Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Операционные системы»

Межпроцессорное взаимодействие

Студент: О. В. Бабин
Преподаватель: А. А. Соколов
Группа: М8О-206Б-19
Дата: 25.12.2020
Оценка:
Подпись:

Москва, 2020

1 Постановка задачи

Цель работы:

Приобретение практических навыков в:

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

Задание (вариант 18):

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 и дочерний процесс должны быть представлены разными программами.

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.

Правило фильтрации: нечетные строки отправляются в pipe1, четные в pipe2. Дочерние процессы удаляют все гласные из строк.

2 Общие сведения о программе

Программа компилируется из файла main.c. Подключены заголовочные файлы: unistd.h, fcntl.h, stdlib.h. В программе используются следующие системные вызовы:

1. **pipe** — принимает массив из двух целых чисел, в случае успеха массив будет содержать два файловых дескриптора, которые будут использоваться для конвейера, первое число в массиве предназначено для чтения, второе для записи, а так же вернется 0. В случае неуспеха вернется -1.

2. **fork** — создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.

3. **open** — создает или открывает файл, если он был создан. В качестве аргументов принимает путь до файла, режим доступа (запись, чтение и т.п.), модификатор доступа (при создании можно указать права для файла). Возвращает в случае успеха файловый дескриптор – положительное число, иначе возвращает -1.

4. **close** — принимает файловый дескриптор в качестве аргумента, удаляет файловый дескриптор из таблицы дескрипторов, в случае успеха вернет 0, в случае неуспеха вернет -1.

5. **read** — предназначена для чтения какого-то числа байт из файла, принимает в качестве аргументов файловый дескриптор, буфер, в который будут записаны данные и число байт. В случае успеха вернет число прочитанных байт, иначе -1.

6. **write** — предназначена для записи какого-то числа байт в файл, принимает в качестве аргументов файловый дескриптор, буфер, из которого будут считаны данные для записи и число байт. В случае успеха вернет число записанных байт, иначе -1.

3 Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы pipe и fork.
2. Написать функцию считывания имён выходных файлов
3. Создать каналы связи для каждого из дочерних процессов
4. Создать функцию обработки ввода
5. Создать функцию фильтрации в родительском процессе
6. Создать функцию фильтрации в дочерних процессах
7. Написать обработку ошибок
8. Написать тесты

4 Исходный код

main.c

```
1
2  #include <fcntl.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  void child_work(int from, int to) {
7    char buf[1];
8    while (read(from, buf, 1) > 0) {
9      char c = buf[0];
10     if (c != 'a' && c != 'e' && c != 'i' && c != 'o' && c != 'u' && c != 'y' &&
11         c != 'A' && c != 'E' && c != 'I' && c != 'O' && c != 'U' && c != 'Y') {
12       write(to, buf, 1);
13     }
14   }
15   close(to);
16   close(from);
17 }
18
19 void parrent_work(int child1, int child2) {
20   char buf[1];
21   int is_even = 0;
22   while (read(STDIN_FILENO, buf, 1) > 0) {
23     if (!is_even) {
24       write(child1, buf, 1);
25     } else {
26       write(child2, buf, 1);
27     }
28     if (buf[0] == '\n') {
29       is_even = !is_even;
30     }
31   }
32
33   close(child1);
34   close(child2);
35 }
36
37 int read_name_and_open_file() {
38   const size_t FILE_NAME_SIZE = 64;
39   char f_name[FILE_NAME_SIZE];
40   char buf[1];
41   int idx = 0;
42   while (idx < FILE_NAME_SIZE && read(STDIN_FILENO, buf, 1) > 0) {
43     if (buf[0] == '\n') {
44       break;
45     }
46     f_name[idx++] = buf[0];
```

```
47    }
48    f_name[idx++] = '\0';
49    return open(f_name, O_WRONLY | O_TRUNC);
50  }
51
52  void error(char* buf, size_t size) { write(STDERR_FILENO, buf, size); }
53  void check_file_id(int id) {
54    if (id == -1) {
55      error("File not found\n", 15);
56      exit(-1);
57    }
58  }
59  void check_pipe_creation(int* pipefd) {
60    if (pipe(pipefd) == -1) {
61      error("Cannot create pipe\n", 19);
62      exit(-2);
63    }
64  }
65  int check_fork() {
66    int fd = fork();
67    if (fd == -1) {
68      error("Cannot create process\n", 22);
69      exit(-3);
70    }
71    return fd;
72  }
73
74  int main(int argc, char* argv[]) {
75    int f1 = read_name_and_open_file();
76    check_file_id(f1);
77    int f2 = read_name_and_open_file();
78    check_file_id(f2);
79
80    int pipefd1[2];
81    check_pipe_creation(pipefd1);
82
83    int child1 = check_fork();
84    if (child1 == 0) {
85      close(pipefd1[1]);
86      child_work(pipefd1[0], f1);
87      return 0;
88    }
89    close(pipefd1[0]);
90
91    int pipefd2[2];
92    check_pipe_creation(pipefd2);
93
94    int child2 = check_fork();
95    if (child2 == 0) {
```

```
96      close(pipefd1[1]);
97      close(pipefd2[1]);
98      child_work(pipefd2[0], f2);
99      return 0;
100   }
101   close(pipefd2[0]);
102
103   parrent_work(pipefd1[1], pipefd2[1]);
104
105   return 0;
106 }
```

5 Пример работы

Первый тест - проверка на обработку случая отсутсвия первого файла

```
windicor@Lina-HP:~$ cat test1
f
windicor@Lina-HP:~$ ./a.out <test1
File not found
```

Второй тест - проверка на обработку случая отсутсвия второго файла

```
windicor@Lina-HP:~$ cat test2
f1
f
windicor@Lina-HP:~$ ./a.out <test2
File not found
```

Третий и четвёртый тесты - проверка работоспособности программы на корректных данных

```
windicor@Lina-HP:~$ cat test3
f1
f2
a b c d e f g h i j k l
m n o p q r s t u v w x y z
windicor@Lina-HP:~$ ./a.out <test3
windicor@Lina-HP:~$ cat f1
b c d  f g h  j k l
windicor@Lina-HP:~$ cat f2
m n  p q r s t  v w x  z
windicor@Lina-HP:~$ cat test4
f1
f2
m n o p q r s t u v w x y z
a b c d e f g h i j k l
m n o p q r s t u v w x y z
```

```
a b c d e f g h i j k l
m n o p q r s t u v w x y z
a b c d e f g h i j k l
m n o p q r s t u v w x y z
a b c d e f g h i j k l
windicor@Lina-HP:~$ ./a.out <test4
windicor@Lina-HP:~$ cat f1
m n  p q r s t  v w x  z
m n  p q r s t  v w x  z
m n  p q r s t  v w x  z
m n  p q r s t  v w x  z
windicor@Lina-HP:~$ cat f2
b c d  f g h  j k l
b c d  f g h  j k l
b c d  f g h  j k l
b c d  f g h  j k l
```

Пятый тест - проверка работоспособности программы на пустых данных

```
windicor@Lina-HP:~$ cat test5
f1
f2
windicor@Lina-HP:~$ ./a.out <test5
windicor@Lina-HP:~$ cat f1
windicor@Lina-HP:~$ cat f2
```

6 Вывод

В процессе работы над лабораторной я научился основам работы с конвейерами и процессами в Си. Процессы занимают важную роль в разработке ПО, так как программы зачастую состоят из нескольких, относительно обособленных, подпрограмм, то есть процессов. Конвейеры как один из способов обмена данными между процессами таже играют немаловажную роль.