

Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра ПОИТ

Дисциплина «Архитектура компьютерной техники и
операционных систем»

ОТЧЁТ
к лабораторной работе №7

ПРОЦЕССЫ И ПОТОКИ В ОС LINUX

Вариант 8

Студент группы №351001
Ушаков А.Д.
Преподаватель
Леванцевич В.А.

Минск 2024

СОДЕРЖАНИЕ

Введение	3
1 Задание к работе	4
1.1 Требования к выполнению	4
1.2 Индивидуальное задание.....	4
2 Реализация с использованием процессов	5
2.1 Код программы.....	5
2.2 Запуск исполняемого файла.....	9
2.3 Результат работы программы	9
3 Реализация с использованием потоков	10
3.1 Код программы.....	10
3.2 Запуск исполняемого файла.....	14
3.3 Результат работы программы	14
Заключение	15
Список использованных источников	16

ВВЕДЕНИЕ

Процесс в операционной системе Linux – это экземпляр программы, который выполняется с набором инструкций и данных. Каждый процесс обладает своим уникальным идентификатором (PID), и, несмотря на то что процессы могут выполняться параллельно, ОС предоставляет каждому процессу иллюзию выделенного процессорного времени.

Поток представляет собой более легковесную единицу выполнения, являясь частью процесса. Несколько потоков одного процесса могут совместно использовать память и ресурсы, что позволяет эффективно управлять задачами, которые требуют параллельной обработки.

Образ процесса (Process Image) представляет собой совокупность всех данных, необходимых для выполнения процесса, таких как код, данные, открытые файлы и окружение. Он включает в себя системный регистровый и пользовательский контексты. Системный контекст (или контекст ядра) содержит данные, управляющие работой процесса на уровне ядра ОС, такие как содержимое регистров процессора и адреса памяти. Пользовательский контекст – это данные и инструкции, находящиеся в пользовательской области памяти, доступные процессу при выполнении операций. Совокупность этих контекстов позволяет системе приостанавливать и возобновлять процесс в нужное время, сохраняя его состояние.

Диаграмма состояний процесса описывает различные этапы, которые может проходить процесс, начиная с момента создания до завершения. Основные состояния процесса включают: новый (создание), готовый (ожидание доступа к процессору), исполняемый (выполнение инструкций на процессоре), ожидающий (ожидание ввода-вывода) и завершённый (окончание работы). ОС управляет переходом между этими состояниями, используя планировщик, который решает, какой процесс будет следующим выполнять свои задачи, оптимизируя использование ресурсов системы.

Для создания нового процесса в Linux используется системный вызов `fork()`, который создает дочерний процесс – полную копию родительского процесса. Этот вызов является уникальным для Unix-подобных ОС. В процессе копирования `fork()` использует механизм «копирования при записи» (Copy-on-Write, CoW), который оптимизирует выделение памяти. При создании дочернего процесса вместо немедленного копирования памяти родительского процесса дочернему, ОС позволяет обоим процессам совместно использовать одну и ту же область памяти до тех пор, пока один из них не попытается изменить её. Лишь тогда происходит фактическое копирование, что значительно экономит ресурсы и ускоряет выполнение операций.

1 ЗАДАНИЕ К РАБОТЕ

1.1 Требования к выполнению

Написать программу на языке C, согласно варианту задания.

Минимальные требования к отчету:

Отчет оформляется в бумажном виде.

Отчет должен содержать титульный лист с фамилией и номером варианта.

В отчете должны присутствовать: текст задания и листинг программы.

Варианты заданий выбираются по списку подгруппы. Первые девять человек выбирают номера 1-9, вторые девять человек также выбирают номера 1-9 и т.д.

1.2 Индивидуальное задание

По двум частям лабораторной работы выполняется одно и то же задание: для ЛР 7.1 с использованием процессов, для ЛР 7.2 с использованием потоков.

