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: Попов А.В.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 26.12.24

Москва, 2024

Постановка задачи

Вариант 3.

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 (см. пример на GitHub Gist).

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);

Вариант 3

Списки свободных блоков (первое подходящее) и алгоритм двойников

Общий метод и алгоритм решения

Списки свободных блоков

Данный алгоритм основан на хранении блоков свободной памяти в списке. Когда пользователь запрашивает новый блок памяти, алгоритм проходится по списку свободных блоков и находит первый подходящий блок. Если блок размера большего, чем нужный, то он разделяется на две части, ненужная часть добавляется в список свободных блоков. При очищении блок добавляется в список свободных блоков и также может выполняться слияние с соседними блоками, если они свободны.

Алгоритм двойников

Все блоки памяти имеют размер степени двойки. Когда нужно выделить новый блок, алгоритм проходится по массиву списков из свободных блоков размера 2^n, где n – индекс в этом массиве, и находит ближайший блок к нужному размеру. Если найденный блок больше, то он делится пополам, пока он не достигнет ближайшего возможного размера. Ненужные половинки добавляются в массив списков свободных блоков.

Код программы

main.c

```c
/**
 * @file
 * @brief
 * @details
 * @author xsestech
 * @date 23.12.2024
 */

#include <liballoc/alloc.h>
#include <liballoc/fallback.h>
#include <libio/io.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <sys/time.h>

#include "timer.h"
```

```c
#define ALLOC_MEMORY_SIZE 33554732 // 32MB

int main(int argc, char **argv) {
  (void) argc;

  if (argc != 2) {
    print_fd(STDERR_FILENO, "Error: wrong number of args\n");
    return -1;
  }
  allocator_lib_t lib;
  if (allocator_lib_load(&lib, argv[1]) == -1) {
    print_fd(STDERR_FILENO, "Error: couldn't load allocator");
    return -1;
  }
  void *memory = mmap(NULL, ALLOC_MEMORY_SIZE, PROT_WRITE | PROT_READ, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
  if (memory == MAP_FAILED) {
    print_fd(STDERR_FILENO, "Error: can't allocate memory\n");
    return -1;
  }
  void* allocator = lib.allocator_create(memory, ALLOC_MEMORY_SIZE);
  if (allocator == NULL) {
    print_fd(STDERR_FILENO, "Alloc error\n");
    munmap(memory, ALLOC_MEMORY_SIZE);
    return -1;
  }
  srand(0);
  timer_t timer_alloc, timer_free;

  const size_t mems_size = ALLOC_MEMORY_SIZE / 1024 * 0.8;
  void** mems[mems_size];
  timer_start(&timer_alloc);
  for (size_t i = 0; i < mems_size; ++i) {
    mems[i] = lib.allocator_alloc(allocator, 600);
    if (!mems[i]) {
      print_fd(STDERR_FILENO, "Error during allocation");
      lib.allocator_destroy(allocator);
      munmap(memory, ALLOC_MEMORY_SIZE);
      return -1;
    }
  }
  timer_end(&timer_alloc);
  print_fd(STDOUT_FILENO, "Allocation time:\n");
  timer_print(&timer_alloc);
  size_t deallocated = 0;
  for (size_t i = 0; i < mems_size; ++i) {
    if (rand()  % 2 == 0) {
      lib.allocator_free(allocator, mems[i]);
      deallocated++;
    }
  }
  timer_start(&timer_free);
  deallocated *= 0.8;
  for (size_t i = 0; i < deallocated / 2; ++i) {
    if (!lib.allocator_alloc(allocator, 512)) {
      print_fd(STDERR_FILENO, "Error during allocation");
      lib.allocator_destroy(allocator);
      munmap(memory, ALLOC_MEMORY_SIZE);
```

```
      return -1;
    }
  }
  timer_end(&timer_free);
  print_fd(STDOUT_FILENO, "Free time:\n");
  timer_print(&timer_free);

  size_t allocated_count = 0;
  while(lib.allocator_alloc(allocator, 256)) {
    allocated_count++;
  }

  print_fd(STDOUT_FILENO, "Max 256 allocation count: %lu", allocated_count);
  lib.allocator_destroy(allocator);



  munmap(memory, ALLOC_MEMORY_SIZE);
  return 0;
}
}
```

