

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 №4
на тему

**УПРАВЛЕНИЕ ПРОЦЕССАМИ И
ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ**

Студент
Преподаватель

Д. С. Кончик
Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 Цель работы	3
2 Теоретические сведения	4
3 Результат выполнения	5
Заключение	5
Список использованных источников	7
Приложение А (обязательное) Листинг кода	8

1 ЦЕЛЬ РАБОТЫ

Изучение основных особенностей подсистемы управления процессами и средств взаимодействия процессов в *Unix*. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов.

Написать программу (программы), соответствующую схеме «агент-менеджер». Процесс-«менеджер» получает (или генерирует) задание, порождает процессы-«агенты» и интерфейсы для взаимодействия с ними, декомпозирует задание на фрагменты (подзадания) и раздает их «агентам», принимает от «агентов» частичные результаты и собирает из них итоговый, ведет учет подзаданий и «агентов». Процессы-«агенты» (копии процесса-«менеджера, выполняющиеся по другой ветви алгоритма, или отдельные исполняемые файлы) принимают от «менеджера» фрагменты заданий, выполняют свои подзадания, возвращают «менеджеру» частичные результаты.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Процесс в *Linux* (как и в *UNIX*) – это программа, которая выполняется в отдельном виртуальном адресном пространстве. Когда пользователь регистрируется в системе, автоматически создается процесс, в котором выполняется оболочка (*shell*), например, */bin/bash*.

В *Linux* поддерживается классическая схема мультипрограммирования. *Linux* поддерживает параллельное (или квазипараллельное при наличии только одного процессора) выполнение процессов пользователя. Каждый процесс выполняется в собственном виртуальном адресном пространстве, т.е. процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на всю систему в целом. Один процесс не может прочесть что-либо из памяти (или записать в нее) другого процесса без "разрешения" на то другого процесса. Санкционированные взаимодействия между процессами допускаются системой.

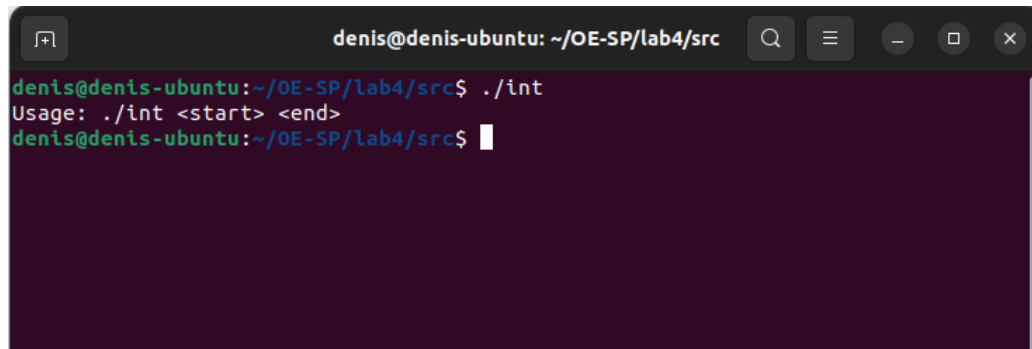
Ядро предоставляет системные вызовы для создания новых процессов и для управления порожденными процессами. Любая программа может начать выполняться только если другой процесс ее запустит.

Для создания процессов используются два системных вызова: *fork* и *exec*. *Fork* создает новое адресное пространство, которое полностью идентично адресному пространству основного процесса. Функция *fork* возвращает 0 в порожденном процессе и *PID* (*Process ID* – идентификатор порожденного процесса) – в основном. *PID* – это целое число. Теперь, когда процесс уже создан, можно запустить программу с помощью вызова *exec*. В адресное пространство порожденного с помощью *fork* процесса будет загружена новая программа и ее выполнение начнется с точки входа (адрес функции *main*) [1].

