Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

ВЗАИМОДЕЙСТВИЕ МЕЖДУ ПРОЦЕССАМИ

Студент: Харьков Павел Александрович
Группа: М8О–206Б–19
Вариант: 20
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021.

Постановка задачи

Цель работы

Приобретение практических навыков в:

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

Задание

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.

Вариант 20: 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 инвертируют строки. Процессы пишут результаты своей работы в стандартный вывод.

Общие сведения о программе.

Программа компилируется из файлов laba_2.c, child.c, get_line.c . Также используется заголовочные файлы: unistd.h, stdlib.h, stdio.h, fcntl.h, string.h. В программе используются следующие системные вызовы:

- fork – создает копию процесса.
- pipe – создает канал данных для взаимодействия между процессами.
- read – читает в память из файлового дескриптора определенное количество байт.
- write – пишет в файловый дескриптор из памяти определенное количество байт.
- open – открывает определенный файл и возвращает его файловый дескриптор.
- close – закрывает файловый дескриптор.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы fork, pipe, open, close, read, write.
2. Написать программу, которая будет работать с 3-мя процессами: один родительский и два дочерних, процессы связываются между собой при помощи pipe.
3. Отладить программу и протестировать на тестах.

Основные файлы программы.

laba_2.c

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

#include "child/child.h"
#include "get_line/get_line.h"

void ReadLineAndWriteToPipe(int fd, int pipe)
{
   char* line = NULL;
   int size;
   if((size = get_line(&line, fd)) == 0)
   {
      exit(-1);
   }

   write(pipe, &size, sizeof(int));
   write(pipe, line, size*sizeof(char));

   free(line);
}

int main()
{
   int id1, id2;
   int pipe1[2];
   int pipe2[2];
   pipe(pipe1);
   pipe(pipe2);

   id1 = fork();
   if (id1 == -1)
```

```
{
   perror("fork");
   exit(-1);
}
else if(id1 == 0)
{
   close(pipe2[0]);
   close(pipe2[1]);

   Child(pipe1);
}
else
{
   id2 = fork();
   if (id2 == -1)
   {
      perror("fork");
      exit(-1);
   }
   else if(id2 == 0)
   {
      close(pipe1[0]);
      close(pipe1[1]);

      Child(pipe2);
   }
   else
   {
      close(pipe1[0]);
      close(pipe2[0]);

      ReadLineAndWriteToPipe(STDIN_FILENO, pipe1[1]);
      ReadLineAndWriteToPipe(STDIN_FILENO, pipe2[1]);

      char* line = NULL;
      int size;
      while((size = get_line(&line, STDIN_FILENO)) != 0)
      {
         if(size > 10)
         {
            if(write(pipe2[1], &size, sizeof(int)) != sizeof(int))
               exit(-1);
            if(write(pipe2[1], line, size*sizeof(char)) != size*sizeof(char))
               exit(-1);
         }
         else
         {
            if(write(pipe1[1], &size, sizeof(int)) != sizeof(int))
               exit(-1);
            if(write(pipe1[1], line, size*sizeof(char)) != size*sizeof(char))
```

```
                exit(-1);
            }
        }
        free(line);

        close(pipe1[1]);
        close(pipe2[1]);
    }
  }

  return 0;
}
```

child.c

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

#include "../get_line/get_line.h"
#include "child.h"

void ReverseLine(char* line, int size)
{
  for(int i = 0; i < size / 2; ++i)
  {
    char tmp = line[size - i - 1];
    line[size - i - 1] = line[i];
    line[i] = tmp;
  }
}

void Child(int pipe[2])
{
  close(pipe[1]);
  int size;
  read(pipe[0], &size, sizeof(int));

  char file_path[size+1];
  file_path[size] = '\0';
  read(pipe[0], file_path, size * sizeof(char));

  int fd;
  if ((fd = open(file_path, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0)
  {
    perror(file_path);
```

```
        exit(-1);
    }

    while(read(pipe[0], &size, sizeof(int)) > 0)
    {
        char line[size+2];
        line[size] = '\n';
        line[size+1] = '\0';
        read(pipe[0], line, size * sizeof(char));

        ReverseLine(line, size);

        write(fd, line, (size+1) * sizeof(char));
    }

    close(pipe[0]);
    close(fd);
}
```

get_line.c

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

#include "get_line.h"

int get_char(int fd)
{
    char c;
    if (read(fd, &c, 1) == 1)
        return (unsigned char)c;
    return EOF;
}

int get_line(char** line_, int fd)
{
    free(*line_);
    int c;
    int size = 0;
    int cap = 4;
    char* line = (char*)malloc(cap * sizeof(char));
    while((c = get_char(fd)) != EOF && c != '\n' && c!= '\r')
    {
        if(size == cap)
        {
            cap *= 2;
            line = (char*)realloc(line, cap * sizeof(char));
            if(line == NULL)
                exit(-1);
        }
```

```
        line[size] = c;
        ++size;
    }
    if(size == cap)
    {
        cap += 1;
        line = (char*)realloc(line, cap * sizeof(char));
        if(line == NULL)
            exit(-1);
    }
    line[size] = '\0';
    *line_ = line;
    return size;
}
```

Пример работы программы.

```
pablo$ make
gcc -c -o src/laba_2.o src/laba_2.c
gcc -c -o src/child/child.o src/child/child.c
gcc -c -o src/get_line/get_line.o src/get_line/get_line.c
gcc -o laba_2 src/laba_2.o src/child/child.o src/get_line/get_line.o
pablo$ cat tests/test1 f
f
s
hello
very long text
pablo$ ./laba_2 < tests/test1
pablo$ cat f
olleh
pablo$ cat s
txet gnol yrev

pablo$ ./laba_2
firstfile
secondfile
shorttext
notshorttext
good
pablo$ cat firstfile
txettrohs
doog
pablo$ cat secondfile
txettrohston
```

Выводы.

Выполнив данную лабораторную работу, я изучил базовые системные вызовы. Научился создавать процессы и узнал, что создается именно копия текущего процесса, а не какой-либо «пустой» процесс. Также я познакомился с однонаправленными каналами (pipe), с помощью которых процессы могли передавать друг другу данные. Каналы очень удобны для таких маленьких задач, так как они не требуют никаких сложных ухищрений при их создании.