Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Кафедра «Информатика»
кафедра

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 1

Управление процессами в ОС GNU/Linux
Тема

Преподаватель			А. С. Кузнецов
		Подпись, дата	Инициалы, Фамилия
Студент	КИ19-17/1Б, №031939174		А. К. Никитин
	Номер группы, зачетной книжки	Подпись, дата	Инициалы, Фамилия

Красноярск 2021

1 Цель

Изучение особенностей программной реализации многозадачных приложений в ОС GNU/Linux.

2 Задачи

1. Ознакомиться с краткими теоретическими сведениями по управлению процессами в ОС GNU/Linux.

2. Разработать программу согласно индивидуальному заданию, являющейся дочерним процессом, которая запускается родительским процессом с использованием концепции *fork-and-exec*.

3. Произвести разработку юнит-тестов основных функциональных блоков кода дочернего процесса с использованием *CUnit*.

Описание варианта:

Программа принимает от пользователя три строки, (первая и третья строки — это правильные рациональные или десятичные дроби вида «1/3» или «0,5», вторая строка — это знак арифметической операции вида «+», «-», «*», «/» либо операции сравнения «<», «>», «=», «!=», «>=», «<=»), выполняет требуемую операцию над полученными операндами, и выводит результат на экран. Обеспечить также сокращение дроби при необходимости. Если оба операнда арифметической операции являются рациональными дробями, результатом тоже должна быть рациональная дробь. Для операций сравнения достаточно результата «Истина» или «Ложь».

3 Исходные тексты программ

На листинге 1 представлен код программы с реализацией основного алгоритма.

