Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №4 по курсу

«Операционные системы»

Группа: М8О-210Б-23

Студент: Коваленко Д.А

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 09.01.25

Москва, 2025

Постановка задачи

Цель работы:

Приобретение практических навыков в:

1. Создании аллокаторов памяти и их анализу;
2. Создании динамических библиотек и программ, использующие динамические библиотеки.

Задание:

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

- Фактор использования
- Скорость выделения блоков
- Скорость освобождения блоков
- Простота использования аллокатора

Требуется создать две динамические библиотеки, реализующие два аллокатора, соответственно. Библиотеки загружаются в память с помощью интерфейса ОС (dlopen / LoadLibrary) для работы с динамическими библиотеками. Выбор библиотеки, реализующей аллокатор, осуществляется чтением первого аргумента при запуске программы (argv[1]). Этот аргумент должен содержать путь до динамической библиотеки (относительный или абсолютный).

Если аргумент не передан или по переданному пути библиотеки не оказалось, то указатели на функции, реализующие API аллокатора ниже, должны быть присвоены функциям, которые оборачивают системный аллокатор ОС (mmap / VirtualAlloc) в этот API. Эти аварийные оберточные функции должны быть реализованы внутри программы, которая загружает динамические библиотеки.

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям malloc и free (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра (mmap / VirtualAlloc). Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.

Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

- Allocator* allocator_create(void *const memory, const size_t size) (инициализация аллокатора на памяти memory размера size);

- void allocator_destroy(Allocator *const allocator) (деинициализация структуры аллокатора);
- void* allocator_alloc(Allocator *const allocator, const size_t size) (выделение памяти аллокатором памяти размера size);
- void allocator_free(Allocator *const allocator, void *const memory) (возвращает выделенную память аллокатору);

Вариант 4. Алгоритм Мак-Кьюзика-Кэрелса и блоки по 2^n.

Общий метод и алгоритм решения

Алгоритм аллокатора памяти на основе блоков 2^n

Краткое описание:
Этот алгоритм делит память на блоки размеров, кратных степени двойки. Если требуется меньший блок, память рекурсивно делится пополам, пока не будет найден подходящий размер.

Принцип работы:

- **Инициализация**:
  Создаётся список свободных блоков для каждого размера.
  Например, блоки размером 16 байт, 32 байта, 64 байта и т.д.
- **Выделение памяти**:
  Запрашивается блок минимального подходящего размера. Если подходящего блока нет, берётся больший блок и делится пополам. Один из "братьев" остаётся свободным.
- **Освобождение памяти**:
  При освобождении памяти система пытается объединить блок с его "братом" если он тоже свободен. Это уменьшает фрагментацию.

Структура и операции:

- **Инициализация**:
  Списки свободных блоков (`free_lists`) инициализируются на основе доступной памяти.
- **Выделение** (`allocator_alloc`):
  1. Находится подходящий список через `log2` размера.
  2. Если блока нет, больший блок делится до нужного размера.
- **Освобождение** (`allocator_free`):
  Проверяется, можно ли объединить блоки в один более крупный.

Преимущества:

1. **Быстрота операций**:
   Все действия (выделение, освобождение) работают с помощью простых вычислений индексов.
2. **Меньше фрагментации**:
   Благодаря объединению "братьев".

Недостатки:

1. **Внутренняя фрагментация**:
   Память выделяется кратной степени двойки, что может приводить к перерасходу.
2. **Сложность управления большими блоками**:
   При исчерпании крупной памяти восстановление может быть трудным.

Алгоритм

Краткое описание:
Алгоритм McKusick-Karels

McKusick-Karels организует память в виде блоков, оптимизируя использование памяти для фиксированных или предсказуемых размеров объектов.

Принципы работы и структура:

Инициализировать аллокатор и подготовить список свободных блоков. Список свободных блоков представляет собой структуру данных (обычно односвязный или двусвязный список), где каждый элемент описывает свободный участок памяти. Первый блок инициализируется как единственный элемент списка.

Структура блока содержит:

◦ Размер блока: количество байтов, доступных в этом свободном блоке.

◦ Указатель на следующий свободный блок: используется для связывания элементов списка.

Структура аллокатора содержит:

◦ Указатель на начало память;

◦ Указатель на голову списка свободных элементов;

◦ Размер памяти общий.

Преимущества:

• Простота реализации: основные алгоритмы достаточно просты для понимания и реализации.

• Гибкость: подходит для запросов любого размера, так как блоки не фиксированного размера.

