Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

Студент: Шевлякова София Сергеевна
Группа: М8О–208Б–21
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Постановка задачи

Задание

Необходимо написать 3-и программы. Далее будем обозначать эти программы A, B, C. Программа A принимает из стандартного потока ввода строки, а далее их отправляет программе C. Отправка строк должна производится построчно. Программа C печатает в стандартный вывод, полученную строку от программы A. После получения программа C отправляет программе A сообщение о том, что строка получена. До тех пор пока программа A не примет «сообщение о получение строки» от программы C, она не может отправлять следующую строку программе C. Программа B пишет в стандартный вывод количество отправленных символов программой A и количество принятых символов программой C. Данную информацию программа B получает от программ A и C соответственно. Способ организация межпроцессорного взаимодействия выбирает студент.

Общие сведения о программе

Программа состоит из трех файлов: a.c, b.c, c.c, используются заголовочные файлы: stdio.h, stdlib.h, unistd.h, sys/wait.h, fcntl.h, string.h. В ходе работы были применены следующие системные вызовы:

1. **write () -** переписывает count байт из буффера в файл. Возвращает количество записанных байт или -1;
2. **read () -** считывает count байт из файла в буффер. Возвращает количество считаных байт (оно может быть меньше count) или -1;
3. **pipe () -** создаёт канал между двумя процессами. Создаёт и помещает в массив 2 файловых дескриптора для чтения и для записи. Возвращает 0 или -1;
4. **close() -** закрывает файловый дескриптор, который больше не ссылается ни на один файл, возвращает 0 или -1;

5. **fork () -** порождается процесс-потомок. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родители возвращает идентификатор pid потомка или -1. Процесс-потомок возвращает 0 или -1;
6. **execv () -** заменяет текущий образ процесса новым образом процесса. Новая программа наследует от вызывавшего процесса идентификатор и открытые файловые дескрипторы;

Общий метод и алгоритм решения

С помощью двух вызовов fork(), создаются два дочерних процесса B и C. A получает от пользователя строку, длинна которой ограничивается 100 символами, и передает ее в C через pipeA_to_C. C получает строку и выводит ее в стандартный поток вывода, после этого отправляет программе A через канал длину строки (т.к. по условию, пока программа A не примет «сообщение о получение строки» от программы C, она не может отправлять следующую строку программе C). A и C отправляют через каналы в программу B размеры строк и B выводит их. Программа прекращает свою работу в случае введения пустой строки.

Основные файлы программы

========== a.c ==========

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <sys/wait.h>

#define READ 0
#define WRITE 1

void send_error_and_stop(char *massage, int code) {
    int i = strlen(massage);
    write(STDERR_FILENO, massage, sizeof(char) * i);
```

```
    exit(code);
}
int main() {
    int file_descriptorsA_to_C[2];
    int file_descriptorsA_to_B[2];
    int file_descriptorsC_to_B[2];
    int file_descriptorsC_to_A[2];

    int pipeA_to_C = pipe(file_descriptorsA_to_C);
    int pipeA_to_B = pipe(file_descriptorsA_to_B);
    int pipeC_to_B = pipe(file_descriptorsC_to_B);
    int pipeC_to_A = pipe(file_descriptorsC_to_A);
    if (pipeA_to_C == -1 || pipeA_to_B == -1 || pipeC_to_B == -1 || pipeC_to_A == -1) {
        send_error_and_stop("Pipe error\n", 1);
    }

    switch (fork()) {
    case -1:
        send_error_and_stop("Fork error\n", 2);
    case 0:
        close(file_descriptorsA_to_C[WRITE]);
        close(file_descriptorsA_to_C[READ]);
        close(file_descriptorsC_to_B[WRITE]);
        close(file_descriptorsA_to_B[WRITE]);
        close(file_descriptorsC_to_A[WRITE]);
        close(file_descriptorsC_to_A[READ]);

        char file_descriptor_read_A[10];
        char file_descriptor_read_C[10];
        sprintf(file_descriptor_read_A, "%d", file_descriptorsA_to_B[READ]);
        sprintf(file_descriptor_read_C, "%d", file_descriptorsC_to_B[READ]);

        char *B_argv[] = {"b", file_descriptor_read_A, file_descriptor_read_C, NULL};

        if (execv("b", B_argv) == -1) {
            send_error_and_stop("Exec B error\n", 3);
        }
        break;
    default:
        switch (fork()) {
        case -1:
            send_error_and_stop("Fork error\n", 2);
        case 0:
            close(file_descriptorsA_to_C[WRITE]);
            close(file_descriptorsA_to_B[WRITE]);
            close(file_descriptorsA_to_B[READ]);
            close(file_descriptorsC_to_B[READ]);
            close(file_descriptorsC_to_A[READ]);
```

