Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу
«Операционные системы»

Тема работы
“Межпроцессорное взаимодействие через memory-mapped files”

Студент: Борисов Ян Артурович
Группа: М8О-208Б-20
Вариант: 19
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/Yannikupy/OS

Постановка задачи

Задача: реализовать программу, в которой родительский процесс создает два дочерних процесса. Родительский процесс принимает строки, которые отправляются в тот или иной дочерний процесс в зависимости от следующего правила: с вероятностью 80% строки отправляются в pipe1, иначе в pipe2. Оба процесса удаляют гласные из строк. Межпроцессорное взаимодействие осуществляется посредством отображаемых файлов (memory-mapped files).

Общие сведения о программе

Программа содержится в файлах parent.c и child.c

Общий метод и алгоритм решения

При запуске программы пользователю предлагается ввести имя файла для первого и для второго дочернего процесса. В эти файлы будет записываться вывод соответствующих процессов.

После запуска программы выполняется отображение двух файлов, имена которых известны заранее. Так как операционная система не позволяет выполнить отображение пустого файла, то перед отображением в файлы записываются «пустые» строки. В качестве «пустой» строки используется строка, состоящая из одного системного символа.
Затем создаются два дочерних процесса. Родительский процесс считывает строки с консольного ввода при помощи функции get_string(). Данная функция считывает строку произвольной длины из стандартного ввода. Затем при помощи функции rand() определяется дочерний процесс, которому отправится эта строка на обработку.
Передача строки дочерним процессам осуществляется с помощью ее копирования в отображенный файл.
Дочерние процессы перенаправляют свой стандартный вывод с помощью dup2 в созданный файл. Затем они заменяют свой образ памяти и выполняют программу child, в которой они считывают строки и удаляют из нее все

гласные буквы.
В качестве сигнала используется «пустая» строка. Если дочерний процесс считал «пустую» строку, то ему не нужно ничего выполнять. Если же считана другая строка, то её необходимо обработать. После обработки в отображённый файл вновь записывается «пустая» строка.
Если пользователь нажал Ctrl+D, то родительский процесс посылает обоим дочерним процессам сигнал о завершении работы, закрывает все файлы и завершается сам. Отображаемые файлы, использованные для взаимодействия процессов, удаляются.

Исходный код

Parent.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define MAP_SIZE 4096

// files for mapping
char* file1_name = "file1_mapped";
char* file2_name = "file2_mapped";

// empty string as a signal
char empty = 1;
char* empty_string = &empty;

// scan a string with unknown length
char* get_string() {
    int len = 0, capacity = 10;
    char* s = (char*)malloc(10 * sizeof(char));
    if (s == NULL) {
        perror("Can't read a string");
        exit(6);
    }

    char c;
    while ((c = getchar()) != '\n') {
        s[len++] = c;
        if (c == EOF) {
            break;
        }
        if (len == capacity) {
            capacity *= 2;
            s = (char*)realloc(s, capacity * sizeof(char));
            if (s == NULL) {
                perror("Can't read a string");
```

```
                exit(6);
            }
        }
    };
    s[len] = '\0';
    return s;
}


int main() {
    srand(time(NULL));

    // creating files for output of child processes
    printf("Enter file's name for child process 1: ");
    char* output_file1_name = get_string();

    printf("Enter file's name for child process 2: ");
    char* output_file2_name = get_string();

    int output_file1 = open(output_file1_name, O_WRONLY | O_CREAT |
O_TRUNC, S_IWRITE | S_IREAD);
    int output_file2 = open(output_file2_name, O_WRONLY | O_CREAT |
O_TRUNC, S_IWRITE | S_IREAD);
    if (output_file1 < 0 || output_file2 < 0) {
        perror("Can't open file");
        exit(1);
    }


    // creating files for mapping
    int fd1 = open(file1_name, O_RDWR | O_CREAT, S_IWRITE | S_IREAD);
    int fd2 = open(file2_name, O_RDWR | O_CREAT, S_IWRITE | S_IREAD);
    if (fd1 < 0 || fd2 < 0) {
        perror("Can't open file");
        exit(1);
    }

    // empty files can't be mapped, so we'll put our empty_string
there
    if (write(fd1, empty_string, sizeof(empty_string)) < 0) {
        perror("Can't write to file");
        exit(1);
    }
    if (write(fd2, empty_string, sizeof(empty_string)) < 0) {
        perror("Can't write to file");
        exit(1);
    }

    // mapping files
    char* file1 = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd1, 0);
    char* file2 = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd2, 0);
    if (file1 == MAP_FAILED || file2 == MAP_FAILED) {
        perror("Can't map a file");
        exit(2);
    }
```