К варианту 8 относится: написать программу на C поиска одинаковых по содержимому файлов в двух каталогов, например, Dir1 и Dir2. Пользователь задаёт имена Dir1 и Dir2. В результате работы программы файлы, имеющиеся в Dir1, сравниваются с файлами в Dir2 по их содержимому. Процедуры сравнения должны запускаться с использованием функции `fork()` в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой `pid`, имя файла, число просмотренных байт и результаты сравнения. Число запущенных процессов любой момент времени не должно превышать N (вводится пользователем).

2 РЕАЛИЗАЦИЯ С ИСПОЛЬЗОВАНИЕМ ПРОЦЕССОВ

2.1 Код программы

```
#include <stdio.h>      /* Подключение библиотеки для работы с вводом-
выводом */
#include <stdlib.h>      /* Подключение стандартной библиотеки для
использования функций, таких как atoi() и exit() */
#include <string.h>      /* Подключение библиотеки для работы со
строками и функцией memcmp() */
#include <unistd.h>      /* Подключение POSIX-библиотеки для
использования fork(), getpid() и других функций */
#include <sys/types.h>   /* Подключение библиотеки для использования
типов данных, таких как pid_t */
#include <sys/wait.h>    /* Подключение библиотеки для работы с
функцией wait() для ожидания завершения процессов */
#include <dirent.h>      /* Подключение библиотеки для работы с
каталогами */
#include <fcntl.h>       /* Подключение библиотеки для работы с
файловыми дескрипторами и функцией open() */
#include <sys/stat.h>    /* Подключение библиотеки для использования
функции stat() для проверки типа файла */

#define BUFFER_SIZE 1024 /* Определение размера буфера для чтения
данных из файлов */
#define PATH_MAX 4096    /* Устанавливаем значение вручную, если
PATH_MAX не определён */

/* Функция для побайтового сравнения содержимого двух файлов */
int compare_files(const char *file1, const char *file2) {
    int fd1 = open(file1, O_RDONLY); /* Открытие первого файла в
режиме только для чтения */
    int fd2 = open(file2, O_RDONLY); /* Открытие второго файла в
режиме только для чтения */
    if (fd1 < 0 || fd2 < 0) {        /* Проверка, были ли файлы
открыты успешно */
        perror("Ошибка открытия файлов"); /* Вывод ошибки, если один
из файлов не открылся */
        return -1;                  /* Возврат -1 в случае
ошибки */
    }

    char buffer1[BUFFER_SIZE], buffer2[BUFFER_SIZE]; /* Буферы для
хранения прочитанных данных из файлов */
    ssize_t bytes_read1, bytes_read2; /* Переменные для количества
прочитанных байт из файлов */

    /* Чтение данных из обоих файлов и их сравнение */
    while (1) {
```

```

        bytes_read1 = read(fd1, buffer1, BUFFER_SIZE); /* Чтение
данных из первого файла */
        bytes_read2 = read(fd2, buffer2, BUFFER_SIZE); /* Чтение
данных из второго файла */

        /* Если файлы не одинаковой длины, возвращаем 0 */
        if (bytes_read1 != bytes_read2) {
            close(fd1); /* Закрытие первого файла */
            close(fd2); /* Закрытие второго файла */
            return 0; /* Возврат 0, указывая, что файлы не
совпадают */
        }

        /* Если прочитанные данные не совпадают, файлы разные */
        if (bytes_read1 == 0) {
            break; /* Если оба файла закончились, они одинаковы */
        }

        if (memcmp(buffer1, buffer2, bytes_read1) != 0) {
            close(fd1); /* Закрытие первого файла */
            close(fd2); /* Закрытие второго файла */
            return 0; /* Возврат 0, указывая, что файлы не
совпадают */
        }
    }

    close(fd1); /* Закрытие первого файла */
    close(fd2); /* Закрытие второго файла */
    return 1; /* Возврат 1, указывая, что файлы совпадают */
}

/* Основная функция программы */
int main(int argc, char *argv[]) {
    if (argc != 4) { /* Проверка, переданы ли три аргумента командной
строки */
        fprintf(stderr, "Использование: %s <Dir1> <Dir2> <N>\n",
argv[0]); /* Вывод инструкции по использованию */
        return 1; /* Возврат 1, указывая на ошибку */
    }

    const char *dir1 = argv[1]; /* Первый каталог для поиска файлов
*/
    const char *dir2 = argv[2]; /* Второй каталог для поиска файлов
*/
    int max_processes = atoi(argv[3]); /* Максимальное число
одновременно запущенных процессов */
    int active_processes = 0; /* Текущая активная работа процессов */

    DIR *d1 = opendir(dir1); /* Открытие первого каталога */
    if (!d1) { /* Проверка, открыт ли каталог */

