|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ  Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-4 «Интеллектуальные системы информационной безопасности»

**Алгоритмы параллельных вычислений**

**Лабораторная работа №1**

**ОТЧЕТ**

Выполнил студент группы:

БСБО-07-20

Любовский С.В.

**Москва 2022**

**Постановка задачи**

**Задача 2.4.6.1**

**Условие**

Требуется разработать программу, выполняющую следующие действия:

1. Отображение файла в память (имя файла передается первым аргументом командной строки).
2. Изменение порядка следования байт в отображаемом блоке памяти на противоположный.
3. Отключение отображения файла в память.

**Требования**

**Оформление программы**

1. Программа должна являться консольным приложением на языке C или C++
2. Программа должна разрабатывать для OC UNIX или Windows.

**Используемые функции**

1. В программе могут использоваться вызовы следующих функций.
   1. UNIX: open(), lseek(), mmap(), munmap(), close().
2. Вызовы функций должны сопровождаться проверкой возвращаемых значений.

**Формат входных данных**

Входных данные состоят из имени файла, передаваемого программе первым аргументов командой строки.

**Формат выхода**

Выходные данные отсутствуют

**Проверка корректности**

Проверка производится посредством создания текстового файла и вывода его на консоль до и после работы программы (порядок следования байт должен измениться на противоположный). Детальная проверка выполняется посредством просмотра и оценки исходного кода программы.

**Метод решения**

Будем использовать OC Linux (UNIX), стандартную библиотеку *sys/mman.h* для работы с отображением файла в память. Для работы с файлом будем использовать вызовы из стандартной библиотеки *fcntl.h* и вызовы стандартных библиотек *unistd.h* и *sys/wait.h* для работы с процессами.

**Решение**

Решение будет как синхронным, так и асинхронным в зависимости от переданного параметра – количества параллельных процессов.

Алгоритм решения, следующий:

1. Считываем переданные аргументы: имя файла, в котором необходимо развернуть последовательность байт и количество параллельных процессов, где 1 – синхронное выполнение кода.
2. Открываем файл на чтение и запись.
3. Получаем размер файла и отображаем его в память.
4. Создаем буфер и копируем туда содержимое файла из памяти.
5. Считаем количество обрабатываемых байт на каждый процесс, а также остаточное количество байт после целочисленного деления.
6. Запускаем процессы. Каждый процесс знает свой порядковый номер.
7. Каждый процесс обрабатывает свой участок (берет информацию из буфера, разворачивает и вставляет в нужное место в отображенной памяти)
8. Родительский процесс ожидает завершения всех дочерних.
9. Закрываем отображение файла в память
10. Считаем затраченное время.

**Реализация программы**

Далее будет представлен листинг программы с комментариями.

#include <stdio.h>

#include <stdlib.h>

#include <sys/mman.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <sys/wait.h>

#include <errno.h>

#include <sys/time.h>

int main(int argc, char \*\*argv)

{

    // Проверяем наличие имени файла первым аргументом

    char \*filename;

    if (argc >= 2)

    {

        filename = argv[1];

    }

    else

    {

        printf("Usage: %s FILENAME PROCESS\_NUM\n", argv[0]);

        return -1;

    }

    // Проверяем наличие аргумента количества процессов

    int p\_num;

    if (argc >= 3)

    {

        p\_num = atoi(argv[2]);

    }

    else

    {

        printf("Usage: %s FILENAME PROCESS\_NUM\n", argv[0]);

        return -1;

    }

    // Открываем файл по переданному имени

    int fd = open(filename, O\_RDWR);

    if (fd < 0)

    {

        printf("Error opening file %s", filename);

        return -1;

    }

    // Получаем размер файла

    int size = lseek(fd, 0, SEEK\_END);

    // Отображаем файл в память

    char \*mm\_addr = mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);

    // По описанию mmap файловый дескриптор можно закрыть сразу после отображения

    close(fd);

    // Создаем дополнительный буфер для чтения

    char \*buf = malloc(size);

    // Копируем данные файла в буфер

    memcpy(buf, mm\_addr, size);

    // Рассчитываем какая часть файла будет обрабатываться каждым процессом

    int bytes\_per\_process = size / p\_num;

    int bytes\_remainder = size % p\_num;

    // Создаем структуру для записи времени

    struct timeval start;

    gettimeofday(&start, NULL);

    int p\_i = 0;

    // Создаем дочерние процессы

    for (int i = 1; i < p\_num; i++)

    {

        int f\_res = fork();

        if (f\_res == -1)

        {

            printf(“Error creating process %d”, p\_i);

            return -1;

        }

        else if (f\_res == 0)

        {

            p\_i = i;

            break;

        }

    }

    int offset = 0;

    if (bytes\_remainder && (p\_i == (p\_num – 1)))

    {

        offset = bytes\_remainder;

    }

    // Процесс обрабатывает свою часть файла

    for (int i = bytes\_per\_process + offset – 1; i >= 0; i--)

    {

        mm\_addr[bytes\_remainder + (p\_num – p\_i) \* bytes\_per\_process – 1 – i] = buf[i + p\_i \* bytes\_per\_process];

    }

    // Для родительского процесса ждем завершения всех дочерних

    if (p\_i == 0)

    {

        while (1)

        {

            int status;

            pid\_t done = wait(&status);

            if (done == -1)

            {

                if (errno == ECHILD)

                {

                    break;

                }

            }

            else

            {

                if (!WIFEXITED(status) || WEXITSTATUS(status) != 0)

                {

                    printf(“Error in child process %d\n”, done);

                    return -1;

                }

            }

        }

        // Закрываем отображение в память

        if (munmap(mm\_addr, size))

        {

            printf(“Error unmaping file %s”, filename);

            return -1;

        }

        // Освобождаем память под буфер

        free(buf);

        // Записываем время выполнения

        struct timeval stop;

        gettimeofday(&stop, NULL);

        printf(“took %lu us\n”, (stop.tv\_sec – start.tv\_sec) \* 1000000 + stop.tv\_usec – start.tv\_usec);

    }

    return 0;

}

**Результаты работы программы**

Все измерения проведены с фалом размером 26141343 Байт. Файл является достаточно большим для отслеживания разница во времени выполнения при параллельном исполнении. Для каждого значения процессов (1-24) проведено 100 экспериментов и получено среднее значение.

На графике можно увидеть, как с количеством процессов меняется скорость выполнения задачи. При одном процессе имеем значение около 100000 наносекунд, но уже при двух процессах время выполнения сокращается почти в 3 раза. Далее можно видеть постепенное уменьшение времени выполнения до точки, где количество процессов совпадает с количеством логических ядер процессора (в используемом для тестов ноутбуке их было 8). Далее можем видеть скачок во времени выполнения, его можно объяснить появлением затрат на переключения контекста процессора, т.к. процессы уже не могут выполняться параллельно. Далее снова видим снижение времени выполнения до точки 16. Тут выигрыш от параллельного выполнения двух задач на одном ядре смог компенсировать затраты на переключения контекста, но показатель все равно больше, чем при числе процессов равному числу логических ядер. На 17 процессах снова видим увеличение и плавное снижение к 24.

**Заключение**

В рамках этой лабораторной работы была реализована программа для изменения порядка следования байт в файле путем отображения его в память. Были проведены тесты с использованием одного и нескольких процессов. Полученные в ходе экспериментов данные были описаны и показывают ускорение работы программы при параллельной реализации.ы