• Минимизация внутренней фрагментации: блоки выделяются под запрошенный размер (с учетом выравнивания)

• Легкость отладки: простая структура данных позволяет отслеживать состояние памяти и находить ошибки.

Недостатки:

• Фрагментация: при частом выделении и освобождении блоков возникает фрагментация – множество мелких свободных блоков, непригодных для выделения под большие запросы.

• Зависимость от стратегии: эффективность сильно зависит от выбранной стратегии поиска свободного блока и сценария использования.

• Нет объединения свободных блоков: освобожденные блоки не объединяются с соседними свободными блоками, что усугубляет фрагментацию.

Результаты тестирования

1. **Сравнение производительности**
   Аллокатор McKusick-Karels продемонстрировал более высокую скорость работы по сравнению с аллокатором, использующим блоки, кратные степеням двойки (2^n). Среднее время выполнения операций для McKusick-Karels составило 1**.3 мс ± 0.6 мс**, тогда как аллокатор 2^n показал результат 1**.1 мс ± 0.5 мс**. Это означает, что алгоритмы работают с примерно одинаковой скоростью.

- **Аллокатор 2^n** успешно справился с основными тестами, включая выделение и освобождение памяти, а также обработку нескольких блоков одновременно.
- **McKusick-Карелс**, напротив, корректно прошёл первые три теста, включая выделение и освобождение памяти. Но в Тестах 4 и 5 возникли сбои, что говорит о проблемах при работе с несколькими блоками или повторным использованием освобождённой памяти.

3. **Анализ памяти (Valgrind)**
   Обе реализации продемонстрировали отсутствие утечек памяти: все

выделенные блоки успешно освобождались. Это подтверждает корректность работы механизмов освобождения памяти в обоих аллокаторах.

4. **Общий вывод**

- **McKusick-Карелс** выигрывает в скорости и лучше справляется с базовыми тестами, однако его эффективность снижается при одновременной работе с несколькими блоками. Такой аллокатор будет эффективен в задачах, где приоритетны быстродействие и простота.
- **Аллокатор 2^n** уступает в производительности, но более устойчив к сложным сценариям, таким как одновременная обработка нескольких запросов. Он подходит для задач, требующих устойчивости к фрагментации и работы с крупными объёмами данных.

1. **Системные вызовы:**
    - void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset); – выделяет блок памяти, резервирует память для аллокатора.
    - int munmap(void *start, size_t length); – освобождает ранее выделенную память и очищает ресурсы, выделенные через mmap.
    - void *dlopen(const char *filename, int flag); – загружает динамическую библиотеку, имя которой указано в строке filename, и возвращает прямой указатель на начало динамической библиотеки.
    - void *dlsym(void *handle, char *symbol); – использует указатель на динамическую библиотеку, возвращаемую dlopen, нужен для использования функций аллоеатора
    - int dlclose(void *handle); – закрывает динамическую библиотеку, освобождает ресурсы.