Листинг 1 – Сложение и сравнение дробей

```
/*! \file   algorithm.c
 *  \brief  Fraction calculation and comparison
 *  \author Nikitin Alexander, KI19-17/1Б
 */

#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define TRUE 1
#define FALSE 0
#define COMMON 1
#define DECIMAL 0
#define MAX_DECIMAL_PART_LENGTH 2
#define NUMBER_OF_COMPARISON_OPERATORS 10
#define MAX_LENGTH_COMPARISON_OPERATORS 2

typedef struct arrayInfo
{
    int* array;
    int length;
} arrayInfo_t;

struct fractionParts
{
    int firstPart;
    int secondPart;
    int isNegative;
    int isWrong;
};

typedef struct fractionInfo
{
    int firstPart;
```

```c
    int secondPart;
    int type;
    int isNegative;
    int isWrong;
} fractionInfo_t;

/*! \brief Inputs the string from user console into the variable
 *
 *  \param[out] word String variable of arbitrary length
 *
 *  \return Nothing
 */

void inputString(char** word)
{
    int count = 0;
    char inputChar = 0;

    fflush(stdin);
    *word = NULL;

    while(1)
    {
        inputChar = getchar();
        if (inputChar == '\n')
            break;
        else
        {
            *word = realloc(*word, count + 1);
            (*word)[count] = inputChar;
            count++;
        }
    }
    (*word)[count] = '\0';
}

/*! \brief Checks if the string contains only digits
 *
 *  \param[in] string Numeric or non-numeric string
 *
 *  \return Is the string contains only digits or not (TRUE or FALSE)
 */
```

```
int checkInt(char* string)
{
    char* intChars = "0123456789";

    for (int i = 0; i < strlen(string); i++)
    {
        // Проверка минуса перед числом
        if (i == 0 && string[i] == '-')
            continue;

        if (strchr(intChars, string[i]) == NULL)
            return FALSE;
    }

    return TRUE;
}

/*! \brief Divides fraction in two parts and returns both of the values. 
First part is numeric part, second part is
 *  fractional part. Both are integers.
 *
 *  \param[in] string Source fraction
 *  \param[in] pos Position of separator ('.' or '/')
 *
 *  \return First and second numbers of fraction
 */

struct fractionParts splitFraction(char* string, int pos)
{
    struct fractionParts output;

    char* firstPart = NULL;
    char* secondPart = NULL;

    int firstNumber;
    int secondNumber;

    char* pEnd = NULL;

    if (pos <= 0 || pos > strlen(string))
    {
        output.isWrong = TRUE;
        return output;
    }

    // Срез целой части
    for (int i = 0; i < pos; i++)
    {
        firstPart = realloc(firstPart, i + 1);
        firstPart[i] = string[i];
    }
    firstPart[pos] = '\0';

    // Срез дробной части
    for (int i = pos + 1, j = 0; i < strlen(string); i++, j++)
    {
        secondPart = realloc(secondPart, j + 1);
        secondPart[j] = string[i];
    }
    secondPart[strlen(string) - pos - 1] = '\0';

    if (!checkInt(firstPart) || !checkInt(secondPart))
    {
        output.isWrong = TRUE;
        return output;
    }

    if (firstPart[0] == '-')
        output.isNegative = TRUE;
    else
        output.isNegative = FALSE;

    firstNumber = abs(strtol(firstPart, &pEnd, 10));
    secondNumber = abs(strtol(secondPart, &pEnd, 10));

    output.firstPart = firstNumber;
    output.secondPart = secondNumber;
    output.isWrong = FALSE;

    return output;
}

/*! \brief Tries to transform string into fractionString.
```

```
 *
 *  \param[in] fractionString Potential fractionString (X.X or X/X)
 *
 *  \return All essential information about fraction (if fraction.isWrong == FALSE)
 */

fractionInfo_t makeIntoFraction(char* fractionString)
{
    int posSeparator;

    char fractionSeparator;
    char fractionSeparatorString[2];

    struct fractionParts fraction;
    fractionInfo_t convertedFraction;

    fractionSeparator = strpbrk(fractionString, "/,.")[0];
    fractionSeparatorString[0] = fractionSeparator;
    posSeparator = strcspn(fractionString, fractionSeparatorString) + 1;

    fraction = splitFraction(fractionString, posSeparator - 1);

    if (fraction.isWrong || fraction.secondPart == 0)
    {
        convertedFraction.isWrong = TRUE;
        return convertedFraction;
    }

    // Заполнение информации о дроби
    convertedFraction.firstPart = fraction.firstPart;
    convertedFraction.secondPart = fraction.secondPart;
    convertedFraction.isNegative = fraction.isNegative;

    switch (fractionSeparator)
    {
    case '/':
    {
        convertedFraction.type = COMMON;
        convertedFraction.isWrong = FALSE;
        return convertedFraction;
    }
```

```
          case ',':
          case '.':
          {
              convertedFraction.type = DECIMAL;
              convertedFraction.isWrong = FALSE;
              return convertedFraction;
          }
          default:
          {
              convertedFraction.isWrong = TRUE;
              return convertedFraction;
          }
          }
      }

      /*! \brief Calculates the highest power of the number. For example, 1274 =
1 * 10^3 + 2 * 10^2 + 7 * 10^1 + 4 * 10^0.
       * So, the highest power (10^3) is 4.
       *
       *  \param[in] number Source number
       *
       *  \return The highest power
       */

      int calculateHighestPower(int number)
      {
          int power = 0;

          while (abs(number) > 0)
          {
              number /= 10;
              power++;
          }
          return power;
      }

      /*! \brief Calculates the highest negative power of the decimal number. For
example, 12.74 = 1 * 10^1 + 2 * 10^0
       * + 7 * 10^-1 + 4 * 10^-2. So, the highest negative power (10^-2) is -2.
       *
       *  \param[in] number Source number
       *
       *  \return The highest negative power. Maximum value is
MAX_DECIMAL_PART_LENGTH.
       */

      int calculateHighestNegativePower(double number)
      {
          int power = 0;
          while ((number - (int) number) != 0 && power <= MAX_DECIMAL_PART_LENGTH)
          {
              number *= 10;
              power++;
          }

          return power;
      }

      /*! \brief Checks if the operation type is supportable in the program.
       * List of operations: ("+", "-", "*", "/", "<", ">", "=", "!=", ">=", "<=")
       *
       *  \param[in] operation Operation string
       *
       *  \return If the operator is appropriate (TRUE or FALSE)
       */

      int checkOperation(char* operation)
      {
          char
allOperations[NUMBER_OF_COMPARISON_OPERATORS][MAX_LENGTH_COMPARISON_OPERATORS +
1]
          = {"+", "-", "*", "/", "<", ">", "=", "!=", ">=", "<="};

          for (int i = 0; i < sizeof (allOperations) / sizeof (*allOperations);
i++)
              if (strcmp(allOperations[i], operation) == 0)
                  return TRUE;
          return FALSE;
      }

      /*! \brief Finds and returns simple dividers of the number. For example,
simple dividers of 60 are 2, 2, 3, 5.
       *
```

```
 *  \param[in] number Positive integer number
 *
 *  \return Simple dividers of a number
 */

arrayInfo_t findSimpleDividers(int number)
{
    arrayInfo_t dividers;
    int length = 0;
    int i = 2;
    int j = 0;

    int numberCopy = number;

    while (numberCopy > 1)
    {
        while (numberCopy % i == 0)
        {
            numberCopy /= i;
            length++;
        }
        i++;
    }

    int* dividersArray = (int*) malloc(length * sizeof (int));

    i = 2;
    numberCopy = number;
    while (numberCopy > 1)
    {
        while (numberCopy % i == 0)
        {
            numberCopy /= i;
            dividersArray[j] = i;
            j++;
        }
        i++;
    }
    dividers.array = dividersArray;
    dividers.length = length;

    return dividers;
```

```
      }

      /*! \brief Subtract one array from another and returns the result. For
example, (3, 4, 2, 6, 2) - (2, 6) = (3, 4, 2)
       *
       *  \param[in] minuend The minuend array
       *  \param[in] subtrahend The subtrahend array
       *
       *  \return The residual array
       */

      arrayInfo_t subtractArrays(arrayInfo_t minuend, arrayInfo_t subtrahend)
      {
          arrayInfo_t residual;

          if (subtrahend.length == 0)
              return  minuend;

          if (minuend.length == 0)
          {
              residual.array = NULL;
              residual.length = 0;
              return residual;
          }

          int count = 0;

          // Вспомогательный массив для определения, какие элементы удалять
          int minuendIndexes[minuend.length];
          for (int i = 0; i < minuend.length; i++)
          {
              minuendIndexes[i] = i;
          }

          for (int i = 0; i < subtrahend.length; i++)
          {
              for (int j = 0; j < minuend.length; j++)
              {
                  if ((minuend.array[j] == subtrahend.array[i]) &&
(minuendIndexes[j] != -1))
                  {
                      minuendIndexes[j] = -1;
```

```
                count++;
                break;
            }
        }
    }

    int* residualArray = (int*) malloc(count * sizeof (int));

    count = 0;
    for (int i = 0; i < minuend.length; i++)
        if (minuendIndexes[i] != -1)
        {
            residualArray[count] = minuend.array[i];
            count++;
        }

    residual.array = residualArray;
    residual.length = count;

    return residual;
}

/*! \brief Finds least common multiple of two numbers.
 *
 *  \param[in] number1 First number of LCM
 *  \param[in] number2 Second number of LCM
 *
 *  \return Least common multiple
 */

int findLCM(int number1, int number2)
{
    arrayInfo_t dividers1 = findSimpleDividers(number1);