```

```

        perror("Ошибка открытия Dir1"); /* Вывод ошибки, если не
удалось открыть каталог */
        return 1; /* Возврат 1, указывая на ошибку */
    }

    struct dirent *entry1; /* Структура для хранения информации о
файле в первом каталоге */
    struct stat statbuf1; /* Структура для хранения информации о
файле из stat() */

    while ((entry1 = readdir(d1)) != NULL) { /* Чтение всех файлов в
первом каталоге */
        char file1_path[PATH_MAX]; /* Полный путь к файлу из первого
каталога */
        snprintf(file1_path, PATH_MAX, "%s/%s", dir1, entry1->d_name);
/* Формирование пути */

        /* Используем stat для проверки, является ли текущий объект
обычным файлом */
        if (stat(file1_path, &statbuf1) == -1 ||
!S_ISREG(statbuf1.st_mode)) {
            continue; /* Пропуск всех объектов, которые не являются
обычными файлами */
        }

        DIR *d2 = opendir(dir2); /* Открытие второго каталога */
        if (!d2) { /* Проверка, открыт ли каталог */
            perror("Ошибка открытия Dir2"); /* Вывод ошибки, если не
удалось открыть каталог */
            closedir(d1); /* Закрытие первого каталога перед выходом
*/
            return 1; /* Возврат 1, указывая на ошибку */
        }

        struct dirent *entry2; /* Структура для хранения информации о
файле во втором каталоге */
        struct stat statbuf2; /* Структура для хранения информации о
файле из stat() */

        while ((entry2 = readdir(d2)) != NULL) { /* Чтение всех
файлов во втором каталоге */
            char file2_path[PATH_MAX]; /* Полный путь к файлу из
второго каталога */
            snprintf(file2_path, PATH_MAX, "%s/%s", dir2, entry2-
>d_name); /* Формирование пути */

            /* Используем stat для проверки, является ли текущий
объект обычным файлом */
            if (stat(file2_path, &statbuf2) == -1 ||
!S_ISREG(statbuf2.st_mode)) {

```

```

        continue; /* Пропуск всех объектов, которые не
являются обычными файлами */
    }

    /* Ожидание, пока число активных процессов не снизится до
допустимого значения */
    while (active_processes >= max_processes) {
        wait(NULL); /* Ожидание завершения любого дочернего
процесса */
        active_processes--; /* Уменьшение счетчика активных
процессов */
    }

    pid_t pid = fork(); /* Создание нового процесса для
сравнения файлов */
    if (pid == 0) { /* Ветвление: выполнение дочернего
процесса */
        int result = compare_files(file1_path, file2_path);
        /* Сравнение файлов */
        if (result > 0) {
            printf("PID: %d, Файл1: %s, Файл2: %s, Байтов: %d
- Совпадают\n",
                getpid(), file1_path, file2_path, result);
        /* Вывод результата совпадения */
        } else {
            printf("PID: %d, Файл1: %s, Файл2: %s - Не
совпадают\n",
                getpid(), file1_path, file2_path); /*
Вывод результата несовпадения */
        }
        exit(0); /* Завершение дочернего процесса */
    } else if (pid > 0) { /* Ветвление: выполнение
родительского процесса */
        active_processes++; /* Увеличение счетчика активных
процессов */
    } else {
        perror("Ошибка создания процесса"); /* Вывод ошибки
при неудаче fork() */
        closedir(d2); /* Закрытие второго каталога */
        closedir(d1); /* Закрытие первого каталога */
        return 1; /* Возврат 1, указывая на ошибку */
    }
}

closedir(d2); /* Закрытие второго каталога перед переходом к
следующему файлу из первого каталога */
}

/* Ожидание завершения всех оставшихся активных процессов */
while (active_processes > 0) {
    wait(NULL); /* Ожидание завершения любого дочернего процесса
*/

