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 операционной системе «UNIX».**

Студент: Селивёрстов Д. С. Преподаватель: Миронов Е. С.

Группа: М8О-201Б-21

Вариант 8

Дата: Оценка: Подпись:

**Москва, 2023**

# Условие

Составить и отладить программу на языке Си, осуществляющую работу с процесса- ми и взаимодействие между ними в операционной системе UNIX/LINUX. В резуль- тате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляет- ся через системные сигналы/события и/или каналы (*pipe*). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

# Задание

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 *pipe*1 или в *pipe*2 в зависимости от правила фильтрации. Процесс *child*1 и *child*2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: с вероятностью 80% строки отправляются в pipe1, иначе в pipe2. Дочерние процессы удаляют все гласные из строк.

# Код программы

## utils.h

**#ifndef** UTILS\_H

**#define** UTILS\_H

**#include** <s t d i o . h> **#include** <s t d l i b . h> **#include** <s t r i n g . h>

**char** ∗ Read String ( FILE∗ stream ) ;

**char** ∗ Read String WithoutVowels ( FILE∗ stream ) ;

## #endif parent.h

**#ifndef** PARENT\_H

**#define** PARENT\_H

**#include** <s t d i o . h> **#include** <s t d l i b . h> **#include** <s t r i n g . h> **#include** <unistd . h> **#include** <f c n t l . h> **#include** <errno . h> **#include** <sys / wait . h> **#include** <sys / s t a t . h> **#include** <sys / types . h>

**void** Parent Routine ( **char** ∗ pathToChild , FILE∗ input ) ;

## #endif

**0.0.1 utils.c**

**#include** <u t i l s . h>

**char** ∗ Read String ( FILE∗ stream )

{

**i f** ( f e o f ( stream )) {

**return** NULL;

}

**const int** chunk Size = 256 ;

**char** ∗ b u f f e r = ( **char** ∗) malloc ( chunk Size ) ;

**int** b u f f e r S i z e = chunk Size ;

**i f** ( b u f f e r == NULL)

{

p r i n t f ( " Couldn ’ t ␣ a l l o c a t e ␣ b u f f e r " ) ; e x i t (EXIT\_FAILURE) ;

}

**int** readChar ;

**int** idx = 0 ;

**while** (( readChar = getc ( stream )) != EOF)

{

b u f f e r [ idx++] = readChar ;

**i f** ( idx == b u f f e r S i z e )

{

b u f f e r = r e a l l o c ( b u f f e r , b u f f e r S i z e + chunk Size ) ; b u f f e r S i z e += chunk Size ;

}

**i f** ( readChar == ’ \n ’ ) {

## break ;

}

}

b u f f e r [ idx ] = ’ \0 ’ ;

**return** b u f f e r ;

}

**char** ∗ Read String WithoutVowels ( FILE∗ stream )

{

**i f** ( f e o f ( stream )) {

**return** NULL;

}

**const int** chunk Size = 256 ;

**char** ∗ b u f f e r = ( **char** ∗) malloc ( chunk Size ) ;

**int** b u f f e r S i z e = chunk Size ;

**i f** ( b u f f e r == NULL)

{

p r i n t f ( " Couldn ’ t ␣ a l l o c a t e ␣ b u f f e r " ) ; e x i t (EXIT\_FAILURE) ;

}

**int** readChar ;

**int** idx = 0 ;

**char** ∗ vowels = {"AEIOUYaeiouy" };

**while** (( readChar = getc ( stream )) != EOF)

{

**int** is Vowel = 0 ;

**for** ( **int** i = 0 ; i < ( **int** ) s t r l e n ( vowels ) ; ++i ){

**i f** ( readChar == vowels [ i ] ) { is Vowel = 1 ;

}

}

**i f** ( is Vowel == 0 ){

b u f f e r [ idx++] = readChar ;

}

**i f** ( idx == b u f f e r S i z e )

{

b u f f e r = r e a l l o c ( b u f f e r , b u f f e r S i z e + chunk Size ) ; b u f f e r S i z e += chunk Size ;

}

**i f** ( readChar == ’ \n ’ ) {

## break ;

}

}

b u f f e r [ idx ] = ’ \0 ’ ;

**return** b u f f e r ;

}

## parent.c

**#include** " parent . h"

**#include** " u t i l s . h"

**int** ChoosePipe (){

**return** ( **int** ) rand () % 100 < 8 0 ;

}

**void** Parent Routine ( **char** ∗ pathToChild , FILE∗ f i n )

{

**char** ∗ f i le Name 1 = Read String ( f i n ) ;

**char** ∗ f i le Name 2 = Read String ( f i n ) ;

f i le Name 1 [ s t r l e n ( f i le Name 1 ) − 1 ] = ’ \0 ’ ; f i le Name 2 [ s t r l e n ( f i le Name 2 ) − 1 ] = ’ \0 ’ ;

unlink ( f i le Name 1 ) ; unlink ( f i le Name 2 ) ;