    arrayInfo_t dividers2 = findSimpleDividers(number2);
    arrayInfo_t remainingDividers = subtractArrays(dividers2, dividers1);
    int LCM = 1;

    for (int i = 0; i < dividers1.length; i++)
        LCM *= dividers1.array[i];

    // Домножаю только неповторяющиеся значения
    for (int i = 0; i < remainingDividers.length; i++)
    {
        LCM *= remainingDividers.array[i];
    }

    return LCM;
}

/*! \brief Finds greatest common divisor of two numbers.
 *
 *  \param[in] number1 First number of GCD
 *  \param[in] number2 Second number of GCD
 *
 *  \return Greatest common divisor
 */

int findGCD(int number1, int number2)
{
    int GCD = 1;
    if (number1 == 0 || number2 == 0)
        return GCD;

    GCD = number1 * number2 / findLCM(number1, number2);
    return GCD;
}

/*! \brief Reduce fraction to a common denominator.
 *
 *  \param[in] fraction Source fraction
 *
 *  \return Nothing
 */

void reduceFraction(fractionInfo_t* fraction)
{
    if (fraction->type == DECIMAL)
    {
        fraction->isWrong = TRUE;
        return;
    }

    int GCD = findGCD(fraction->firstPart, fraction->secondPart);
```

```
          fraction->firstPart /= GCD;
          fraction->secondPart /= GCD;
      }

      /*! \brief Transforms fraction into decimal number.
       *
       *  \param[in] fraction Source fraction
       *
       *  \return Decimal (double) format of fraction
       */

      double toDouble(fractionInfo_t fraction)
      {
          double doubleFraction;
          if (fraction.type == COMMON)
              doubleFraction = (double) fraction.firstPart / (double)
fraction.secondPart;
          else
              doubleFraction = (double) fraction.firstPart +
                               (double) fraction.secondPart / pow(10,
calculateHighestPower(fraction.secondPart));

          if (fraction.isNegative)
              doubleFraction *= -1;

          return doubleFraction;
      }

      /*! \brief Performs arithmetic operations on two fractions. Possible
operations are '+', '-', '*', '/'. Fractions
       * can be decimal or common. If both are common, the result is common, too.
In other cases it is decimal.
       *
       *  \param[in] fraction1 First part of arithmetic operation
       *  \param[in] fraction2 Second part of arithmetic operation
       *  \param[in] operation Operation type ('+', '-', '*', or '/')
       *
       *  \return Result of arithmetic operation
       */

      fractionInfo_t calculate(fractionInfo_t fraction1, fractionInfo_t
fraction2, char* operation)
```

```
      {
          enum Case {Plus, Minus, Multiply, Divide};

          enum Case operationCode;
          if (strcmp(operation, "+") == 0)
              operationCode = Plus;
          else if (strcmp(operation, "-") == 0)
              operationCode = Minus;
          else if (strcmp(operation, "*") == 0)
              operationCode = Multiply;
          else
              operationCode = Divide;

          fractionInfo_t result;
          if (fraction1.type == COMMON && fraction2.type == COMMON)
          {
              int LCM = findLCM(fraction1.secondPart, fraction2.secondPart);
              result.type = COMMON;

              // Учет знака
              if (fraction1.isNegative)
                  fraction1.firstPart *= -1;

              if (fraction2.isNegative)
                  fraction2.firstPart *= -1;

              switch (operationCode)
              {
              case Plus:
                  result.firstPart    =    fraction1.firstPart    *    LCM    /
fraction1.secondPart +
                                     fraction2.firstPart      *      LCM      /
fraction2.secondPart;
                  result.secondPart = LCM;
                  break;

              case Minus:
                  result.firstPart    =    fraction1.firstPart    *    LCM    /
fraction1.secondPart -
                                     fraction2.firstPart      *      LCM      /
fraction2.secondPart;
                  result.secondPart = LCM;
```

```
                break;

            case Multiply:
                result.firstPart = fraction1.firstPart * fraction2.firstPart;
                result.secondPart = fraction1.secondPart * fraction2.secondPart;
                break;

            case Divide:
                result.firstPart = fraction1.firstPart * fraction2.secondPart;
                result.secondPart = fraction1.secondPart * fraction2.firstPart;
                break;
            }

            if ((result.firstPart < 0 && result.secondPart > 0) || (result.firstPart > 0 && result.secondPart < 0))
                result.isNegative = TRUE;
            else
                result.isNegative = FALSE;

            result.firstPart = abs(result.firstPart);
            result.secondPart = abs(result.secondPart);

            result.isWrong = FALSE;

            reduceFraction(&result);
            return result;
        }
        else
        {
            result.type = DECIMAL;

            double decimal1 = toDouble(fraction1);
            double decimal2 = toDouble(fraction2);
            double resultDecimal;

            switch (operationCode)
            {
            case Plus:
                resultDecimal = decimal1 + decimal2;
                break;
```

```
              case Minus:
                  resultDecimal = decimal1 - decimal2;
                  break;

              case Multiply:
                  resultDecimal = decimal1 * decimal2;
                  break;

              case Divide:
                  resultDecimal = decimal1 / decimal2;
                  break;
              }

              if (resultDecimal < 0)
              {
                  result.isNegative = TRUE;
                  resultDecimal *= -1;
              }
              else
                  result.isNegative = FALSE;

              int negativePower = calculateHighestNegativePower(resultDecimal);

              result.firstPart = (int) resultDecimal;
              result.secondPart = (int) (resultDecimal * pow(10, negativePower) -
                                         (int)    resultDecimal    *    pow(10,
negativePower));

              result.isWrong = FALSE;
              return result;
          }
      }

      /*! \brief Compares two fractions. Possible comparison operations are '>',
'<', '=', '!=', '>=', '<='.
       * Fractions can be decimal or common. Returns true or false.
       *
       *  \param[in] fraction1 First part of comparison
       *  \param[in] fraction2 Second part of comparison
       *  \param[in] operation Operation type ('>', '<', '=', '!=', '>=', or '<=')
       *
       *  \return Result of comparison (TRUE or FALSE)
```

```
 */

int  compare(fractionInfo_t  fraction1,  fractionInfo_t  fraction2,  char*
operation)
{
    enum Case {Lesser, Greater, Equal, NotEqual, GreaterEqual, LesserEqual};

    enum Case operationCode;
    if (strcmp(operation, "<") == 0)
        operationCode = Lesser;
    else if (strcmp(operation, ">") == 0)
        operationCode = Greater;
    else if (strcmp(operation, "=") == 0)
        operationCode = Equal;
    else if (strcmp(operation, "!=") == 0)
        operationCode = NotEqual;
    else if (strcmp(operation, ">=") == 0)
        operationCode = GreaterEqual;
    else
        operationCode = LesserEqual;

    double decimal1 = toDouble(fraction1);
    double decimal2 = toDouble(fraction2);

    switch (operationCode)
    {
        case Lesser:
            return decimal1 < decimal2;

        case Greater:
            return decimal1 > decimal2;

        case Equal:
            return decimal1 == decimal2;

        case NotEqual:
            return decimal1 != decimal2;

        case GreaterEqual:
            return decimal1 >= decimal2;

        case LesserEqual:
```

```
                  return decimal1 <= decimal2;
          }
      }

      /*! \brief Allows user to perform basic fraction calculations and
comparisons.
       *
       * \return Successful execution code (0)
       */
      int userInput()
      {
          char* firstFraction;
          char* secondFraction;
          char* operation;

          fractionInfo_t fraction1;
          fractionInfo_t fraction2;
          fractionInfo_t result;

          setlocale(LC_ALL, "C");

          do
          {
              puts("Input first fraction:");
              inputString(&firstFraction);
              fraction1 = makeIntoFraction(firstFraction);
          } while (fraction1.isWrong);

            do
          {
              puts("Input operation:");
              inputString(&operation);
          } while (checkOperation(operation) == 0);

          do
          {
              puts("Input second fraction:");
              inputString(&secondFraction);
              fraction2 = makeIntoFraction(secondFraction);
          } while (fraction2.isWrong);

          if (strcmp(operation, "+") == 0 || strcmp(operation, "-") == 0 ||
              strcmp(operation, "/") == 0 || strcmp(operation, "*") == 0 )
          {
              result = calculate(fraction1, fraction2, operation);

              if (result.isNegative)
                  printf("-");
              if (result.type == COMMON)
                  printf("%d/%d\n", result.firstPart, result.secondPart);
              else
                  printf("%d.%d\n", result.firstPart, result.secondPart);
          }
          else
              if (compare(fraction1, fraction2, operation))
                        puts("True");
                  else
                        puts("False");

          return 0;
      }
```