```



```

        active_processes--; /* Уменьшение счетчика активных процессов
*/
    }

    closedir(d1); /* Закрытие первого каталога после завершения
работы */
    return 0; /* Успешное завершение программы */
}

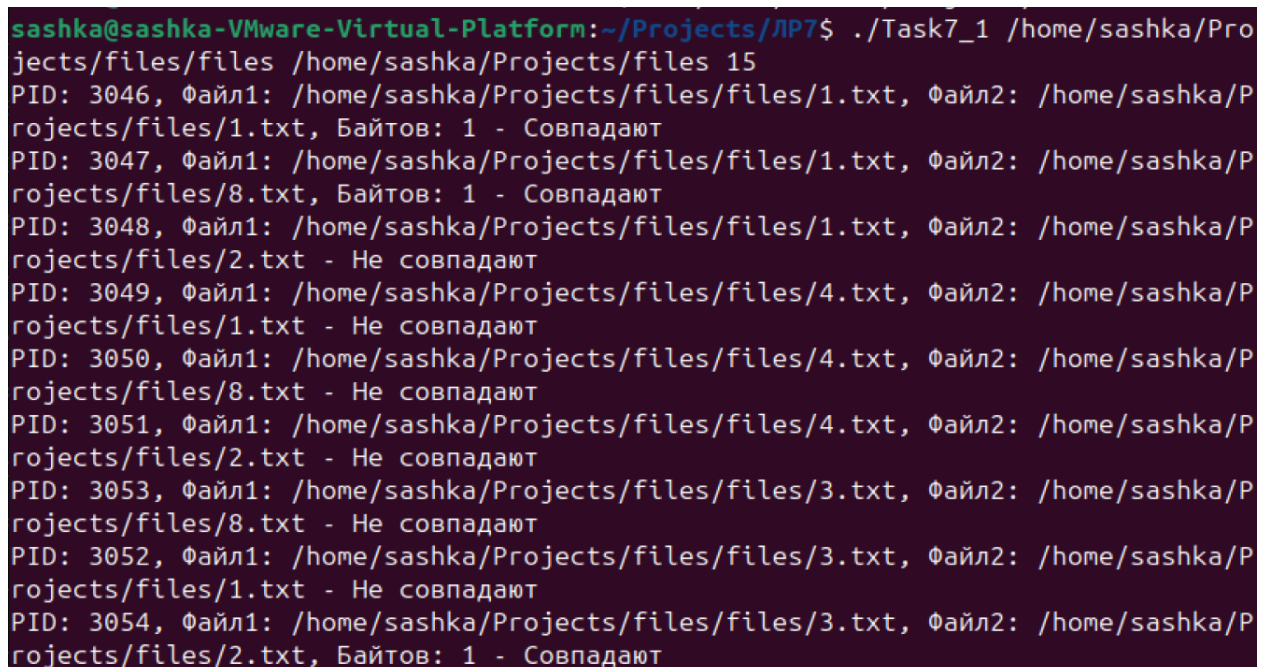
```

2.2 Запуск исполняемого файла

Для запуска программы необходимо перейти в директорию /home/sashka/Projects/ЛР7. После этого нужно запустить исполняемый файл с помощью команды `./Task7_1 /home/sashka/Projects/files /home/sashka/Projects/files/files N`, где первый аргумент – сам исполняемый файл, второй и третий – сравниваемые директории, четвёртый – максимальное число запущенных процессов в любой момент времени.