Каналы – неименованные (*pipe*) и именованные (*fifo*) – это средство передачи данных между процессами. Можно представить себе канал как небольшой кольцевой буфер в ядре операционной системы. С точки зрения процессов, канал выглядит как пара открытых файловых дескрипторов – один на чтение и один на запись (можно больше, но неудобно). Мы можем писать в канал до тех пор, пока есть место в буфере, если место в буфере кончится – процесс будет заблокирован на записи. Можем читать из канала пока есть данные в буфере, если данных нет – процесс будет заблокирован на чтении. Если закрыть дескриптор отвечающий за запись, то попытка чтения покажет конец файла. Если закрыть дескриптор отвечающий за чтение, то попытка записи приведет к доставке сигнала *SIGPIPE* и ошибке *EPIPE* [2].

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ

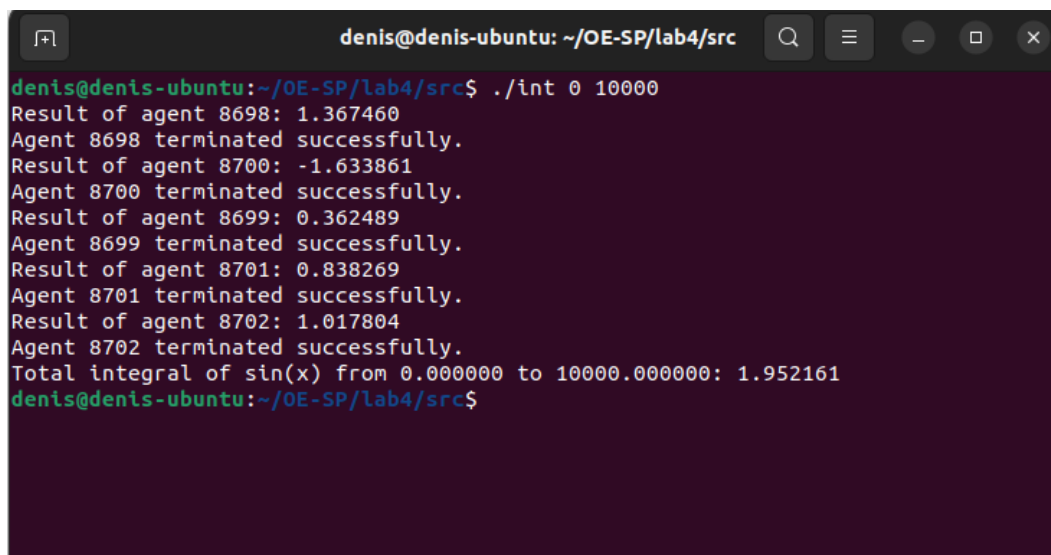
В результате лабораторной работы была создана программа, соответствующая схеме «агент-менеджер». Задача – вычисление интеграла на определенном промежутке (промежуток задается через параметры командной строки). При запуске программы без передачи аргументов появляется информация о возможных параметрах командной строки (рисунок 1).



```
denis@denis-ubuntu: ~/OE-SP/lab4/src
denis@denis-ubuntu:~/OE-SP/lab4/src$ ./int
Usage: ./int <start> <end>
denis@denis-ubuntu:~/OE-SP/lab4/src$
```

Рисунок 1 – Вывод инструкции

Программа через параметры командной строки принимает промежуток, на котором нужно вычислить интеграл $\int_a^b \sin(x) dx$, разделяет задачу между пятью процессами-«агентами», ждет их завершения и складывает получившиеся результаты на соответствующих промежутках. Также отдельно происходит вывод результата каждого «агента».



```
denis@denis-ubuntu: ~/OE-SP/lab4/src
denis@denis-ubuntu:~/OE-SP/lab4/src$ ./int 0 10000
Result of agent 8698: 1.367460
Agent 8698 terminated successfully.
Result of agent 8700: -1.633861
Agent 8700 terminated successfully.
Result of agent 8699: 0.362489
Agent 8699 terminated successfully.
Result of agent 8701: 0.838269
Agent 8701 terminated successfully.
Result of agent 8702: 1.017804
Agent 8702 terminated successfully.
Total integral of sin(x) from 0.000000 to 10000.000000: 1.952161
denis@denis-ubuntu:~/OE-SP/lab4/src$
```

Рисунок 2 – Результат работы программы

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были изучены основные особенности подсистемы управления процессами и средств взаимодействия процессов в *Unix*. Практическое проектирование, реализация и отладка программных комплексов, состоящих из нескольких взаимодействующих процессов, были рассмотрены на примере программы, реализующей схему «агент-менеджер».