buddy.c

```
/**
 * @file
 * @brief
 * @details
 * @author xsestech
 * @date 19.12.2024
 */
#include <stdbool.h>
#include <liballoc/buddy/buddy.h>

#define MAX_SIZE_T_POW2 32
#define MIN_BLOCK_SIZE_POW2 5

#define get_block_mem_from_meta(block) \
  &(block[1])

#define get_meta_from_mem_ptr(mem) \
  &(((allocator_block_meta_t*)mem)[-1])

#define get_next_block_ptr(block, new_size) \
(void*)((char*)block + (1 << new_size))

typedef struct allocator_block_meta_t allocator_block_meta_t;

struct allocator_block_meta_t {
  size_t size;
  allocator_block_meta_t *prev_in_mem;
  allocator_block_meta_t *prev;
  allocator_block_meta_t *next;
};

struct allocator_t {
  allocator_block_meta_t *pow2blocks[MAX_SIZE_T_POW2];
  void *max_ptr;
};
```

```
allocator_t *allocator_create(void *const memory, size_t size) {
  if (!memory) {
    return NULL;
  }
  if (size < sizeof(allocator_t)) {
    return NULL;
  }
  allocator_t *alloc = memory;
  alloc->max_ptr = (char *) memory + size;
  allocator_block_meta_t *next_free = memory + sizeof(allocator_t);
  next_free->size = log2(size - sizeof(allocator_t));
  next_free->next = NULL;
  next_free->prev_in_mem = NULL;
  for (size_t i = 0; i < MAX_SIZE_T_POW2; ++i) {
    alloc->pow2blocks[i] = NULL;
  }
  alloc->pow2blocks[next_free->size] = next_free;
  return alloc;
}
void allocator_destroy(allocator_t *const allocator) {
  for (size_t i = 0; i < MAX_SIZE_T_POW2; ++i) {
    allocator->pow2blocks[i] = NULL;
  }
  allocator->max_ptr = NULL;
}

static void remove_from_pow2blocks(allocator_t *const allocator, allocator_block_meta_t
*block, size_t level) {
  if (block->prev) {
    block->prev->next = block->next;
  } else {
    allocator->pow2blocks[level] = block->next;
  }
  if (block->next) {
    block->next->prev = block->prev;
  }
}

static void add_to_pow2blocks(allocator_t *const allocator, allocator_block_meta_t
*block, size_t level) {
  block->next = allocator->pow2blocks[level];
  block->prev = NULL;
  if (allocator->pow2blocks[level]) {
    allocator->pow2blocks[level]->prev = block;
  }
  allocator->pow2blocks[level] = block;
}

void *allocator_alloc(allocator_t *const allocator, const size_t size) {
  size_t pow2size = ceil(log2(size + sizeof(allocator_block_meta_t)));
  if (pow2size < MIN_BLOCK_SIZE_POW2) {
    pow2size = MIN_BLOCK_SIZE_POW2;
  }
  allocator_block_meta_t *block = NULL;
  size_t closest_idx;
  for (closest_idx = pow2size; closest_idx < MAX_SIZE_T_POW2; ++closest_idx) {
    if (allocator->pow2blocks[closest_idx]) {
```

```
      block = allocator->pow2blocks[closest_idx];
      break;
    }
  }
  if (block == NULL) {
    return NULL;
  }
  if (closest_idx == pow2size) {
    remove_from_pow2blocks(allocator, block, pow2size);
    block->next = NULL;
    block->prev = NULL;
    return get_block_mem_from_meta(block);
  }
  while (closest_idx > pow2size) {
    remove_from_pow2blocks(allocator, block, closest_idx);
    size_t new_size = block->size - 1;
    allocator_block_meta_t *new_block = get_next_block_ptr(block, new_size);
    new_block->size = new_size;
    new_block->next = block;
    new_block->prev_in_mem = block;
    new_block->prev = NULL;

    block->size = new_size;
    block->prev = new_block;
    block->prev_in_mem = NULL;
    add_to_pow2blocks(allocator, new_block, new_size);
    closest_idx--;
  }
  remove_from_pow2blocks(allocator, block, pow2size);
  block->next = NULL;
  block->prev = NULL;
  return get_block_mem_from_meta(block);
}

allocator_block_meta_t *allocator_merge(allocator_t *const allocator,
allocator_block_meta_t *block, bool *was_merged) {
  *was_merged = false;

  // Merge with the previous block if possible
  if (block->prev_in_mem && block->prev_in_mem->size == block->size &&
    get_next_block_ptr(block->prev_in_mem, block->prev_in_mem->size) == block) {
    remove_from_pow2blocks(allocator, block->prev_in_mem, block->size);
    block->prev_in_mem->size++;
    block = block->prev_in_mem;
    *was_merged = true;
  }

  // Merge with the next block if possible
  allocator_block_meta_t *next_block = get_next_block_ptr(block, block->size);
  if ((void *) next_block < allocator->max_ptr && next_block->size == block->size &&
    (next_block->next || next_block == allocator->pow2blocks[next_block->size])) {
    remove_from_pow2blocks(allocator, next_block, block->size);
    block->size++;
    *was_merged = true;
  }

  return block;
}
```

```
void allocator_free(allocator_t *const allocator, void *const memory) {
  if (!allocator || !memory) {
    return;
  }
  allocator_block_meta_t *block = get_meta_from_mem_ptr(memory);
  bool was_merged = true;
  while (was_merged) {
    block = allocator_merge(allocator, block, &was_merged);
  }
  add_to_pow2blocks(allocator, block, block->size);
}
```

