by `size` bytes. The contents of the new object must be the same as the contents of the object before reallocation, to the smallest of the new and previous sizes. Any bytes in the new object that have offsets greater than the size of the original object have undefined values. If `ptr` is a null pointer, then the `mem_realloc()` function behaves like the `mem_alloc()` function for the specified size. Otherwise, if `ptr` does not match the pointer previously returned by the allocator API function, or if the corresponding space was freed by calling the allocator API functions, then the behavior is undefined. If memory for a new object cannot be allocated, then the original object is not freed and its contents remains the same. The `mem_realloc()` function returns a pointer to a new object (which can have the same value as a pointer to the original object), or a null pointer if the new object could not be allocated.

- The pointer returned in the case of a successful allocation is properly aligned so that it can be assigned to the pointer to an object of any type with the alignment requirement, and then can be

used to access such an object or array of such objects in the allocated space (until this space is not explicitly freed). The allocated object can be used from its allocation to its release. Each allocation must point to an object that is not related to any other object. The returned pointer points to the beginning (lowest byte address) of the allocated space. If the size of the requested space is zero, then the behavior is determined by the implementation: either a null pointer is returned, or the behavior is as if the size is a non-zero value, except that the returned pointer must not be used to access the object. A null pointer is guaranteed to be unequal to any object or function.

Implementing an effective (does not matter what it means) general-purpose memory allocator is not an easy task. The difficulty is that the sequence of function calls for allocating, freeing, and resizing memory blocks is unknown. Therefore, it is impossible to talk about the optimal, best or ideal solution.

If a user program is running in the system managed by the kernel, then the general-purpose memory allocator of a user program requests memory for allocation from the kernel. The memory allocator requests the kernels to add some memory to the process address space, or returns part of the process address space to the kernel. When the memory allocator of the user program requests memory from the kernel, then received memory is called an arena. The memory in the computer can be without abstraction, with segmented organization or consist of pages. If the memory consists of pages, then all the RAM memory is an array of pages of the same size. If the memory consists of pages, then one or several pages can be requested from the kernel, otherwise 1 byte or more can be requested from the kernel.

Is it possible to implement a memory allocator in a user program using only a kernel memory allocator? It is not possible, because if the memory consists of pages, then giving one or several pages for each request is not effective. It is also not efficient to transfer memory allocation to the kernel, because it is better for the system if the system runs in kernel mode for the least amount of time and if the system performs less number of context switching.

There are several ways to implement general-purpose memory allocators. In this laboratory work, it is necessary to use a method that keeps accounting information about each block in its header (tag).

The simplest way to implement a memory allocator is as follows. The block header has the block size and block status (busy or free), and the block header is located at the beginning of the block. Also, each arena has its own header, which determines the size of the arena and a pointer to the next arena. Such information is enough. But such accounting of blocks and arenas is not effective, because it is necessary to go through all the arenas and all the blocks in each arena to find free space, to release the block or to change the size of the block.

There are two types of arenas in the memory allocator. The first type is the default arena, an arena that is divided into blocks. The size of the default arena may vary, but the memory allocator has information about its size. The second type is an arena that is larger than the default arena, and is requested from the kernel if the size of the default arena is not large enough to fulfill the program request and this arena is not divided into blocks, it contains a single block.

Conditions and requirements for a memory allocator

- The memory allocator uses headers for memory blocks to implement their accounting.
- A user program runs in the system that is managed by the kernel.