2.3 Результат работы программы

На рисунке 2.3.1 представлены результаты запуска исполняемого файла. Если посмотреть на содержимое заданных директорий, можно сделать вывод: программа выполняется корректно.



```

sashka@sashka-VMware-Virtual-Platform:~/Projects/ЛР7$ ./Task7_1 /home/sashka/Pro
jects/files/files /home/sashka/Projects/files 15
PID: 3046, Файл1: /home/sashka/Projects/files/files/1.txt, Файл2: /home/sashka/P
rojects/files/1.txt, Байтов: 1 - Совпадают
PID: 3047, Файл1: /home/sashka/Projects/files/files/1.txt, Файл2: /home/sashka/P
rojects/files/8.txt, Байтов: 1 - Совпадают
PID: 3048, Файл1: /home/sashka/Projects/files/files/1.txt, Файл2: /home/sashka/P
rojects/files/2.txt - Не совпадают
PID: 3049, Файл1: /home/sashka/Projects/files/files/4.txt, Файл2: /home/sashka/P
rojects/files/1.txt - Не совпадают
PID: 3050, Файл1: /home/sashka/Projects/files/files/4.txt, Файл2: /home/sashka/P
rojects/files/8.txt - Не совпадают
PID: 3051, Файл1: /home/sashka/Projects/files/files/4.txt, Файл2: /home/sashka/P
rojects/files/2.txt - Не совпадают
PID: 3053, Файл1: /home/sashka/Projects/files/files/3.txt, Файл2: /home/sashka/P
rojects/files/8.txt - Не совпадают
PID: 3052, Файл1: /home/sashka/Projects/files/files/3.txt, Файл2: /home/sashka/P
rojects/files/1.txt - Не совпадают
PID: 3054, Файл1: /home/sashka/Projects/files/files/3.txt, Файл2: /home/sashka/P
rojects/files/2.txt, Байтов: 1 - Совпадают