io.c

```
/**
 * @file
 * @brief
 * @details
 * @author xsestech
 * @date 27.10.2024
 */

#include <libio/io.h>

ssize_t print_fd(const int fd, char *fmt, ...) {
  va_list args;
  va_start(args, fmt);
  char buff[IO_MAX_STR_LEN];
  size_t len = vsnprintf(buff, IO_MAX_STR_LEN - 1, fmt, args);
  const ssize_t writen_bytes = write(fd, buff, len);
  va_end(args);
  return writen_bytes;
}
ssize_t write_str(const int fd, const char *buff) {
  return write(fd, buff, strlen(buff));
}

ssize_t reads_fd(const int fd, char *buff, const size_t buff_size) {
  ssize_t read_bytes = 0;
  return read(fd, buff, buff_size);
}
```

child.c

```
/**
 * @file
 * @brief
 * @details
 * @author xsestech
 * @date 26.10.2024
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <libconfig/config.h>

#include <libio/io.h>
```

```c
int main(const int argc, char *argv[]) {
  if (argc != 2) {
    print_fd(STDERR_FILENO, "No file specified");
    exit(EXIT_FAILURE);
  }
  const pid_t pid = getpid();
  int file = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC | O_APPEND, 0600);

  if (file == -1) {
    print_fd(STDERR_FILENO, "%d: Error opening file %s\n", pid, argv[1]);
    exit(EXIT_FAILURE);
  }

  print_fd(STDOUT_FILENO, "%d: opened file %s\n", getpid(), argv[1]);
  char buffer[MAX_LINE_LENGTH];
  ssize_t bytes = 0;
  while ((bytes = read(STDIN_FILENO, buffer, MAX_LINE_LENGTH)) > 0) {
    if (bytes == -1) {
      print_fd(STDERR_FILENO, "%d: Error reading from pipe\n", pid);
      exit(EXIT_FAILURE);
    }
    if (buffer[0] == '\n') {
      break;
    }


    buffer[bytes - 1] = '\0'; // remove newline
    print_fd(STDOUT_FILENO, "%d: got: %s\n", pid, buffer);
    if (write(file, buffer, bytes - 1) != bytes - 1) {
      print_fd(STDERR_FILENO, "%d: Error writing to file\n", pid);
      exit(EXIT_FAILURE);
    }
  }
  const char term = '\0';
  write(file, &term, sizeof(term));
  close(file);
  return 0;
}
```