На листинге 2 представлен код юнит-тестов к функциям основного алгоритма.

Листинг 2 – Юнит-тесты

```
#include <CUnit/Cunit.h>
#include “algorithm.c”

void test_checkIntTrue()
{
    CU_ASSERT(checkInt(“654321”) == 1);
}

void test_checkIntFalse()
{
    CU_ASSERT(checkInt(“654321qwerty”) == 0);
}

void test_splitFractionRight1()
{
    struct fractionParts expectedResult = {1, 3, FALSE, FALSE};
    struct fractionParts realResult = splitFraction(“1/3”, 1);
    CU_ASSERT(expectedResult.firstPart == realResult.firstPart);
    CU_ASSERT(expectedResult.secondPart == realResult.secondPart);
    CU_ASSERT(expectedResult.isNegative == realResult.isNegative);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_splitFractionRight2()
{
    struct fractionParts expectedResult = {3, 2, TRUE,  FALSE};
    struct fractionParts realResult = splitFraction(“-3.2”, 2);
    CU_ASSERT(expectedResult.firstPart == realResult.firstPart);
    CU_ASSERT(expectedResult.secondPart == realResult.secondPart);
    CU_ASSERT(expectedResult.isNegative == realResult.isNegative);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_splitFractionWrong()
{
    struct fractionParts expectedResult;
    expectedResult.isWrong = TRUE;
    struct fractionParts realResult = splitFraction(“3.3/2”, 1);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_makeIntoFractionCommonNormal()
{
    fractionInfo_t expectedResult = {23, 2, COMMON, FALSE, FALSE};
    fractionInfo_t realResult = makeIntoFraction(“23/2”);
    CU_ASSERT(expectedResult.firstPart == realResult.firstPart);
    CU_ASSERT(expectedResult.secondPart == realResult.secondPart);
    CU_ASSERT(expectedResult.type == realResult.type);
    CU_ASSERT(expectedResult.isNegative == realResult.isNegative);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_makeIntoFractionCommonZeroNumerator()
{
    fractionInfo_t expectedResult = {0, 2, COMMON, FALSE, FALSE};
    fractionInfo_t realResult = makeIntoFraction(“0/2”);
    CU_ASSERT(expectedResult.firstPart == realResult.firstPart);
    CU_ASSERT(expectedResult.secondPart == realResult.secondPart);
    CU_ASSERT(expectedResult.type == realResult.type);
    CU_ASSERT(expectedResult.isNegative == realResult.isNegative);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_makeIntoFractionCommonZeroDenominator()
{
    fractionInfo_t expectedResult;
    expectedResult.isWrong = TRUE;
    fractionInfo_t realResult = makeIntoFraction(“2/0”);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_makeIntoFractionDecimal()
{
    fractionInfo_t expectedResult = {23, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t realResult = makeIntoFraction(“23.2”);
    CU_ASSERT(expectedResult.firstPart == realResult.firstPart);
    CU_ASSERT(expectedResult.secondPart == realResult.secondPart);
    CU_ASSERT(expectedResult.type == realResult.type);
    CU_ASSERT(expectedResult.isNegative == realResult.isNegative);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_makeIntoFractionNegative()
{
    fractionInfo_t expectedResult = {23, 2, DECIMAL, TRUE, FALSE};
    fractionInfo_t realResult = makeIntoFraction(“-23.2”);
    CU_ASSERT(expectedResult.firstPart == realResult.firstPart);
    CU_ASSERT(expectedResult.secondPart == realResult.secondPart);
    CU_ASSERT(expectedResult.type == realResult.type);
    CU_ASSERT(expectedResult.isNegative == realResult.isNegative);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_makeIntoFractionWrong()
{
    fractionInfo_t expectedResult;
    expectedResult.isWrong = TRUE;
    fractionInfo_t realResult = makeIntoFraction(“f23/2”);
    CU_ASSERT(expectedResult.isWrong == realResult.isWrong);
}

void test_calculateHighestPowerNormal1()
```

```
{
    CU_ASSERT(calculateHighestPower(12345678) == 8);
}

void test_calculateHighestPowerNormal2()
{
    CU_ASSERT(calculateHighestPower(1) == 1);
}

void test_calculateHighestPowerNegative()
{
    CU_ASSERT(calculateHighestPower(-12345678) == 8);
}

void test_calculateHighestNegativePowerNormal()
{
    CU_ASSERT(calculateHighestNegativePower(123.45) == 2);
}

void test_calculateHighestNegativePowerBordered()
{
    CU_ASSERT(calculateHighestNegativePower(123.4567890) == 3);
}

void test_calculateHighestNegativePowerNegative()
{
    CU_ASSERT(calculateHighestNegativePower(-123.42) == 2);
}

void test_checkOperationTrue()
{
    CU_ASSERT(checkOperation(“>=”) == TRUE);
}

void test_checkOperationFalse()
{
    CU_ASSERT(checkOperation(“+-“) == FALSE);
}

void test_findSimpleDividersNormal()
{
    int expectedArray[] = {2, 2, 3, 5};
    arrayInfo_t realResult = findSimpleDividers(60);
    CU_ASSERT(realResult.length = 4);
    for (int I = 0; I < 4; i++)
        CU_ASSERT(expectedArray[i] == realResult.array[i]);
}

void test_subtractArraysNormal()
{
    arrayInfo_t minuendInfo;
    int minuend[] = {4, 2, 3, 5, 2};
    minuendInfo.array = minuend;
    minuendInfo.length = 5;

    arrayInfo_t subtrahendInfo;
    int subtrahend[] = {2, 5};
    subtrahendInfo.array = subtrahend;
    subtrahendInfo.length = 2;

    int expectedArray[] = {4, 3, 2};
    arrayInfo_t realResult = subtractArrays(minuendInfo, subtrahendInfo);
    CU_ASSERT(realResult.length = 3);

    for (int I = 0; I < 3; i++)
        CU_ASSERT(expectedArray[i] == realResult.array[i]);
}

void test_subtractArraysZeroSubtrahend()
{
    arrayInfo_t minuendInfo;
    int minuend[] = {4, 2, 3, 5, 2};
    minuendInfo.array = minuend;
    minuendInfo.length = 5;

    arrayInfo_t subtrahendInfo;
    int subtrahend[] = {};
    subtrahendInfo.array = subtrahend;
    subtrahendInfo.length = 0;

    int expectedArray[] = {4, 2, 3, 5, 2};
    arrayInfo_t realResult = subtractArrays(minuendInfo, subtrahendInfo);
    CU_ASSERT(realResult.length = 5);
    for (int I = 0; I < 5; i++)
        CU_ASSERT(expectedArray[i] == realResult.array[i]);
}

void test_subtractArraysZeroMinuend()
{
    arrayInfo_t minuendInfo;
    int minuend[] = {};
    minuendInfo.array = minuend;
    minuendInfo.length = 0;

    arrayInfo_t subtrahendInfo;
    int subtrahend[] = {2, 5};
    subtrahendInfo.array = subtrahend;
    subtrahendInfo.length = 2;

    int expectedArray[] = {};
    arrayInfo_t realResult = subtractArrays(minuendInfo, subtrahendInfo);