```

Рисунок 2.3.1 – Результат работы для ЛР 7.1

3 РЕАЛИЗАЦИЯ С ИСПОЛЬЗОВАНИЕМ ПОТОКОВ

3.1 Код программы

```
#include <stdio.h>    /* Подключение библиотеки для стандартного
ввода-вывода */
#include <stdlib.h>    /* Подключение библиотеки для работы с памятью и
преобразованиями */
#include <string.h>    /* Подключение библиотеки для работы со строками
*/
#include <unistd.h>    /* Подключение библиотеки для POSIX API, включая
функции работы с процессами */
#include <pthread.h>    /* Подключение библиотеки для работы с потоками
*/
#include <dirent.h>    /* Подключение библиотеки для работы с
каталогами */
#include <fcntl.h>    /* Подключение библиотеки для работы с файловыми
дескрипторами */
#include <sys/stat.h> /* Подключение библиотеки для работы с файловыми
атрибутами */
#include <sys/types.h> /* Подключение библиотеки для системных типов
данных */

#define BUFFER_SIZE 1024 /* Определение размера буфера для чтения
данных */
#define PATH_MAX 4096    /* Установка максимальной длины пути
вручную, если PATH_MAX не определен */

/* Структура данных для передачи путей к файлам в поток */
typedef struct {
    char file1_path[PATH_MAX]; /* Путь к первому файлу */
    char file2_path[PATH_MAX]; /* Путь ко второму файлу */
} FilePair;

/* Инициализация мьютекса для синхронизации доступа к общим данным */
pthread_mutex_t mutex;
/* Переменная для отслеживания текущего числа активных потоков */
int active_threads = 0;
/* Переменная для хранения максимального допустимого количества
активных потоков */
int max_threads;

/* Функция для побайтового сравнения содержимого двух файлов */
int compare_files(const char *file1, const char *file2) {
    int fd1 = open(file1, O_RDONLY); /* Открытие первого файла для
чтения */
    int fd2 = open(file2, O_RDONLY); /* Открытие второго файла для
чтения */
    if (fd1 < 0 || fd2 < 0) {        /* Проверка ошибок при открытии
файлов */
```

```

        perror("Ошибка открытия файлов"); /* Вывод ошибки */
        return -1;                          /* Возврат ошибки */
    }

    char buffer1[BUFFER_SIZE], buffer2[BUFFER_SIZE]; /* Буферы для
чтения данных из файлов */
    ssize_t bytes_read1, bytes_read2;                /* Переменные для
хранения числа прочитанных байтов */

    while (1) { /* Цикл для побайтового сравнения содержимого файлов
*/
        bytes_read1 = read(fd1, buffer1, BUFFER_SIZE); /* Чтение
данных из первого файла */
        bytes_read2 = read(fd2, buffer2, BUFFER_SIZE); /* Чтение
данных из второго файла */

        if (bytes_read1 != bytes_read2) { /* Проверка на равенство
числа прочитанных байтов */
            close(fd1); /* Закрытие первого файла */
            close(fd2); /* Закрытие второго файла */
            return 0;    /* Возвращаем 0, если файлы отличаются */
        }

        if (bytes_read1 == 0) { /* Если достигнут конец файлов */
            break; /* Завершаем цикл */
        }

        if (memcmp(buffer1, buffer2, bytes_read1) != 0) { /* Сравнение
данных в буферах */
            close(fd1); /* Закрытие первого файла */
            close(fd2); /* Закрытие второго файла */
            return 0;    /* Возвращаем 0, если данные отличаются */
        }
    }

    close(fd1); /* Закрытие первого файла */
    close(fd2); /* Закрытие второго файла */
    return 1;    /* Возвращаем 1, если файлы совпадают */
}

/* Функция, выполняемая в каждом потоке для сравнения файлов */
void *compare_files_thread(void *arg) {
    FilePair *files = (FilePair *)arg; /* Преобразование аргумента в
указатель на структуру FilePair */

    int result = compare_files(files->file1_path, files->file2_path);
/* Сравнение файлов */
    if (result > 0) { /* Если файлы совпадают */
        printf("Thread ID: %lu, Файл1: %s, Файл2: %s - Совпадают\n",
pthread_self(), files->file1_path, files->file2_path); /* Вывод
совпадения */
    }
}

```

```

    } else { /* Если файлы не совпадают */
        printf("Thread ID: %lu, Файл1: %s, Файл2: %s - Не
совпадают\n", pthread_self(), files->file1_path, files->file2_path);
/* Вывод несовпадения */
    }

    free(files); /* Освобождение памяти, выделенной для путей файлов
*/

    pthread_mutex_lock(&mutex); /* Захват мьютекса для синхронизации
*/
    active_threads--; /* Уменьшение числа активных потоков */
    pthread_mutex_unlock(&mutex); /* Освобождение мьютекса */

    pthread_exit(NULL); /* Завершение потока */
}

int main(int argc, char *argv[]) {
    if (argc != 4) { /* Проверка правильности количества аргументов
командной строки */
        fprintf(stderr, "Использование: %s <Dir1> <Dir2> <N>\n",
argv[0]); /* Сообщение об ошибке */
        return 1; /* Завершение программы с ошибкой */
    }

    const char *dir1 = argv[1]; /* Первый каталог */
    const char *dir2 = argv[2]; /* Второй каталог */
    max_threads = atoi(argv[3]); /* Преобразование третьего аргумента
в целое число для max_threads */

    DIR *d1 = opendir(dir1); /* Открытие первого каталога */
    if (!d1) { /* Проверка ошибок при открытии каталога */
        perror("Ошибка открытия Dir1"); /* Сообщение об ошибке */
        return 1; /* Завершение программы с ошибкой */
    }

    struct dirent *entry1; /* Переменная для чтения содержимого
каталога */
    struct stat statbuf1; /* Структура для хранения информации о файле
*/

    pthread_mutex_init(&mutex, NULL); /* Инициализация мьютекса */

    while ((entry1 = readdir(d1)) != NULL) { /* Чтение файлов из
первого каталога */
        char file1_path[PATH_MAX]; /* Массив для пути к файлу из
первого каталога */
        snprintf(file1_path, PATH_MAX, "%s/%s", dir1, entry1->d_name);
/* Формирование полного пути к файлу */