Целью программы было вычисление определенного интеграла на заданном промежутке. Параметры командной строки используются для задания этого промежутка. После запуска программы без передачи аргументов выводится информация о возможных параметрах командной строки.

С помощью механизма взаимодействия процессов в *Unix* программа разделяет задачу на пять процессов-«агентов», каждый из которых выполняет вычисления на своем подпромежутке. «Менеджер» процесса ожидает завершения всех «агентов», затем собирает полученные частичные результаты и складывает их, чтобы получить окончательный результат интегрирования на заданном промежутке. Кроме того, результат каждого «агента» выводится отдельно.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Управление процессами в Linux [Электронный ресурс]. – Режим доступа: https://www.opennet.ru/docs/RUS/lnx_process/process2.html.

[2] Каналы (pipe, fifo) [Электронный ресурс]. – Режим доступа: <https://parallel.uran.ru/node/464>.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

Листинг 1 – Файл *main.c*

```
#include <sys/wait.h>
#include <unistd.h> // для pipe, fork, write, close, getpid
#include <stdio.h> // для printf
#include <stdlib.h> // для exit
#include "integral.h"
#define NUM_AGENTS 5

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <start> <end>\n", argv[0]);
        exit(1);
    }
    double total_result = 0.0;
    double start = atof(argv[1]);
    double end = atof(argv[2]);
    // Массив для хранения файловых дескрипторов каналов
    int pipefd[NUM_AGENTS][2];
    // Создание каналов для каждого агента
    for (int i = 0; i < NUM_AGENTS; i++) {
        if (pipe(pipefd[i]) == -1) {
            perror("pipe");
            exit(1);
        }
    }
    double step = (end - start) / NUM_AGENTS;
    for (int i = 0; i < NUM_AGENTS; i++) {
        pid_t pid = fork();
        if (pid == 0) {
            // Дочерний процесс (агент)
            double agent_start = start + i * step;
            double agent_end = start + (i + 1) * step;
            double result = compute_integral(agent_start, agent_end);
            // Записываем результат вычислений в канал
            if (write(pipefd[i][1], &result, sizeof(result)) == -1) {
                perror("write");
                exit(1);
            }
            // Закрываем записывающий конец канала
            close(pipefd[i][1]);
            printf("Result of agent %d: %f\n", getpid(), result);
            exit(0);
        } else if (pid < 0) {
            perror("fork");
            exit(1);
        }
    }
    // Ожидание завершения всех агентов
    for (int i = 0; i < NUM_AGENTS; i++) {
        int status;
        pid_t pid = waitpid(-1, &status, 0);
        if (pid == -1) {
            perror("waitpid");
            exit(1);
        } else {
            total_result += result;
        }
    }
}
```



```

        if (WIFEXITED(status)) {
            printf("Agent %d terminated successfully.\n", pid);
        } else {
            printf("Agent %d terminated abnormally.\n", pid);
            exit(1);
        }
    }
}

// Родительский процесс (менеджер)
for (int i = 0; i < NUM_AGENTS; i++) {

    double agent_result;

    // Читаем результат вычислений из канала
    if (read(pipefd[i][0], &agent_result, sizeof(agent_result)) == -1) {
        perror("read");
        exit(1);
    }

    total_result += agent_result;

    close(pipefd[i][0]); // Закрываем читающий конец канала
}

printf("Total integral of sin(x) from %f to %f: %f\n", start, end,
total_result);

return 0;
}

```

Листинг 2 – Файл *integral.c*

```

#include <math.h>

#define DX 0.00001

double compute_integral(double start, double end) {
    double result = 0.0;

    for (double x = start; x < end; x += DX) {
        result += sin(x) * DX;
    }

    return result;
}

```

Листинг 3 – Файл *makefile*

```

CC = gcc
TARGET = int

$(TARGET): main.o integral.o
    $(CC) $^ -o $@ -lm

main.o: main.c
    $(CC) -c $<

integral.o: integral.c
    $(CC) -c $<

clean:
    rm -f *.o

```