    CU_ASSERT(realResult.length == 0);
}

void test_findLCM()
{
    CU_ASSERT(findLCM(60, 75) == 300);
}

void test_findGCD()
{
    CU_ASSERT(findGCD(60, 75) == 15);
}

void test_reduceFractionNormal1()
{
    fractionInfo_t fraction = {12, 16, COMMON, FALSE, FALSE};
    reduceFraction(&fraction);
    CU_ASSERT(fraction.firstPart == 3);
    CU_ASSERT(fraction.secondPart == 4);
    CU_ASSERT(fraction.type == COMMON);
    CU_ASSERT(fraction.isNegative == FALSE);
    CU_ASSERT(fraction.isWrong == FALSE);
}
```

```
void test_reduceFractionNormal2()
{
    fractionInfo_t fraction = {14, 2, COMMON, TRUE, FALSE};
    reduceFraction(&fraction);
    CU_ASSERT(fraction.firstPart == 7);
    CU_ASSERT(fraction.secondPart == 1);
    CU_ASSERT(fraction.type == COMMON);
    CU_ASSERT(fraction.isNegative == TRUE);
    CU_ASSERT(fraction.isWrong == FALSE);
}

void test_reduceFractionWrongType()
{
    fractionInfo_t fraction = {16, 32, DECIMAL, FALSE, FALSE};
    reduceFraction(&fraction);
    CU_ASSERT(fraction.isWrong == TRUE);
}

void test_toDoubleCommonNormal()
{
    fractionInfo_t fraction = {12, 16, COMMON, FALSE, FALSE};
    CU_ASSERT(toDouble(fraction) == 0.75);
}

void test_toDoubleCommonNegative()
{
    fractionInfo_t fraction = {12, 16, COMMON, TRUE, FALSE};
    CU_ASSERT(toDouble(fraction) == -0.75);
}

void test_toDoubleCommonZero()
{
    fractionInfo_t fraction = {0, 16, COMMON, FALSE, FALSE};
    CU_ASSERT(toDouble(fraction) == 0.0);
}

void test_toDoubleDecimalNormal()
{
    fractionInfo_t fraction = {12, 16, DECIMAL, FALSE, FALSE};
    CU_ASSERT(toDouble(fraction) == 12.16);
}
```

```
void test_toDoubleDecimalNegative()
{
    fractionInfo_t fraction = {0, 16, DECIMAL, TRUE, FALSE};
    CU_ASSERT(toDouble(fraction) == -0.16);
}

void test_toDoubleDecimalZero()
{
    fractionInfo_t fraction = {0, 0, DECIMAL, FALSE, FALSE};
    CU_ASSERT(toDouble(fraction) == 0.0);
}

void test_calculateCommonPlus()
{
    fractionInfo_t first = {23, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 35, COMMON, FALSE, FALSE};
    char* operation = “+”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 837);
    CU_ASSERT(realResult.secondPart == 70);
    CU_ASSERT(realResult.type == COMMON);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateCommonMinus()
{
    fractionInfo_t first = {23, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 25, COMMON, TRUE, FALSE};
    char* operation = “-“;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 607);
    CU_ASSERT(realResult.secondPart == 50);
    CU_ASSERT(realResult.type == COMMON);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateCommonMultiply()
{
    fractionInfo_t first = {23, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 25, COMMON, FALSE, FALSE};
    char* operation = “*”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 184);
    CU_ASSERT(realResult.secondPart == 25);
    CU_ASSERT(realResult.type == COMMON);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateCommonDivide()
{
    fractionInfo_t first = {23, 6, COMMON, FALSE, FALSE};
    fractionInfo_t second = {8, 76, COMMON, FALSE, FALSE};
    char* operation = “/”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 437);
    CU_ASSERT(realResult.secondPart == 12);
    CU_ASSERT(realResult.type == COMMON);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateDecimalPlus()
{
    fractionInfo_t first = {23, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = “+”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 40);
    CU_ASSERT(realResult.secondPart == 15);
    CU_ASSERT(realResult.type == DECIMAL);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateDecimalMinus()
{
    fractionInfo_t first = {8, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = “-“;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 8);
    CU_ASSERT(realResult.secondPart == 75);
    CU_ASSERT(realResult.type == DECIMAL);
    CU_ASSERT(realResult.isNegative == TRUE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateDecimalMultiply()
{
    fractionInfo_t first = {8, 3, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 85, DECIMAL, TRUE, FALSE};
    char* operation = “*”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 139);
    CU_ASSERT(realResult.secondPart == 855);
    CU_ASSERT(realResult.type == DECIMAL);
    CU_ASSERT(realResult.isNegative == TRUE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateDecimalDivide()
{
    fractionInfo_t first = {9, 2, DECIMAL, TRUE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, TRUE, FALSE};
    char* operation = “/”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 0);
    CU_ASSERT(realResult.secondPart == 542);
    CU_ASSERT(realResult.type == DECIMAL);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_calculateBothType()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = “+”;
    fractionInfo_t realResult = calculate(first, second, operation);
    CU_ASSERT(realResult.firstPart == 21);
    CU_ASSERT(realResult.secondPart == 45);
    CU_ASSERT(realResult.type == DECIMAL);
    CU_ASSERT(realResult.isNegative == FALSE);
    CU_ASSERT(realResult.isWrong == FALSE);
}

void test_compareCommonLesser()
{
    fractionInfo_t first = {9, 2, COMMON,  FALSE, FALSE};
    fractionInfo_t second = {16, 95, COMMON, FALSE, FALSE};
    char* operation = “<”;
    CU_ASSERT(compare(first, second, operation) == FALSE);
}

void test_compareCommonGreater()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 95, COMMON, FALSE, FALSE};
    char* operation = “>”;
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

void test_compareCommonEqualTrue()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {9, 2, COMMON, FALSE, FALSE};
    char* operation = “=”;
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

void test_compareCommonEqualFalse()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 95, COMMON, FALSE, FALSE};
    char* operation = “=”;
    CU_ASSERT(compare(first, second, operation) == FALSE);
}

void test_compareCommonNotEqualTrue()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {9, 2, COMMON, FALSE, FALSE};
    char* operation = “!=”;