```

```

        if (stat(file1_path, &statbuf1) == -1 ||
!S_ISREG(statbuf1.st_mode)) { /* Пропуск, если файл не является
регулярным */
            continue;
        }

        DIR *d2 = opendir(dir2); /* Открытие второго каталога */
        if (!d2) { /* Проверка ошибок при открытии каталога */
            perror("Ошибка открытия Dir2"); /* Сообщение об ошибке */
            closedir(d1); /* Закрытие первого каталога */
            return 1; /* Завершение программы с ошибкой */
        }

        struct dirent *entry2; /* Переменная для чтения файлов из
второго каталога */
        struct stat statbuf2; /* Структура для хранения информации о
файле */

        while ((entry2 = readdir(d2)) != NULL) { /* Чтение файлов из
второго каталога */
            char file2_path[PATH_MAX]; /* Массив для пути к файлу из
второго каталога */
            snprintf(file2_path, PATH_MAX, "%s/%s", dir2, entry2-
>d_name); /* Формирование полного пути к файлу */

            if (stat(file2_path, &statbuf2) == -1 ||
!S_ISREG(statbuf2.st_mode)) { /* Пропуск, если файл не является
регулярным */
                continue;
            }

            pthread_mutex_lock(&mutex); /* Захват мьютекса для
синхронизации */
            while (active_threads >= max_threads) { /* Проверка, что
количество активных потоков не превышает max_threads */
                pthread_mutex_unlock(&mutex); /* Освобождение мьютекса
*/

                usleep(100); /* Ожидание перед повторной проверкой */
                pthread_mutex_lock(&mutex); /* Захват мьютекса */
            }
            active_threads++; /* Увеличение счетчика активных потоков
*/

            pthread_mutex_unlock(&mutex); /* Освобождение мьютекса */

            FilePair *files = malloc(sizeof(FilePair)); /* Выделение
памяти под пути файлов */
            strncpy(files->file1_path, file1_path, PATH_MAX); /*
Копирование пути первого файла */
            strncpy(files->file2_path, file2_path, PATH_MAX); /*
Копирование пути второго файла */

```

```

        pthread_t thread; /* Переменная для хранения
идентификатора потока */
        if (pthread_create(&thread, NULL, compare_files_thread,
files) != 0) { /* Создание нового потока для сравнения файлов */
            perror("Ошибка создания потока"); /* Сообщение об
ошибке */

            free(files); /* Освобождение памяти */
            closedir(d2); /* Закрытие второго каталога */
            closedir(d1); /* Закрытие первого каталога */
            return 1; /* Завершение программы с ошибкой */
        }

        pthread_detach(thread); /* Освобождение ресурсов потока
после завершения */
    }

    closedir(d2); /* Закрытие второго каталога */
}

/* Ожидание завершения всех потоков */
while (active_threads > 0) { /* Пока есть активные потоки */
    usleep(100); /* Ожидание перед повторной проверкой */
}

closedir(d1); /* Закрытие первого каталога */
pthread_mutex_destroy(&mutex); /* Уничтожение мьютекса */

return 0; /* Завершение программы */
}

```

3.2 Запуск исполняемого файла

Запуск такой же, как и для ЛР 7.1, только вместо исполняемого файла Task7_1, необходимо запустить Task7_2.

