Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе №5

на тему

**УПРАВЛЕНИЕ ПОТОКАМИ, СРЕДСТВА СИНХРОНИЗАЦИИ**

Студент В. М. Вергасов

Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 4](#_Toc146752069)

[3 Результат выполнения 6](#_Toc146752070)

[Заключение 6](#_Toc146752071)

[Список использованных источников 7](#_Toc146752072)

[Приложение А (обязательное) Листинг кода 8](#_Toc146752073)

1. **ЦЕЛЬ РАБОТЫ**

Изучение подсистемы потоков (pthread), основных особенностей функционирования и управления, средств взаимодействия потоков.

Практическое проектирование, реализация и отладка программ с параллельными взаимодействующими (конкурирующими) потоками.

Реализация программы, которая сортирует большой массив данных, получаемый из файла в нескольких потоках.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Многопоточность – это возможность операционной системы эффективно обрабатывать и управлять несколькими потоками выполнения внутри одного процесса. Linux предоставляет мощные механизмы для работы с многопоточностью, которые позволяют разработчикам создавать многопоточные программы с высокой производительность.

Поток выполнения (англ. thread – нить) – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени) [1].

У всех исполняемых процессов есть как минимум один поток исполнения. Некоторые процессы этим и ограничиваются в тех случаях, когда дополнительные нити исполнения не дают прироста производительности, но только усложняют программу. Однако таких программ с каждым днем становится относительно меньше.

Практически все современные ОС – включая Windows, Linux, Mac OS X, и Solaris – поддерживают управление потоками в режиме ядра. Однако потоки могут быть созданы не только в режиме ядра, но и в режиме пользователя. При использовании этого уровня ядро не знает о существовании потоков – все управление потоками реализуется приложением с помощью специальных библиотек. Пользовательские потоки по-разному отображаются на потоки в режиме ядра. Всего существует три модели, из которых 1:1 является наиболее часто используемой [2].

1. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В результате работы была создана программа, пример результата запуска программы рисунке 1:

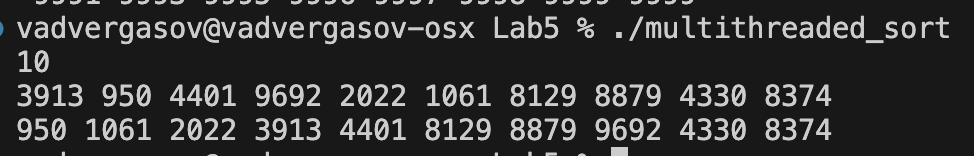


Рисунок 1 – Вывод программы при запуске

Мы можем указать размер массива, который сгенерируется и после чего отсортируется в многопоточном режиме (рисунок 2):

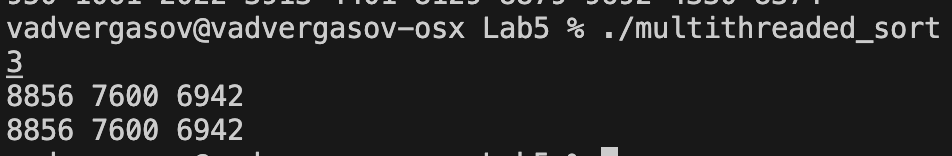


Рисунок 2 – Результат отправки сигнала процессу

# ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы был реализован программный продукт, выполняющий многопоточную сортировку.

Были изучены основные принципы работы многопоточности в Linux, изучены возможности взаимодействия между потоками.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Unix2018/Потоки [Электронный ресурс]. – Режим доступа: https://acm.bsu.by/wiki/Unix2018/Потоки. – Дата доступа: 26.03.2024.
2. Pthreads [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/326138/. – Дата доступа: 26.03.2024.

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – main.c

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <stdlib.h>

#define NUM\_THREADS 4

typedef struct {

int\* array;

int start;

int end;

} ThreadArgs;

int compare(const void\* a, const void\* b) { return (\*(int\*)a - \*(int\*)b); }

void\* sort(void\* arg) {

ThreadArgs\* args = (ThreadArgs\*)arg;

qsort(args->array + args->start, args->end - args->start + 1, sizeof(int),

compare);

pthread\_exit(NULL);

}

void merge(int\* array, int start, int mid, int end) {

int leftSize = mid - start + 1;

int rightSize = end - mid;

int\* leftArray = malloc(leftSize \* sizeof(int));

int\* rightArray = malloc(rightSize \* sizeof(int));

for (int i = 0; i < leftSize; i++) {

leftArray[i] = array[start + i];

}

for (int i = 0; i < rightSize; i++) {

rightArray[i] = array[mid + 1 + i];

}

int i = 0, j = 0, k = start;

while (i < leftSize && j < rightSize) {

if (leftArray[i] <= rightArray[j]) {

array[k++] = leftArray[i++];

} else {

array[k++] = rightArray[j++];

}

}

while (i < leftSize) {

array[k++] = leftArray[i++];

}

while (j < rightSize) {

array[k++] = rightArray[j++];

}

free(leftArray);

free(rightArray);

}

// Функция для создания и запуска потоков

void createThreads(pthread\_t threads[], ThreadArgs threadArgs[], int\* array,

int chunkSize) {

for (int i = 0; i < NUM\_THREADS; i++) {

threadArgs[i].array = array;

threadArgs[i].start = i \* chunkSize;

threadArgs[i].end = (i + 1) \* chunkSize - 1;

pthread\_create(&threads[i], NULL, sort, (void\*)&threadArgs[i]);

}

}

int main() {

srand(time(NULL));

int size;

scanf("%d", &size);

int\* array = (int\*)malloc(size \* sizeof(int));

for (int i = 0; i < size; i++) {

array[i] = rand() % 10000;

}

for (int i = 0; i < size; i++) {

printf("%d ", array[i]);

}

printf("\n");

int chunkSize = size / NUM\_THREADS;

pthread\_t threads[NUM\_THREADS];

ThreadArgs threadArgs[NUM\_THREADS];

createThreads(threads, threadArgs, array, chunkSize);

for (int i = 0; i < NUM\_THREADS; i++) {

pthread\_join(threads[i], NULL);

}

for (int i = 1; i < NUM\_THREADS; i++) {

merge(array, 0, i \* chunkSize - 1, (i + 1) \* chunkSize - 1);

}

for (int i = 0; i < size; i++) {

printf("%d ", array[i]);

}

printf("\n");

return 0;

}

Листинг 4 – Makefile

CC = gcc

CFLAGS = -Wall -Wextra -pthread

TARGET = multithreaded\_sort

all: $(TARGET)

$(TARGET): main.o

$(CC) $(CFLAGS) -o $(TARGET) main.o

main.o: main.c

$(CC) $(CFLAGS) -c main.c

clean:

rm -f $(TARGET) \*.o