    CU_ASSERT(compare(first, second, operation) == FALSE);
}
```

```
void test_compareCommonNotEqualFalse()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 95, COMMON, FALSE, FALSE};
    char* operation = "!=";
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

void test_compareDecimalLesser()
{
    fractionInfo_t first = {9, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = "<";
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

void test_compareDecimalGreater()
{
    fractionInfo_t first = {9, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = ">";
    CU_ASSERT(compare(first, second, operation) == FALSE);
}

void test_compareDecimalEqualTrue()
{
    fractionInfo_t first = {9, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {9, 2, DECIMAL, FALSE, FALSE};
    char* operation = "=";
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

void test_compareDecimalEqualFalse()
{
    fractionInfo_t first = {9, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = "=";
    CU_ASSERT(compare(first, second, operation) == FALSE);
}

void test_compareDecimalNotEqualTrue()
```

```
{
    fractionInfo_t first = {9, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {9, 2, DECIMAL, FALSE, FALSE};
    char* operation = “!=”;
    CU_ASSERT(compare(first, second, operation) == FALSE);
}

void test_compareDecimalNotEqualFalse()
{
    fractionInfo_t first = {9, 2, DECIMAL, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = “!=”;
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

void test_compareBothType()
{
    fractionInfo_t first = {9, 2, COMMON, FALSE, FALSE};
    fractionInfo_t second = {16, 95, DECIMAL, FALSE, FALSE};
    char* operation = “<”;
    CU_ASSERT(compare(first, second, operation) == TRUE);
}

int main()
{
    CU_pSuite pSuite = NULL;
    CU_initialize_registry();

    pSuite = CU_add_suite(“checkInt”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_checkIntTrue);
    CU_ADD_TEST(pSuite, test_checkIntFalse);

    pSuite = CU_add_suite(“splitFraction”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_splitFractionRight1);
    CU_ADD_TEST(pSuite, test_splitFractionRight2);
    CU_ADD_TEST(pSuite, test_splitFractionWrong);

    pSuite = CU_add_suite(“makeIntoFraction”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_makeIntoFractionCommonNormal);
    CU_ADD_TEST(pSuite, test_makeIntoFractionDecimal);
    CU_ADD_TEST(pSuite, test_makeIntoFractionNegative);
    CU_ADD_TEST(pSuite, test_makeIntoFractionCommonZeroNumerator);
    CU_ADD_TEST(pSuite, test_makeIntoFractionCommonZeroDenominator);
    CU_ADD_TEST(pSuite, test_makeIntoFractionWrong);

    pSuite = CU_add_suite(“calculateHighestPower”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_calculateHighestPowerNormal1);
    CU_ADD_TEST(pSuite, test_calculateHighestPowerNormal2);
    CU_ADD_TEST(pSuite, test_calculateHighestPowerNegative);

    pSuite = CU_add_suite(“calculateHighestNegativePower”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_calculateHighestNegativePowerNormal);
    CU_ADD_TEST(pSuite, test_calculateHighestNegativePowerBordered);
    CU_ADD_TEST(pSuite, test_calculateHighestNegativePowerNegative);

    pSuite = CU_add_suite(“checkOperation”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_checkOperationTrue);
    CU_ADD_TEST(pSuite, test_checkOperationFalse);

    pSuite = CU_add_suite(“findSimpleDividers”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_findSimpleDividersNormal);

    pSuite = CU_add_suite(“subtractArrays”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_subtractArraysNormal);
    CU_ADD_TEST(pSuite, test_subtractArraysZeroMinuend);
    CU_ADD_TEST(pSuite, test_subtractArraysZeroSubtrahend);

    pSuite = CU_add_suite(“findLCM”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_findLCM);

    pSuite = CU_add_suite(“findGCD”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_findGCD);

    pSuite = CU_add_suite(“reduceFraction”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_reduceFractionNormal1);
    CU_ADD_TEST(pSuite, test_reduceFractionNormal2);
    CU_ADD_TEST(pSuite, test_reduceFractionWrongType);

    pSuite = CU_add_suite(“toDoubleCommon”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_toDoubleCommonNormal);
    CU_ADD_TEST(pSuite, test_toDoubleCommonNegative);
    CU_ADD_TEST(pSuite, test_toDoubleCommonZero);

    pSuite = CU_add_suite(“toDoubleDecimal”, NULL, NULL);
```

```
    CU_ADD_TEST(pSuite, test_toDoubleDecimalNormal);
    CU_ADD_TEST(pSuite, test_toDoubleDecimalNegative);
    CU_ADD_TEST(pSuite, test_toDoubleDecimalZero);

    pSuite = CU_add_suite(“calculateCommon”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_calculateCommonPlus);
    CU_ADD_TEST(pSuite, test_calculateCommonMinus);
    CU_ADD_TEST(pSuite, test_calculateCommonMultiply);
    CU_ADD_TEST(pSuite, test_calculateCommonDivide);

    pSuite = CU_add_suite(“calculateDecimal”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_calculateDecimalPlus);
    CU_ADD_TEST(pSuite, test_calculateDecimalMinus);
    CU_ADD_TEST(pSuite, test_calculateDecimalMultiply);
    CU_ADD_TEST(pSuite, test_calculateDecimalDivide);;

    pSuite = CU_add_suite(“compareCommon”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_compareCommonLesser);
    CU_ADD_TEST(pSuite, test_compareCommonGreater);
    CU_ADD_TEST(pSuite, test_compareCommonEqualTrue);
    CU_ADD_TEST(pSuite, test_compareCommonEqualFalse);
    CU_ADD_TEST(pSuite, test_compareCommonNotEqualTrue);
    CU_ADD_TEST(pSuite, test_compareCommonNotEqualFalse);

    pSuite = CU_add_suite(“compareDecimal”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_compareDecimalLesser);
    CU_ADD_TEST(pSuite, test_compareDecimalGreater);
    CU_ADD_TEST(pSuite, test_compareDecimalEqualTrue);
    CU_ADD_TEST(pSuite, test_compareDecimalEqualFalse);
    CU_ADD_TEST(pSuite, test_compareDecimalNotEqualTrue);
    CU_ADD_TEST(pSuite, test_compareDecimalNotEqualFalse);

    pSuite = CU_add_suite(“calculateCompareBoth”, NULL, NULL);
    CU_ADD_TEST(pSuite, test_calculateBothType);
    CU_ADD_TEST(pSuite, test_compareBothType);

    CU_basic_run_tests();
    CU_cleanup_registry();
    return CU_get_error();
}
```