3.3 Результат работы программы

```

sashka@sashka-VMware-Virtual-Platform:~/Projects/RP7$ ./Task7_2 /home/sashka/Pro
jects/files/files /home/sashka/Projects/files 15
Thread ID: 140716657346240, Файл1: /home/sashka/Projects/files/files/4.txt, Файл
2: /home/sashka/Projects/files/8.txt - Не совпадают
Thread ID: 140716678317760, Файл1: /home/sashka/Projects/files/files/1.txt, Файл
2: /home/sashka/Projects/files/2.txt - Не совпадают
Thread ID: 140716688803520, Файл1: /home/sashka/Projects/files/files/1.txt, Файл
2: /home/sashka/Projects/files/8.txt - Совпадают
Thread ID: 140716699289280, Файл1: /home/sashka/Projects/files/files/1.txt, Файл
2: /home/sashka/Projects/files/1.txt - Совпадают
Thread ID: 140716537808576, Файл1: /home/sashka/Projects/files/files/3.txt, Файл
2: /home/sashka/Projects/files/2.txt - Совпадают
Thread ID: 140716667832000, Файл1: /home/sashka/Projects/files/files/4.txt, Файл
2: /home/sashka/Projects/files/1.txt - Не совпадают
Thread ID: 140716548294336, Файл1: /home/sashka/Projects/files/files/3.txt, Файл
2: /home/sashka/Projects/files/8.txt - Не совпадают
Thread ID: 140716636374720, Файл1: /home/sashka/Projects/files/files/3.txt, Файл
2: /home/sashka/Projects/files/1.txt - Не совпадают
Thread ID: 140716646860480, Файл1: /home/sashka/Projects/files/files/4.txt, Файл
2: /home/sashka/Projects/files/2.txt - Не совпадают

```

На рисунке слева представ-
лен результат работы прог-
раммы для ЛР 7.2. Как и в
случае с ЛР 7.1, он соответ-
ствует действительности,
так как вводимые директо-
рии в обоих случаях одина-
ковые.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была исследована работа с процессами и потоками в операционной системе Linux, что позволило глубже понять основные принципы создания и управления ими. В первой части работы, посвященной процессам, был подробно изучен механизм их создания с помощью системного вызова `fork()`. Мы проанализировали, как родительский процесс порождает дочерний, и как они взаимодействуют между собой. Важным аспектом стало изучение работы с функциями `wait()` и `waitpid()`, которые позволяют родительскому процессу отслеживать завершение дочернего и корректно освобождать ресурсы.

Кроме того, была исследована возможность замены текущей программы с помощью функций семейства `exec`. Эти функции позволяют процессу загрузить другой исполняемый файл, что открывает большие возможности для гибкости в управлении процессами. Они предоставляют различные способы передачи аргументов и среды выполнения, что делает их удобными для различных сценариев работы с внешними программами.

Во второй части лабораторной работы, посвященной потокам, было продемонстрировано, как создаются и управляются нити выполнения с использованием библиотеки `pthread`. Создание потоков позволяет значительно повысить производительность программ, эффективно используя многозадачность и многопоточность. Важно, что потоки разделяют память и ресурсы процесса, что значительно упрощает обмен данными между ними, но также требует внимательного управления синхронизацией для предотвращения ошибок.

Также были рассмотрены механизмы синхронизации потоков, такие как использование функции `pthread_join()`, позволяющей ожидать завершения работы потока, а также особенности работы с системным временем в Linux, что важно для задач, связанных с таймингом и производительностью.

Таким образом, лабораторная работа дала хорошее понимание ключевых механизмов операционной системы Linux, используемых для создания и управления как процессами, так и потоками. Это знание является основой для разработки эффективных многозадачных и многопоточных приложений, а также для оптимизации работы программ с различными процессами и ресурсами системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Леванцевич, В. А. Процессы в ОС Linux / В. А. Леванцевич // Лабораторные работы. – 2024. – №7.1.

[2] Леванцевич, В. А. Потоки в ОС Linux / В. А. Леванцевич // Лабораторные работы. – 2024. – №7.2.