```
    // creating child processes
    pid_t pid1 = fork();
    if (pid1 < 0) {
        perror("Can't create child process");
        exit(3);
    }

    if (pid1 > 0) { // parent
        pid_t pid2 = fork();
        if (pid2 < 0) {
            perror("Can't create child process");
            exit(3);
        }

        if (pid2 > 0) { // parent

            while (1) {
                char* s = get_string();

                if (rand() % 100 + 1 <= 80) {
                    strcpy(file1, s);
                    if (s[0] == EOF) {
                        strcpy(file2, s);
                        break;
                    }
                }
                else {
                    strcpy(file2, s);
                    if (s[0] == EOF) {
                        strcpy(file1, s);
                        break;
                    }
                }
            }
            if (munmap(file1, MAP_SIZE) < 0 || munmap(file2, MAP_SIZE)
< 0) {
                perror("Can't unmap files");
                exit(4);
            }
            if (close(fd1) < 0 || close(fd2) < 0) {
                perror("Can't close files");
                exit(5);
            }
            if (remove(file1_name) < 0 || remove(file2_name) < 0) {
                perror("Can't delete files");
                exit(6);
            }
        }
        else { // child2
            // redirecting output
            if (dup2(output_file2, STDOUT_FILENO) < 0) {
                perror("Can't redirect stdout for child process");
                exit(7);
            }
```

```
            char* arr [] = {"2", NULL};
            execv("child", arr);

            // it won't go here if child executes
            perror("Can't execute child process");
            exit(8);
        }
    }
    else { // child1
        // redirecting output
        if (dup2(output_file1, STDOUT_FILENO) < 0) {
            perror("Can't redirect stdout for child process");
            exit(7);
        }

        char* arr [] = {"1", NULL};
        execv("child", arr);

        // it won't go here if child executes
        perror("Can't execute child process");
        exit(8);
    }
}
```

Child.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define MAP_SIZE 4096

// files for mapping
char* file1_name = "file1_mapped";
char* file2_name = "file2_mapped";

// empty string as a signal
char empty = 1;
char* empty_string = &empty;

void removeChar(char *str, char garbage) {

    char *src, *dst;
    for (src = dst = str; *src != '\0'; src++) {
        *dst = *src;
        if (*dst != garbage) dst++;
    }
    *dst = '\0';
}

void delete_vowels(char *str) {
    int length = strlen(str);
    char *front = str;
```

```
    char *back = str + length - 1;

    while (front <= back) {
        if((*front == 'a') || (*front == 'e') || (*front == 'i') ||
(*front == 'o') || (*front == 'u') || (*front == 'y') ||
           (*front == 'A') || (*front == 'E') || (*front == 'I') ||
(*front == 'O') || (*front == 'U') || (*front == 'Y')){
            removeChar(str, *front);
        }
        ++front;
    }
}

int main(int argc, char* argv[]) {
    char* file_name;
    if (argv[0][0] == '1') {
        file_name = file1_name;
    }
    else if (argv[0][0] == '2') {
        file_name = file2_name;
    }
    else {
        perror("Unknown file");
        exit(8);
    }
    // opening a file for mapping
    int fd = open(file_name, O_RDWR | O_CREAT, S_IWRITE | S_IREAD);
    if (fd < 0) {
        perror("Can't open file");
        exit(1);
    }

    // mapping file
    char* file = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
    if (file == MAP_FAILED) {
        perror("Can't map a file");
        exit(2);
    }

    while (1) {
        // waiting for a string
        while (strcmp(file, empty_string) == 0) {}

        // terminating if Ctrl+D was pressed
        if (file[0] == EOF) {
            if (munmap(file, MAP_SIZE) < 0) {
                perror("Can't unmap file");
                exit(4);
            }
            exit(0);
        }

        char* string = (char*)malloc(strlen(file) * sizeof(char));
        strcpy(string, file);
        delete_vowels(string);
        printf("%s\n", string);
```

```
            fflush(stdout);
            strcpy(file, empty_string);
            free(string);
        }
}
```

Демонстрация работы программы

./parent

Enter file's name for child process 1: file1

Enter file's name for child process 2: file2

dqpowkopekwp

ef[perklf[pelfer

eqpleqwldqw

erpkverpkrep

dwq[pekqwpkdpew

dflmdflpv

qe[qpeklqwp

dfpdpckerpo

xzpxksk

eqwelqwp

wpkcpekcer

fpbkpgf

File1

dqpwkpkwp

f[prklf[plfr

qplqwldqw

rpkvrpkrp

dwq[pkqwpkdpw

dflmdflpv

xzpxksk

qwlqwp

wpkcpkcr

fpbkpgf

File2

q[qpklqwp

dfpdpckrp

Выводы

В ходе выполнения данной работы мы расширили свои навыки работы с процессами и освоили технологию «файл маппинга», научились использовать ее правильно.