Код программы

main.c

```
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
```

```
typedef void *(*allocator_create_t)(void *const memory, const size_t size);
typedef void *(*allocator_alloc_t)(void *const allocator, const size_t size);
typedef void (*allocator_free_t)(void *const allocator, void *const memory);
typedef void (*allocator_destroy_t)(void *const allocator);

void log_info(const char *msg) { write(STDOUT_FILENO, msg, strlen(msg)); }

void log_error(const char *msg) { write(STDERR_FILENO, msg, strlen(msg)); }

int main(int argc, char *argv[]) {
 if (argc < 2) {
  log_error("Usage: <program> <path_to_allocator_library>\n");
  return 1;
 }

 void *library_handle = dlopen(argv[1], RTLD_LAZY);
 if (library_handle == NULL) {
  log_error("Error loading library: ");
  log_error(dlerror());
  log_error("\n");
  return 1;
 }

 allocator_create_t create_allocator =
    (allocator_create_t)dlsym(library_handle, "allocator_create");
 allocator_alloc_t allocate_memory =
    (allocator_alloc_t)dlsym(library_handle, "allocator_alloc");
 allocator_free_t free_memory =
    (allocator_free_t)dlsym(library_handle, "allocator_free");
 allocator_destroy_t destroy_allocator =
    (allocator_destroy_t)dlsym(library_handle, "allocator_destroy");

 if (!create_allocator || !allocate_memory || !free_memory ||
    !destroy_allocator) {
  log_error("Error locating functions: ");
  log_error(dlerror());
  log_error("\n");
  dlclose(library_handle);
```

```
  return 1;
}

size_t pool_size = 8 * 1024 * 1024;
void *memory_pool = mmap(NULL, pool_size, PROT_READ | PROT_WRITE,
                         MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
if (memory_pool == MAP_FAILED) {
  log_error("Memory pool creation failed\n");
  dlclose(library_handle);
  return 1;
}

void *allocator = create_allocator(memory_pool, pool_size);
if (!allocator) {
  log_error("Failed to create allocator\n");
  munmap(memory_pool, pool_size);
  dlclose(library_handle);
  return 1;
}

void *block1 = allocate_memory(allocator, 512);
if (block1) {
  log_info("Test 1: Memory allocated (512 bytes)\n");
  free_memory(allocator, block1);
  log_info("Test 1: Memory freed (512 bytes)\n");
} else {
  log_error("Test 1: Memory allocation failed\n");
}

void *block2 = allocate_memory(allocator, 16 * 1024 * 1024);
if (!block2) {
  log_info("Test 2: Allocation failed (expected for oversized request)\n");
} else {
  log_error("Test 2: Unexpected success in oversized allocation\n");
  free_memory(allocator, block2);
}

void *block3 = allocate_memory(allocator, 1024);
```

```
  if (block3) {
    log_info("Test 3: Memory allocated (1024 bytes)\n");
    free_memory(allocator, block3);
    log_info("Test 3: Memory freed (1024 bytes)\n");
  } else {
    log_error("Test 3: Memory allocation failed\n");
  }

  void *block4 = allocate_memory(allocator, 256);
  void *block5 = allocate_memory(allocator, 128);
  if (block4 && block5) {
    log_info("Test 4: Two memory blocks allocated (256 and 128 bytes)\n");
    free_memory(allocator, block4);
    free_memory(allocator, block5);
    log_info("Test 4: Memory blocks freed\n");
  } else {
    log_error("Test 4: Memory allocation failed\n");
  }

  void *block6 = allocate_memory(allocator, 2048);
  free_memory(allocator, block6);
  block6 = allocate_memory(allocator, 2048);
  if (block6) {
    log_info("Test 5: Memory reallocated (2048 bytes)\n");
    free_memory(allocator, block6);
  } else {
    log_error("Test 5: Memory allocation failed\n");
  }

  destroy_allocator(allocator);
  log_info("Allocator destroyed\n");

  munmap(memory_pool, pool_size);

  dlclose(library_handle);

  return 0;
}
```

mcKusickCarels.c

```
#include <stddef.h>

typedef struct Block {
  struct Block *next;
} Block;

typedef struct MemoryAllocator {
  void *memory_start;
  size_t memory_size;
  Block *available_blocks_head;
} MemoryAllocator;

size_t calculate_aligned_size(size_t size, size_t alignment) {
  if (alignment == 0) {
    return size;
  }
  return (size + (alignment - 1)) & ~(alignment - 1);
}

MemoryAllocator *allocator_create(void *memory_pool, size_t total_size) {
  if (memory_pool == NULL || total_size < sizeof(MemoryAllocator)) {
    return NULL;
  }

  MemoryAllocator *allocator = (MemoryAllocator *)memory_pool;
  allocator->memory_start = (char *)memory_pool + sizeof(MemoryAllocator);
  allocator->memory_size = total_size - sizeof(MemoryAllocator);
  allocator->available_blocks_head = (Block *)allocator->memory_start;

  if (allocator->available_blocks_head != NULL) {
    allocator->available_blocks_head->next = NULL;
  }
```

```
  return allocator;
}

void allocator_destroy(MemoryAllocator *allocator) {
  if (allocator == NULL) {
    return;
  }

  allocator->memory_start = NULL;
  allocator->memory_size = 0;
  allocator->available_blocks_head = NULL;
}

void *allocator_alloc(MemoryAllocator *allocator, size_t size) {
  if (allocator == NULL || size == 0) {
    return NULL;
  }

  size_t aligned_size = calculate_aligned_size(size, 8);
  Block *previous_block = NULL;
  Block *current_block = allocator->available_blocks_head;

  while (current_block != NULL) {
    if (aligned_size <= allocator->memory_size) {
      if (previous_block != NULL) {
        previous_block->next = current_block->next;
      } else {
        allocator->available_blocks_head = current_block->next;
      }
      return current_block;
    }

    previous_block = current_block;
    current_block = current_block->next;
```

```
  }

  return NULL;
}

void allocator_free(MemoryAllocator *allocator, void *memory_block) {
  if (allocator == NULL || memory_block == NULL) {
    return;
  }

  Block *block_to_free = (Block *)memory_block;
  block_to_free->next = allocator->available_blocks_head;
  allocator->available_blocks_head = block_to_free;
}
```