На листинге 3 представлен код дочернего процесса.

Листинг 3 – Дочерний процесс

```
#include "algorithm.c"

int main()
{
    userInput();
    return 0;
}
```

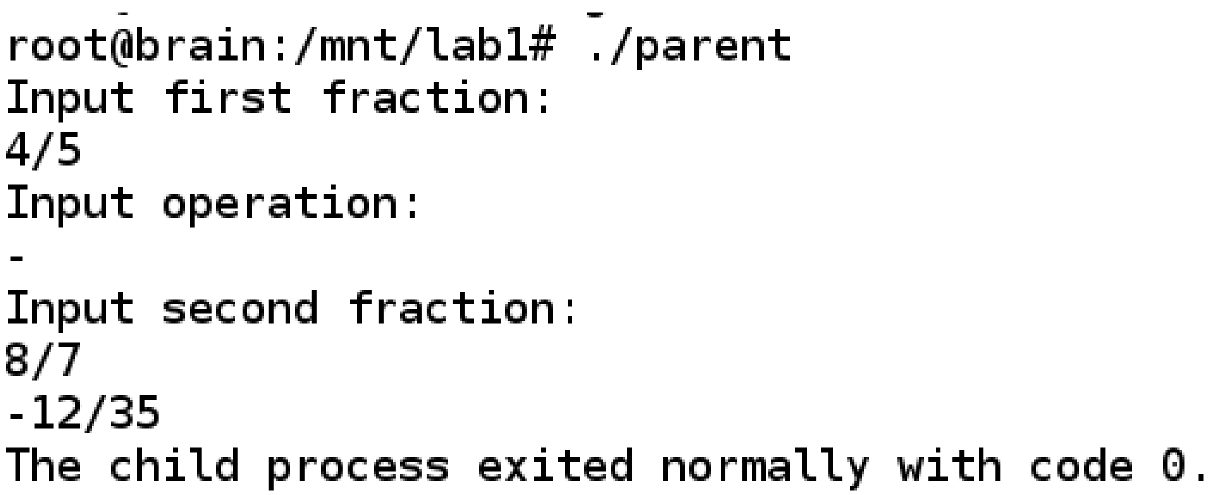
На листинге 4 представлен код родительского процесса.