Freeblocks.c

```c
/**
 * @file
 * @brief
 * @details
 * @author xsestech
 * @date 19.12.2024
 */
#include <liballoc/freeblocks/freeblocks.h>

#define get_block_mem_from_meta(block) \
  &(block[1])

#define get_meta_from_mem_ptr(mem) \
  &(((allocator_block_meta_t*)mem)[-1])

#define get_next_block_ptr(block, new_size) \
(void*)((char*)get_block_mem_from_meta(block) + new_size - 1)
```

```c
typedef struct allocator_block_meta_t allocator_block_meta_t;

struct allocator_block_meta_t {
  size_t size;
  allocator_block_meta_t *prev_in_mem;
  allocator_block_meta_t *prev;
  allocator_block_meta_t *next;
};

struct allocator_t {
  allocator_block_meta_t *next_free;
  void *max_ptr;
};

allocator_t *allocator_create(void *const memory, size_t size) {
  if (!memory) {
    return NULL;
  }
  if (size < sizeof(allocator_t)) {
    return NULL;
  }
  allocator_t *alloc = memory;
  alloc->max_ptr = (char *) memory + size;
  allocator_block_meta_t *next_free = get_next_block_ptr(memory, sizeof(allocator_t));
  next_free->size = size - sizeof(allocator_t) - sizeof(allocator_block_meta_t);
  next_free->next = NULL;
  next_free->prev_in_mem = NULL;
  alloc->next_free = next_free;
  return alloc;
}
void allocator_destroy(allocator_t *const allocator) {
  allocator->next_free = NULL;
  allocator->max_ptr = NULL;
}

void *allocator_alloc(allocator_t *const allocator, const size_t size) {
  allocator_block_meta_t *block = allocator->next_free;
  allocator_block_meta_t *prev_block = NULL;
  if (block == NULL) {
    return NULL;
  }
  while (block->next && block->size < size) {
    prev_block = block;
    block = block->next;
  }
  if (block->size < size) {
    return NULL;
  }
  if (block->size < size + sizeof(allocator_block_meta_t)) {
    if (block->next) {
      allocator->next_free = block->next;
      block->next->prev = NULL; // Update prev pointer
    }
    block->next = NULL;
    return get_block_mem_from_meta(block);
  }
  allocator_block_meta_t *new_free_block = get_next_block_ptr(block, size);
```

```
  new_free_block->size = block->size - size - sizeof(allocator_block_meta_t);
  new_free_block->next = block->next;
  new_free_block->prev_in_mem = block;
  new_free_block->prev = prev_block;
  if (prev_block != NULL) {
    prev_block->next = new_free_block;
  } else {
    allocator->next_free = new_free_block;
  }
  if (new_free_block->next) {
    new_free_block->next->prev = new_free_block;
  }
  block->size = size;
  block->next = NULL;
  return get_block_mem_from_meta(block);
}

allocator_block_meta_t *allocator_merge(allocator_t *const allocator,
allocator_block_meta_t *block) {
  if (block->prev_in_mem && get_next_block_ptr(block->prev_in_mem, block->prev_in_mem-
>size) == block) {
    block->prev_in_mem->size += block->size + sizeof(allocator_block_meta_t);
    block->prev_in_mem->prev = block->prev_in_mem->next;
    block->prev_in_mem->next = block->next;
    block = block->prev_in_mem;
  }
  allocator_block_meta_t *next_block = get_next_block_ptr(block, block->size);
  if ((void *) next_block < allocator->max_ptr && (next_block->next || next_block ==
allocator->next_free)) {
    block->size += next_block->size + sizeof(allocator_block_meta_t);
  }
  return block;
}

void allocator_free(allocator_t *const allocator, void *const memory) {
  allocator_block_meta_t *block = get_meta_from_mem_ptr(memory);
  block->next = allocator->next_free;
  if(allocator->next_free) {
    allocator->next_free->prev = block;
  }
  block = allocator_merge(allocator, block);
  allocator->next_free = block;
}
```