block2n.c

```
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>

#define MIN_BLOCK_SIZE 16
#define MAX_BLOCK_SIZE(size) ((size < 32) ? 32 : size)

int compute_log2(int value) {
  if (value == 0) {
    return -1;
  }
  int res = 0;
  while (value > 1) {
    value >>= 1;
    res++;
  }
```

```
  return res;
}

typedef struct BlockMetadata {
  struct BlockMetadata *next_block;
} BlockMetadata;

typedef struct Allocator {
  BlockMetadata **free_lists;
  size_t num_lists;
  void *base_addr;
  size_t total_size;
} Allocator;

Allocator *allocator_create(void *memory, size_t size) {
  if (memory == NULL || size < sizeof(Allocator)) {
    return NULL;
  }
  Allocator *allocator = (Allocator *)memory;
  allocator->base_addr = memory;
  allocator->total_size = size;

  size_t max_block_size = MAX_BLOCK_SIZE(size);
  size_t min_block_size = sizeof(BlockMetadata) + MIN_BLOCK_SIZE;

  allocator->num_lists = (size_t)floor(compute_log2(max_block_size) / 2) + 3;
  allocator->free_lists =
      (BlockMetadata **)((char *)memory + sizeof(Allocator));

  for (size_t i = 0; i < allocator->num_lists; i++) {
    allocator->free_lists[i] = NULL;
  }

  void *current_block = (char *)memory + sizeof(Allocator) +
                        allocator->num_lists * sizeof(BlockMetadata *);
```

```
size_t remaining_memory =
    size - sizeof(Allocator) - allocator->num_lists * sizeof(BlockMetadata *);


size_t block_size = MIN_BLOCK_SIZE;
while (remaining_memory >= min_block_size) {
  if (block_size > remaining_memory) {
    break;
  }


  if (block_size > max_block_size) {
    break;
  }


  if (remaining_memory >= (block_size + sizeof(BlockMetadata)) * 2) {
    for (int i = 0; i < 2; i++) {
      BlockMetadata *header = (BlockMetadata *)current_block;
      size_t index;
      if (size == 0) {
        index = 0;
      } else {
        index = (size_t)compute_log2(block_size);
      }


      header->next_block = allocator->free_lists[index];
      allocator->free_lists[index] = header;


      current_block = (char *)current_block + block_size;
      remaining_memory -= block_size;
    }
  } else {
    BlockMetadata *header = (BlockMetadata *)current_block;
    size_t index;
    if (size == 0) {
      index = 0;
    } else {
      index = (size_t)compute_log2(block_size);
```

```
      }

      header->next_block = allocator->free_lists[index];
      allocator->free_lists[index] = header;

      current_block = (char *)current_block + remaining_memory;
      remaining_memory = 0;
    }

    block_size <<= 1;
  }
  return allocator;
}

void *allocator_alloc(Allocator *allocator, size_t size) {
  if (allocator == NULL || size == 0) {
    return NULL;
  }

  if (size > allocator->total_size - sizeof(Allocator)) {
    return NULL;
  }
  size_t index = (size == 0) ? 0 : compute_log2(size) + 1;
  if (index >= allocator->num_lists) {
    index = allocator->num_lists;
  }
  bool found_block = false;
  if (allocator->free_lists[index] == NULL) {
    while (index <= allocator->num_lists) {
      if (allocator->free_lists[index] != NULL) {
        found_block = true;
        break;
      } else {
        ++index;
      }
    }
```

```
    if (!found_block) {
      return NULL;
    }
  }

  BlockMetadata *block = allocator->free_lists[index];
  allocator->free_lists[index] = block->next_block;

  return (void *)((char *)block + sizeof(BlockMetadata));
}

void allocator_free(Allocator *allocator, void *ptr) {
  if (!allocator || !ptr) {
    return;
  }

  BlockMetadata *block = (BlockMetadata *)((char *)ptr - sizeof(BlockMetadata));
  size_t block_size_in_bytes =
      (char *)block + sizeof(BlockMetadata) - (char *)allocator->base_addr;
  size_t current_block_size = 32;

  while (current_block_size <= block_size_in_bytes) {
    size_t next_size = current_block_size << 1;
    if (next_size > block_size_in_bytes) {
      break;
    }
    current_block_size = next_size;
  }

  size_t index =
      (block_size_in_bytes == 0) ? 0 : (size_t)compute_log2(current_block_size);
  if (index >= allocator->num_lists) {
    index = allocator->num_lists - 1;
  }
```

```
  block->next_block = allocator->free_lists[index];
  allocator->free_lists[index] = block;
}

void allocator_destroy(Allocator *allocator) {
  if (allocator) {
    munmap(allocator->base_addr, allocator->total_size);
  }
}
```