Листинг 4 – Родительский процесс

```
/*! \file   parent.c
 *  \brief  Fork and execute child process
 *  \author Nikitin Alexander, KI19-17/1Б
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>

/*! \brief Spawns a child process
 *
 *  \param program Name of the compiled program in the directory
 *  \param argList Additional arguments to run the program
 *  \return Child process id
 */
int spawn(char* program, char** argList)
{
    pid_t childPid;
    childPid = fork();
    if (childPid != 0)
        return childPid;
    else
    {
        execvp(program, argList);
        abort();
    }
}
```

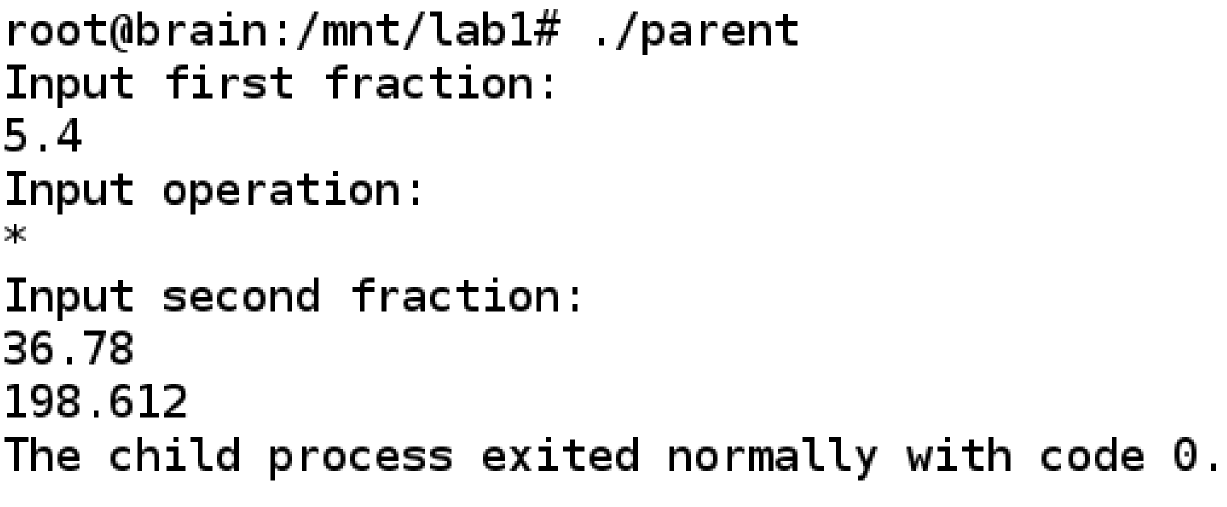
```
int main()
{
    char *args[]={"./child", NULL};

    int childStatus;
    spawn(args[0], args);
    wait(&childStatus);
    if(WIFEXITED(childStatus))
        printf("The child process exited normally with code %d.\n",
WEXITSTATUS(childStatus));
    else
        printf("The child process exited abnormally.\n");
    return 0;
}
```

4 Тестовые примеры работы программ



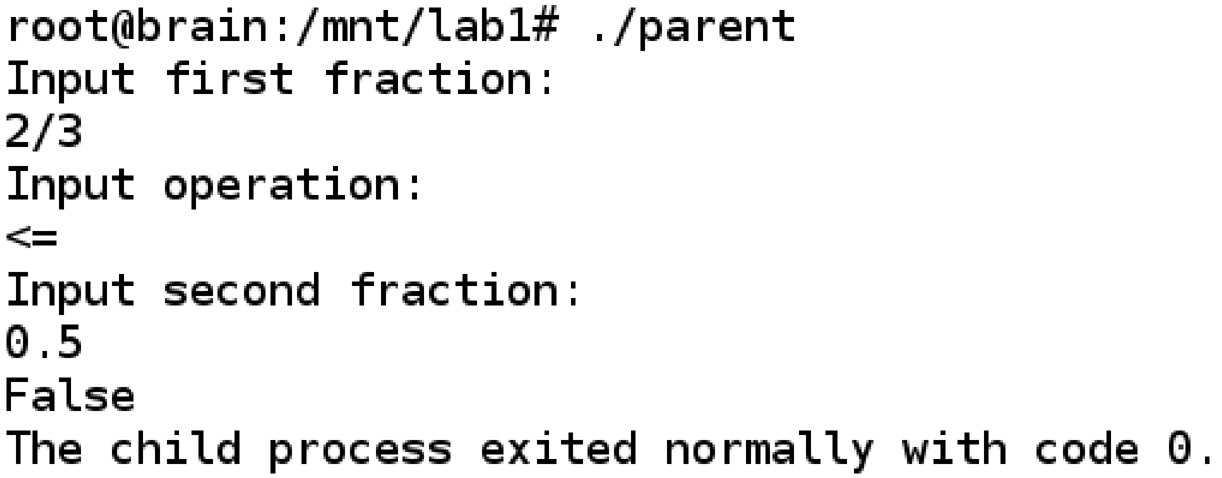
```
root@brain:/mnt/lab1# ./parent
Input first fraction:
4/5
Input operation:
-
Input second fraction:
8/7
-12/35
The child process exited normally with code 0.
```

Рисунок 1 – Пример вычитания двух обыкновенных дробей



```
root@brain:/mnt/lab1# ./parent
Input first fraction:
5.4
Input operation:
*
Input second fraction:
36.78
198.612
The child process exited normally with code 0.
```

Рисунок 2 – Пример умножения двух десятичных дробей



```
root@brain:/mnt/lab1# ./parent
Input first fraction:
2/3
Input operation:
<=
Input second fraction:
0.5
False
The child process exited normally with code 0.
```

Рисунок 3 – Пример сравнения обыкновенной и десятичной дроби