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-211Б-23

Студент: Бугренков В.П. Преподаватель: Бахарев В.Д. Оценка:

Дата: 22.12.24

Москва, 2024

# Постановка задачи

Вариант 4.

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 (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);

void allocator\_free(Allocator \*const allocator, void \*const memory) (возвращает выделенную память аллокатору);

Необходимо реализовать:

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

# Общий метод и алгоритм решения

Использованные системные вызовы:

* pid\_t fork(void); – создает дочерний процесс.
* int shm\_open(const char \*\_\_name, int \_\_oflag, mode\_t \_\_mode) — открывает сегмент shm
* void \*mmap(void \*\_\_addr, size\_t \_\_len, int \_\_prot, int \_\_flags, int \_\_fd, off\_t \_\_offset) — создает новый маппинг в виртуальном адресном пространстве
* sem\_t \*sem\_open (const char \*\_\_name, int \_\_oflag, …) - открывает именнованный семафор
* int sem\_unlink (const char \*\_\_name) — удаляет именованный семафор
* int sem\_wait(sem\_t \*sem) - уменьшает (блокирует) семафо
* int sem\_post(sem\_t \*sem) - увеличивает (разблокирует) семафор
* int open(const char \*pathname, int flags, mode\_t mode) - открытие\создание файла
* int close(int fd) - закрыть файл
* void exit(int status) - завершения выполнения процесса и возвращение статуса
* int execv(const char \*filename, char \*const argv[]) - замена образа памяти процесса
* pid\_t getpid(void) — получение ID процесса
* ssize\_t read(int \_\_fd, void\* \_\_buf, size\_t \_\_nbytes) — чтение из fd в буфер
* ssize\_t write(int \_\_fd, const void\* \_\_buf, size\_t \_\_n) — запись байтов в буфер

В решении задачи используется взаимодействие между родительским и дочерним процессами через общую память (shared memory) и синхронизацию через семафоры. Родительский процесс выполняет следующие шаги:

1. Открывает файл с помощью системной функции fopen.
2. Создает объект общей памяти с помощью shm\_open, задает его размер с помощью ftruncate и мапирует его в пространство памяти с помощью mmap.
3. Создает семафор с помощью sem\_open для синхронизации доступа к общей памяти между процессами.
4. Читает строки из файла и записывает их в общую память с использованием системного вызова memcpy или прямым обращением к области памяти.
5. Сигнализирует дочернему процессу с помощью sem\_post, что данные готовы для обработки.

Дочерний процесс выполняет следующие шаги:

1. Создает локальную ссылку на общую память с помощью mmap, где он будет получать данные от родительского процесса.
2. Ожидает, пока родительский процесс запишет данные в общую память, с помощью sem\_wait (блокировка до получения данных).
3. Читает строку из общей памяти, разбивает ее на токены и выполняет операцию деления чисел, проверяя деление на ноль.
4. После выполнения вычислений выводит результат с помощью системных вызовов write или writev.
5. После завершения работы с данными, дочерний процесс отправляет сигнал родительскому процессу о завершении с помощью sem\_post.

Завершающие шаги:

1. Родительский процесс получает уведомление от дочернего, что данные обработаны, и повторяет цикл до конца

# Код программы

**parent.c**

#include <stdio.h>  
*#include* <stdlib.h>  
*#include* <unistd.h>

# Протокол работы программы

qwental@DESKTOP-NKF1EUK:~/workspace/OS\_LABS/lab3/src$ gcc child.c -o child

Division result is: 6.250000

**Strace:**

qwental@DESKTOP-NKF1EUK:~/workspace/OS\_LABS/lab3/src$ strace ./parent test.txt

+++ exited with 0 +++

# Вывод

В ходе лабораторной работы я приобрел практические навыки