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

Лабораторная работа №2 по курсу

«Операционные системы»

Управление процессами в ОС. Обеспечение обмена данных между процессами посредством каналов.

Студент: Моисеенков Илья Павлович
Группа: М80 – 208Б-19
Вариант: 22
Преподаватель: Миронов Евгений Сергеевич
Дата: 19.10.2020
Оценка: отлично
Подпись: _____

Москва, 2020

1. Постановка задачи

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (pipe).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2.

Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод. Родительский и дочерний процесс должны быть представлены разными программами.

Правило фильтрации: с вероятностью 80% строки отправляются в pipe1, иначе в pipe2. Дочерние процессы инвертируют строки.

2. Общие сведения о программе

Программа написана на языке Си в UNIX-подобной операционной системе (Ubuntu). В программе создается два дочерних процесса child1 и child2. Каждый дочерний процесс связан с родительским при помощи отдельного канала pipe.

Передача строки в pipe в родительском процессе вынесена в отдельную функцию.

Программа принимает на вход неограниченное количество строк произвольной длины.

Один из двух дочерних процессов выполняет инверсию данной строки и выводит её на экран. Программа для дочерних процессов запускается при помощи функции execl.

Программа завершает свою работу при нажатии Ctrl+D.

Программа обрабатывает все возможные системные ошибки и выводит соответствующие сообщения в случае их возникновения.

3. 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. Если пользователь ввёл имя несуществующего файла, он будет создан.

После запуска программы создаются два канала fd1 и fd2, 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. Если сгенерированное случайное число по модулю 100 не больше 80, то строка будет передана первому дочернему процессу, в

противном случае – второму. Таким образом, вероятность попадания строки в первый дочерний процесс составляет 80%.

Передача строки в дочерний процесс реализована в виде процедуры to_pipe, принимающая в качестве аргумента канал связи и саму строку. Строки передаются посимвольно.

Дочерние процессы закрывают ненужные каналы связи и перенаправляют свой стандартный вывод в созданный файл, а стандартный ввод – через соответствующий pipe. Затем они заменяют свой образ памяти и выполняют программу child, в которой они считывают размер строки, саму строку и выполняют её инверсию.

Инверсия строки производится в процедуре reverse_string. Она принимает строку и выполняет ее реверс «на месте», используя технику «двух указателей». После обработки новая строка направляется в стандартный вывод.

Если пользователь нажал Ctrl+D, то родительский процесс посылает обоим дочерним процессам сигнал о завершении работы и завершается сам.

4. Основные файлы программы

parent.c

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

// writes size of str and str to pipe fd
void to_pipe(int* fd, char* str) {
    int i = 0;
    char c;
    do {
        c = str[i++];
        if (write(fd[1], &c, sizeof(char)) < 0) {
            perror("Can't write to the pipe");
            exit(4);
        }
    } while (c != '\0');
}

// 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 child processes
    printf("Enter file's name for chiild process 1: ");
    char* file1_name = get_string();

    printf("Enter file's name for chiild process 2: ");
    char* file2_name = get_string();

    int file1 = open(file1_name, O_WRONLY | O_CREAT, S_IWRITE | S_IREAD);
    int file2 = open(file2_name, O_WRONLY | O_CREAT, S_IWRITE | S_IREAD);
    if (file1 < 0 || file2 < 0) {
        perror("Can't open file");
        exit(1);
    }

    // creating pipes for child processes
    int fd1[2];
    int fd2[2];
    // fd[0] - read, fd[1] - write
    if (pipe(fd1) < 0 || pipe(fd2) < 0) {
        perror("Can't create pipe");
        exit(2);
    }

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

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

        if (pid2 > 0) { // parent
            // close useless file descriptors
            close(fd1[0]);
            close(fd2[0]);

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

                if (rand() % 100 + 1 <= 80) {
                    to_pipe(fd1, s);
                    if (s[0] == EOF) {
                        to_pipe(fd2, s);
                        break;
                    }
                }
```

```
                else {
                    to_pipe(fd2, s);
                    if (s[0] == EOF) {
                        to_pipe(fd1, s);
                        break;
                    }
                }
            }

            close(fd1[1]);
            close(fd2[1]);
        }
        else { // child2
            // close useless file descriptors
            close(fd1[0]);
            close(fd1[1]);
            close(fd2[1]);