**int** fd 1 [ 2 ] , fd 2 [ 2 ] ;

**i f** ( pipe ( fd 1 ) == −1 | | pipe ( fd 2 ) == −1)

{

p e r r o r ( " c r e a t i n g ␣ pipe ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

pid\_t output File 1 , output File 2 ;

**i f** (( output File 1 = open ( f i le Name 1 , O\_WRONLY | O\_CREAT, S\_IRWXU) ) < 0 )

{

p e r r o r ( " opening ␣output ␣ f i l e ␣1 ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

**i f** (( output File 2 = open ( f i le Name 2 , O\_WRONLY | O\_CREAT, S\_IRWXU) ) < 0 )

{

p e r r o r ( " opening ␣output ␣ f i l e ␣2 ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

f r e e ( f i le Name 1 ) ; f r e e ( f i le Name 2 ) ;

**char** ∗ argv [ 2 ] ; argv [ 0 ] = " c h i l d " ; argv [ 1 ] = NULL;

pid\_t pid 1 = f o r k ( ) ; pid\_t pid 2 = 1 ;

**i f** ( pid 1 > 0 ){ pid 2 = f o r k ( ) ;

}

**i f** ( pid 1 < 0 | | pid 2 < 0 )

{

p e r r o r ( " p r o c e s s ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

**i f** ( pid 1 == 0 )

{

c l o s e ( fd 1 [ 1 ] ) ;

**i f** ( dup2 ( fd 1 [ 0 ] , 0 ) < 0 )

{

p e r r o r ( " duping ␣ pipe ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

**i f** ( dup2 ( output File 1 , 1 ) < 0 )

{

p e r r o r ( " duping ␣output ␣ f i l e ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

execv ( pathToChild , argv ) ;

*// perror (" execv error ) ") ;*

*// e x i t (EXIT\_FAILURE) ;*

}

**else i f** ( pid 2 == 0 )

{

c l o s e ( fd 2 [ 1 ] ) ;

**i f** ( dup2 ( fd 2 [ 0 ] , 0 ) < 0 )

{

p e r r o r ( " duping ␣ pipe ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

**i f** ( dup2 ( output File 2 , 1 ) < 0 )

{

p e r r o r ( " duping ␣output ␣ f i l e ␣ e r r o r ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

execv ( pathToChild , argv ) ;

*// perror (" execv error ) ") ;*

*// e x i t (EXIT\_FAILURE) ;*

}

## else

{

c l o s e ( fd 1 [ 0 ] ) ;

c l o s e ( fd 2 [ 0 ] ) ;

**char** ∗ s t r In p u t = NULL;

**while** (( s t r In p u t = Read String ( f i n )) != NULL)

{

**int** s t r S i z e = s t r l e n ( s t r In p u t ) ;

**i f** ( s t r S i z e > 0 )

{

**i f** ( ChoosePipe ( ) )

{

}

## else

{

}

}

w r i te ( fd 1 [ 1 ] , s tr Input , s t r S i z e ) ;

w r i te ( fd 2 [ 1 ] , s tr Input , s t r S i z e ) ;

f r e e ( s t r In p u t ) ;

}

**i f** ( s t r In p u t == NULL)

{

**char** terminator = ’ \0 ’ ;

w r i te ( fd 1 [ 1 ] , &terminator , 1 ) ; w r i te ( fd 2 [ 1 ] , &terminator , 1 ) ;

}

c l o s e ( fd 1 [ 1 ] ) ;

c l o s e ( fd 2 [ 1 ] ) ;

}

}

## child.c

**#include** <s t d i o . h>

**#include** <s t d l i b . h>

**#include** <s t r i n g . h> **#include** <unistd . h> **#include** <f c n t l . h> **#include** <ctype . h> **#include** <sys / wait . h> **#include** <sys / s t a t . h>

**#include** " u t i l s . h"

**int** main ( **int** argc , **char** ∗ argv [ ] )

{

**i f** ( argc < 1 )

{

p e r r o r ( " too ␣few ␣arguments ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

**i f** ( s t r l e n ( argv [ 0 ] ) < 1 )

{

p e r r o r ( " too ␣few ␣arguments ␣ ) " ) ; e x i t (EXIT\_FAILURE) ;

}

**char** ∗ s t r In p u t ;

**while** (( s t r In p u t = Read String WithoutVowels ( s t d i n )) != NULL)

{

w r i te ( 1 , s tr Input , s t r l e n ( s t r In p u t ) ) ; f r e e ( s t r In p u t ) ;

}

**return** 0 ;

}

# Выводы

Вызов *fork* дублирует породивший его процесс со всеми его переменными, файловыми дескрипторами, приоритетами процесса, рабочий и корневой каталоги, и сегментами выделенной памяти.

Ребёнок **не** наследует:

* идентификатора процесса (PID, PPID);
* израсходованного времени ЦП (оно обнуляется);
* сигналов процесса-родителя, требующих ответа;
* блокированных файлов (record locking).

В процессе выполнения лабораторной работы были приобретены навыки практи- ческого применения создания, обработки и отслеживания их состояния. Для выполне- ния данного варианта задания создание потоков как таковых не требуется, так как всю работу выполняет системный вызов «exec».