```
            char file_descriptor_read_A[10];
            char file_descriptor_write_A[10];
            char file_descriptor_write_B[10];
            sprintf(file_descriptor_read_A, "%d", file_descriptorsA_to_C[READ]);
            sprintf(file_descriptor_write_A, "%d", file_descriptorsC_to_A[WRITE]);
            sprintf(file_descriptor_write_B, "%d", file_descriptorsC_to_B[WRITE]);

            char *C_argv[] = {"c", file_descriptor_read_A, file_descriptor_write_A, file_descriptor_write_B,
NULL};

            if (execv("c", C_argv) == -1) {
                send_error_and_stop("Exec C error\n", 4);
            }
            break;
        default:
            close(file_descriptorsA_to_C[READ]);
            close(file_descriptorsA_to_B[READ]);
            close(file_descriptorsC_to_B[WRITE]);
            close(file_descriptorsC_to_B[READ]);
            close(file_descriptorsC_to_A[WRITE]);

            char string[100];
            while (1) {
                if(gets(string) == NULL) {
                    send_error_and_stop("Gets error\n", 5);
                }
                int length = strlen(string);

                if(length == 0) {
                    if(write(file_descriptorsA_to_C[WRITE], &length, sizeof(length)) == -1) {
                        send_error_and_stop("Write in fdA_to_C error\n", 6);
                    }
                    if(write(file_descriptorsA_to_B[WRITE], &length, sizeof(length)) == -1) {
                        send_error_and_stop("Write in fdA_to_B error\n", 7);
                    }
                    break;
                }
                if(write(file_descriptorsA_to_C[WRITE], &length, sizeof(length)) == -1) {
                    send_error_and_stop("Write in fdA_to_C error\n", 6);
                }
                if(write(file_descriptorsA_to_C[WRITE], &string, sizeof(char) * length) == -1) {
                    send_error_and_stop("Write in fdA_to_C error\n", 6);
                }
                if(write(file_descriptorsA_to_B[WRITE], &length, sizeof(length)) == -1) {
                    send_error_and_stop("Write in fdA_to_B error\n", 7);
                }
                int check = 0;
                if(read(file_descriptorsC_to_A[READ], &check, sizeof(check)) == -1) {
                    send_error_and_stop("Read from fdA_to_C error\n", 8);
```

```
                return 1;
            }
        }
        break;
    }
  }
  return 0;
}
========================= b.c ========================

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

#define READ_C 1
#define READ_A 0

void send_error_and_stop(char *massage, int code) {
  int i = strlen(massage);
  write(STDERR_FILENO, massage, sizeof(char) * i);
  exit(code);
}

int main(int argc, const char *argv[]) {
  int length;
  int file_descriptors[2];
  file_descriptors[READ_A] = atoi(argv[1]);
  file_descriptors[READ_C] = atoi(argv[2]);

  while (1){
    if (read(file_descriptors[READ_A], &length, sizeof(int)) == -1) {
      send_error_and_stop("Read from A error", 1);
    }
    if(length == 0) {
      break;
    }
    printf("from A = %d\n", length);
    if(read(file_descriptors[READ_C], &length, sizeof(int)) == -1) {
      send_error_and_stop("Read from C error", 2);
      return 1;
    }
    printf("from C = %d\n", length);
  }
  return 0;
}
========================= c.c ========================
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

#define WRITE_B 2
#define WRITE_A 1
#define READ_A 0

void send_error_and_stop(char *massage, int code) {
    int i = strlen(massage);
    write(STDERR_FILENO, massage, sizeof(char) * i);
    exit(code);
}

int main(int argc, const char *argv[]) {
    int file_descriptors[3];
    file_descriptors[READ_A] = atoi(argv[1]);
    file_descriptors[WRITE_A] = atoi(argv[2]);
    file_descriptors[WRITE_B] = atoi(argv[3]);

    int length;
    char string[100];
    memset(string, 0, 100);

    while (1) {
        if(read(file_descriptors[READ_A], &length, sizeof(int)) == -1) {
            send_error_and_stop("Read from A error", 1);
        }
        if(length == 0) {
            break;
        }
        if(read(file_descriptors[READ_A], &string, sizeof(char) * length) == -1) {
            send_error_and_stop("Read from A error", 1);
        }
        printf("C: %s\n", string);
        if(write(file_descriptors[WRITE_A], &length, sizeof(int)) == -1) {
            send_error_and_stop("Write to A error", 2);
        }
        if(write(file_descriptors[WRITE_B], &length, sizeof(int)) == -1) {
            send_error_and_stop("Write to B error", 3);
        }
        memset(string, 0, 100);
        length = 0;
    }
    return 0;
}
```

Пример работы

```
sonikxx@LAPTOP-9UGJH447:~/OS/cp$ ./a.out
bebra
C: bebra
from A = 5
from C = 5
12121``` sasas 1
C: 12121``` sasas 1
from A = 16
from C = 16
23sss{}{}{ 1 1 1
C: 23sss{}{}{ 1 1 1
from A = 17
from C = 17
```

Вывод

В ходе данного курсового проекта я попрактиковалась в управлении процессами при помощи системных вызовов и обеспечении обмена данными между процессами при помощи неименованных каналов.

Системные вызовы необходимы для управления процессами, файлами и каталогами, а также каналами ввода и вывода данных. Одним из способов создания дочернего процесса является системный вызов fork(), он создает точную копию исходного процесса, включая все дескрипторы файлов, регистры и т. п. После выполнения вызова fork() исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга. Благодаря систему вызову pipe можно создать канал (трубу) между двумя процессами, в которой один процесс сможем писать поток байтов, а другой процесс сможет его читать, так мы переопределяем потоки ввода-вывода.

Благодаря системным вызовам можно упростить программу или выполнить действия, запрещенные в пользовательском режиме.

При написании курсового проекта я закрепила свои знания и навыки, полученные во время прохождения курса операционных систем.