            // redirecting standart input and output for child processes
            if (dup2(fd2[0], STDIN_FILENO) < 0) {
                perror("Can't redirect stdin for child process");
                exit(5);
            };
            if (dup2(file2, STDOUT_FILENO) < 0) {
                perror("Can't redirect stdout for child process");
                exit(5);
            }
            execl("child", NULL, NULL);

            // it won't go here if child executes
            perror("Can't execute child process");
            exit(6);
        }
    }
    else { // child1
        // close useless file descriptors
        close(fd1[1]);
        close(fd2[0]);
        close(fd2[1]);

        // redirecting standart input and output for child processes
        if (dup2(fd1[0], STDIN_FILENO) < 0) {
            perror("Can't redirect stdin for child process");
            exit(5);
        }
        if (dup2(file1, STDOUT_FILENO) < 0) {
            perror("Can't redirect stdout for child process");
            exit(5);
        }
        execl("child", NULL, NULL);

        // it won't go here if child executes
        perror("Can't execute child process");
        exit(6);
    }
}
```

child.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reverse_string(char *str) {
```

```
    int length = strlen(str);
    char *front = str;
    char *back = str + length - 1;

    while (front < back) {
        char tmp = *front;
        *front = *back;
        *back = tmp;
        ++front;
        --back;
    }
}

// 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 string1");
        exit(6);
    }

    char c;
    do {
        c = getchar();
        if (c == EOF) {
            close(0);
            exit(0);
        }
        s[len++] = c;
        if (len == capacity) {
            capacity *= 2;
            s = (char*)realloc(s, capacity * sizeof(char));
            if (s == NULL) {
                perror("Can't read a string2");
                exit(6);
            }
        }
    } while  (c != '\0');
    s[len] = 0;
    return s;
}

int main(int argc, char* argv[]) {
    while (1) {
        char* str = get_string();
        reverse_string(str);
        printf("%s\n", str);
        fflush(stdout);
    }
}
```