Протокол работы программы

Тестирование:
denis@denis-Vivobook-ASUSLaptop-M1402IA-M1402IA:~/projects_c++/os/lab4$ hyperfine -r 10000 './allocator_test ./libblock2n.so'

Benchmark 1: ./allocator_test ./libblock2n.so

  Time (mean ± σ):   1.3 ms ±   0.6 ms   [User: 0.4 ms, System: 1.1 ms]

  Range (min … max):   0.5 ms …   3.4 ms   10000 runs

  Warning: Command took less than 5 ms to complete. Results might be inaccurate.

denis@denis-Vivobook-ASUSLaptop-M1402IA-M1402IA:~/projects_c++/os/lab4$ hyperfine -r 10000 './allocator_test ./libmckusickCarels.so'

Benchmark 1: ./allocator_test ./libmckusickCarels.so

  Time (mean ± σ):   1.1 ms ±   0.5 ms   [User: 0.4 ms, System: 0.9 ms]

  Range (min … max):   0.3 ms …   2.8 ms   10000 runs

  Warning: Command took less than 5 ms to complete. Results might be inaccurate.

  strace ./allocator_test ./libmckusickCarels.so

  execve("./allocator_test", ["./allocator_test", "./libmckusickCarels.so"], 0x7ffcd97ed5a8 /* 55 vars */) = 0

brk(NULL)   = 0x5bc46e798000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd8fe5d730) = -1 EINVAL (Недопустимый аргумент)

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x78b099418000

access("/etc/ld.so.preload", R_OK)   = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=105007, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 105007, PROT_READ, MAP_PRIVATE, 3, 0) = 0x78b0993fe000

close(3)   = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48

pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f\221\2039x\324\224\323\236S"..., 68, 896) = 68

newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x78b099000000

mprotect(0x78b099028000, 2023424, PROT_NONE) = 0

mmap(0x78b099028000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x78b099028000

mmap(0x78b0991bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x78b0991bd000

mmap(0x78b099216000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x78b099216000

mmap(0x78b09921c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x78b09921c000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x78b0993fb000

arch_prctl(ARCH_SET_FS, 0x78b0993fb740) = 0

set_tid_address(0x78b0993fba10) = 144957

set_robust_list(0x78b0993fba20, 24) = 0

rseq(0x78b0993fc0e0, 0x20, 0, 0x53053053) = 0

mprotect(0x78b099216000, 16384, PROT_READ) = 0

mprotect(0x5bc46e4c6000, 4096, PROT_READ) = 0

mprotect(0x78b099452000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

munmap(0x78b0993fe000, 105007) = 0

getrandom("\x48\xad\x1b\x58\x6a\x67\xff\x23", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x5bc46e798000

brk(0x5bc46e7b9000) = 0x5bc46e7b9000

openat(AT_FDCWD, "./libmckusickCarels.so", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0775, st_size=15480, ...}, AT_EMPTY_PATH) = 0

getcwd("/home/denis/projects_c++/os/lab4", 128) = 33

mmap(NULL, 16432, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x78b099413000

mmap(0x78b099414000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x78b099414000

mmap(0x78b099415000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x78b099415000

mmap(0x78b099416000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x78b099416000

close(3) = 0

mprotect(0x78b099416000, 4096, PROT_READ) = 0

mmap(NULL, 8388608, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x78b098800000

write(1, "Test 1: Memory allocated (512 by"..., 37Test 1: Memory allocated (512 bytes)

) = 37

write(1, "Test 1: Memory freed (512 bytes)"..., 33Test 1: Memory freed (512 bytes)

) = 33

write(1, "Test 2: Allocation failed (expec"..., 59Test 2: Allocation failed (expected for oversized request)

) = 59

write(1, "Test 3: Memory allocated (1024 b"..., 38Test 3: Memory allocated (1024 bytes)

) = 38

write(1, "Test 3: Memory freed (1024 bytes"..., 34Test 3: Memory freed (1024 bytes)

) = 34

write(2, "Test 4: Memory allocation failed"..., 33Test 4: Memory allocation failed

) = 33

write(2, "Test 5: Memory allocation failed"..., 33Test 5: Memory allocation failed

) = 33

write(1, "Allocator destroyed\n", 20Allocator destroyed

) = 20

munmap(0x78b098800000, 8388608) = 0

munmap(0x78b099413000, 16432) = 0

```
exit_group(0)                           = ?
+++ exited with 0 +++
```