Alloc.c

```
/**
 * @file
 * @brief
 * @details
 * @author xsestech
 * @date 25.12.2024
 */
#include <liballoc/alloc.h>

void allocator_lib_assign_fallbacks(allocator_lib_t* lib) {
  lib->is_fallback = true;
  lib->allocator_alloc = allocator_alloc_fallback;
  lib->allocator_free = allocator_free_fallback;
```

```
  lib->allocator_destroy = allocator_destroy_fallback;
  lib->allocator_create = allocator_create_fallback;
}

void allocator_lib_load_func(allocator_lib_t *lib, void **pointer_to_func_field, const
char *func_name) {
  if(!lib->is_fallback) {
    *pointer_to_func_field = dlsym(lib->handle, func_name);
    if (!lib->allocator_alloc) {
      print_fd(STDERR_FILENO, "Warning: Could not load function %s, using fallbacks:
%s", func_name, dlerror());
      dlclose(lib->handle);
      lib->handle = NULL;
      allocator_lib_assign_fallbacks(lib);
    }
  }
}


int allocator_lib_load(allocator_lib_t *lib, const char *path) {
  if (!lib || !path) {
    print_fd(STDERR_FILENO, "Error: Path or lib pointer can't be null");
    return -1;
  }
  lib->handle = dlopen(path, RTLD_LOCAL | RTLD_NOW);
  if (!lib->handle) {
    print_fd(STDERR_FILENO, "Warning: Could not load library, using fallback functions:
%s\n", dlerror());
    allocator_lib_assign_fallbacks(lib);
    return 0;
  }

  dlerror();
  allocator_lib_load_func(lib, (void**)&lib->allocator_alloc, "allocator_alloc");
  allocator_lib_load_func(lib, (void**)&lib->allocator_free, "allocator_free");
  allocator_lib_load_func(lib, (void**)&lib->allocator_destroy, "allocator_destroy");
  allocator_lib_load_func(lib, (void**)&lib->allocator_create, "allocator_create");
  return 0;
}
void allocator_lib_unload(allocator_lib_t *lib) {
  if (lib->handle) {
    dlclose(lib->handle);
    lib->handle = NULL;
  }
}
```

Протокол работы программы

Методика тестирования:

Для сравнения аллокаторов мы замерим время очищения памяти и ее выделения, а также для оценки эффективности использования памяти мы замерим, сколько блоков размера 256 будет возможно выделить при заполнении памяти на 80%. Для замера времени выделения будем выделять блоки размера 600, а вот очищение будет происходить случайным образом, но с зафиксированным стартовым числом генератора. Это позволит сравнить затраты на слияние памяти.

Тестирование:

Buddy:

alloc cmake-build-debug/liballoc-buddy/libliballoc-buddy.dylib

Allocation time:

Elapsed time: 0.002843

Free time:

Elapsed time: 0.000160

Max 256 allocation count: 14463

Freeblocks prev_ptr:

alloc cmake-build-debug/liballoc-freeblocks/libliballoc-freeblocks.dylib

Allocation time:

Elapsed time: 0.001596

Free time:

Elapsed time: 0.042390

Max 256 allocation count: 100950

Freeblocks:

Allocation time:

Elapsed time: 0.001640

Free time:

Elapsed time: 0.090969

Max 256 allocation count: 74722

Вывод

Как мы можем видеть алгоритм двойников медленнее выделяет память, т.к. производится разделение блоков, но быстрее ее очищает. Фрагментация у алгоритма двойников выше, т.к. много памяти теряется на выделении блоков размера степени двойки. При добавлении указателя на предыдущий блок в памяти алгоритм свободных блоков фрагментируется еще меньше, но скорость очищения уменьшается в два раза.