- The kernel provides system calls that allow to add memory to the address space of the user process, return part of the address space of the user process to the kernel, tell the kernel that some pages of the address space of the user process do not contain any information.
- The memory in the computer can consist of pages or not (it is necessary to support both variants).
- The memory allocator is given the page size. If the memory in the computer consists of pages, then this value is greater than 0 and is a power of two.
- The memory allocator gets memory for allocation (arenas) and accounting from the kernel (it is not allowed to use another memory allocator in a user program and use the memory for allocation and accounting in the process stack).
- The memory allocator is given the default arena size.
- The number of arenas and blocks in the program potentially is not limited.
- The memory allocator must be hardware-independent (do not depend on the processor architecture).
- The memory allocator must be system-independent (do not dependent on the OS, except for functions that work with the process address space).
- The memory allocator should work with both large address spaces and small address spaces (few KB).
- The maximum block size is determined by the system where the memory allocator is running.
- If some information can be encoded in other information, then use such encoding.
- The `mem_realloc()` function must be able to change the size of the block in-place (without moving the contents of the block).
- The pointer that is returned by the kernel when a user program requests memory is always aligned properly so that it can be assigned to the pointer to the object of any type that requires alignment.
- If the arena becomes free, then its memory must be returned to the kernel.
- If some page of memory managed by the memory allocator does not contain any information, then the allocator must notify the kernel about this by the corresponding system call.
- The complexity of the algorithms in the memory allocator must be O(log n), optimize the algorithms even if they have the complexity O(log n).
- Optimize the amount of memory used for accounting in the memory allocator.
- Optimize memory footprint.
- If it is necessary to use some tree in the memory allocator, then it is allowed to use existent solution.
- The compiler and user program do not know about the implementation of the memory allocator.

The task

- Implement a general-purpose memory allocator for a user program that implements the above-defined API respecting the above-mentioned conditions and requirements for a memory allocator.
- To implement the memory allocator, choose the system programming language yourself.
- Implement the `mem_show()` function, which should output memory allocator data structures for debugging.
- Check the correctness of the memory allocator implementation with the help of an automatic tester.
- If possible, then use a developed memory allocator with a real program that uses a memory allocator. If that program is multithreaded, then add a mutex to each memory allocator API function.

Report

- Description of the developed memory allocator (optional).
- Listing of the developed program.
- Example of usage of the developed memory allocator (several calls to the memory allocator API functions and the corresponding outputs of the `mem_show()` function).

Automatic tester

Automated testing should detect errors in the implementation that were not detected during manual testing. The idea of testing is to call the functions `mem_alloc()`, `mem_free()` and `mem_realloc()` in a random order and check whether their invocations do not destroy the contents of the allocator data structures and block data.

To organize this, it is necessary to create an array that will store the results of successful calls to the `mem_alloc()` and `mem_realloc()` functions. These results are block pointers, their size, and block data checksums. All got blocks must be filled with random data and their checksums must be calculated. Before calling the `mem_realloc()` and `mem_free()` functions, the checksum of the block contents must be checked. The checksum allows to detect the modification of the block contents. After testing, all checksums must be checked and all blocks must be freed.

Literature

- Bill Blunden. Memory Management: Algorithms and Implementation in C/C++.
- Robert Sedgewick. Algorithms in …
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms.

Implementation variant

Consider implementation variant of a memory allocator that meets the above defined API, conditions and requirements.

If the memory of the computer consists of pages, then any arena consists of one or several pages. Each block in any arena has the following header: the block size, the previous block size, three flags (block is busy, a block is the first in the arena, a block is the last in the arena). Because the pointers need to be aligned, these three flags in the block header are encoded in other header fields.

To implement the algorithms in the memory allocator with complexity O(log n), a self-balancing tree can be used. For example, the RB tree or AVL tree can be used. All free blocks from default arenas are combined into one tree, the key is the block size. The header for the tree's node is created in the free block. If there is a request for a free block from the default arena, then it is looked for in the tree using the best fit strategy. If there are free blocks of the same size in the tree, then they are combined into a list, this can be implemented by modifying the code of the tree implementation (elements with the same keys are added to the list headed by the node of the tree). It is possible to not modify the tree code, but then work with the tree will be slower.

To implement a memory allocator system programming language C, C++ or Rust can be used.

If the memory allocator is run on the OS that is POSIX-compliant, then the `mmap()` system call with the `MAP_ANONYMOUS` or `MAP_ANON` argument is used to get anonymous memory from the kernel (this argument is not specified in POSIX, but it exists on most systems). The `munmap()` system is used to return memory to the kernel. To inform the kernel that some pages from the process memory do not contain any information, the `madvise()` system call with the `MADV_FREE` or `MADV_DONTNEED` argument is used (this function is not available in POSIX, but it exists in most systems). If the memory allocator is run on Windows, then the same steps can be achieved by invoking the `VirtualAlloc()` function with the arguments `MEM_RESERVE|MEM_COMMIT` and `MEM_RESET` and the `VirtualFree()` function with the argument `MEM_RELEASE`.