5. Демонстрация работы программы

```
mosik@LAPTOP-69S778GL:~/os_lab2$ gcc child.c -o child
mosik@LAPTOP-69S778GL:~/os_lab2$ gcc parent.c -o parent
mosik@LAPTOP-69S778GL:~/os_lab2$ ./parent
Enter file's name for chiild process 1: first_file
Enter file's name for chiild process 2: second_file
Hey
This is a test
for my program
It sends strings
to one of child
```

```
processes
where
they are reversed
and put into file
through file descriptor
let's check the result

mosik@LAPTOP-69S778GL:~/os_lab2$ cat first_file
yeH
tset a si sihT
margorp ym rof
sessecorp
erehw
desrever era yeht
elif otni tup dna
rotpircsed elif hguorht
tluser eht kcehc s'tel
mosik@LAPTOP-69S778GL:~/os_lab2$ cat second_file
sgnirts sdnes tI
dlihc fo eno ot

mosik@LAPTOP-69S778GL:~/os_lab2$ strace -f -e trace="%process,read,write,dup2,pipe" -o
strace_log.txt ./parent
Enter file's name for chiild process 1: first_file
Enter file's name for chiild process 2: second_file
10
20
30

mosik@LAPTOP-69S778GL:~/os_lab2$ cat strace_log.txt
942   execve("./parent", ["./parent"], 0x7fffe903e778 /* 29 vars */) = 0
942   read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000b\0\0\0\0\0\0"...,
832) = 832
942   read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"...,
832) = 832
942   arch_prctl(ARCH_SET_FS, 0x7f10a9780740) = 0
942   write(1, "Enter file's name for chiild pro"..., 40) = 40
942   read(0, "first_file\n", 1024)     = 11
942   write(1, "Enter file's name for chiild pro"..., 40) = 40
942   read(0, "second_file\n", 1024)    = 12
942   pipe([5, 6])                      = 0
942   pipe([7, 8])                      = 0
942   clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x7f10a9780a10) = 951
942   clone( <unfinished ...>
951   dup2(5, 0)                        = 0
951   dup2(3, 1)                        = 1
951   execve("child", [], 0x7fffdb666388 /* 29 vars */ <unfinished ...>
942   <... clone resumed> child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f10a9780a10) =
952
942   read(0,  <unfinished ...>
952   dup2(7, 0)                        = 0
952   dup2(4, 1)                        = 1
952   execve("child", [], 0x7fffdb666388 /* 29 vars */ <unfinished ...>
951   <... execve resumed> )            = 0
951   read(6, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"...,
832) = 832
952   <... execve resumed> )            = 0
951   arch_prctl(ARCH_SET_FS, 0x7f0a5d4814c0) = 0
951   read(0,  <unfinished ...>
952   read(5, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"...,
832) = 832
952   arch_prctl(ARCH_SET_FS, 0x7ff4eb5f14c0) = 0
952   read(0,  <unfinished ...>
942   <... read resumed> "10\n", 1024)  = 3
```

```
942   write(6, "1", 1)                  = 1
951   <... read resumed> "1", 4096)     = 1
942   write(6, "0", 1 <unfinished ...>
951   read(0,  <unfinished ...>
942   <... write resumed> )             = 1
951   <... read resumed> "0", 4096)     = 1
942   write(6, "\0", 1 <unfinished ...>
951   read(0,  <unfinished ...>
942   <... write resumed> )             = 1
951   <... read resumed> "\0", 4096)    = 1
942   read(0,  <unfinished ...>
951   write(1, "01\n", 3)               = 3
951   read(0,  <unfinished ...>
942   <... read resumed> "20\n", 1024)  = 3
942   write(8, "2", 1)                  = 1
952   <... read resumed> "2", 4096)     = 1
942   write(8, "0", 1 <unfinished ...>
952   read(0,  <unfinished ...>
942   <... write resumed> )             = 1
952   <... read resumed> "0", 4096)     = 1
942   write(8, "\0", 1 <unfinished ...>
952   read(0,  <unfinished ...>
942   <... write resumed> )             = 1
952   <... read resumed> "\0", 4096)    = 1
942   read(0,  <unfinished ...>
952   write(1, "02\n", 3)               = 3
952   read(0,  <unfinished ...>
942   <... read resumed> "30\n", 1024)  = 3
942   write(6, "3", 1)                  = 1
951   <... read resumed> "3", 4096)     = 1
942   write(6, "0", 1)                  = 1
951   read(0,  <unfinished ...>
942   write(6, "\0", 1 <unfinished ...>
951   <... read resumed> "0", 4096)     = 1
942   <... write resumed> )             = 1
951   read(0,  <unfinished ...>
942   read(0,  <unfinished ...>
951   <... read resumed> "\0", 4096)    = 1
951   write(1, "03\n", 3)               = 3
951   read(0,  <unfinished ...>
942   <... read resumed> "", 1024)      = 0
942   exit_group(0)                     = ?
952   <... read resumed> "", 4096)      = 0
951   <... read resumed> "", 4096)      = 0
942   +++ exited with 0 +++
952   exit_group(0)                     = ?
951   exit_group(0)                     = ?
952   +++ exited with 0 +++
951   +++ exited with 0 +++
```

6. Выводы

Управление процессами – одна из ключевых задач операционной системы. Обычно ОС сама создаёт необходимые для себя и для других программ процессы, но возникают ситуации, когда пользователю требуется вмешаться в работу системы.

Язык Си при подключении библиотеки unistd.h (для Unix-подобных ОС) обладает возможностью совершать системные вызовы, связанные с вводом/выводом данных, управлением файлами и каталогами и, что самое важное, управлением процессами.

Внутри программы на языке Си можно создать дополнительный, т.н. дочерний процесс, который продолжит выполнение текущей программы параллельно с родительским

процессом. Для этого используется функция fork, совершающая соответствующий системный вызов. Удобство в том, что при помощи ветвлений в коде программы можно отделить код родительского процесса от кода, предназначенного для ребёнка. А можно заставить ребёнка запустить другую программу. Для этого предназначено семейство функций exec*. Обеспечить связь между процессами можно при помощи канала pipe, запрос на создание которого можно также совершить в языке Си.

Однако не только язык Си способен совершать системные вызовы, связанные с управлением процессами. Похожие библиотеки есть на многих других языках программирования, ведь современное программное обеспечение крайне редко состоит из одного процесса.