Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

# Лабораторная работа №2
# по курсу «Операционные системы»

| | |
|---|---|
| Студент: | Катермин В.С. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Миронов Е.С. |
| Оценка: | |
| Дата: | 22.11.2019 |

Москва
2019

**Содержание**

## 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.

Вариант 18: Дочерний процесс представляет собой сервер по работе со стеками и принимает команды со стороны родительского процесса."

## 2. Общие сведения о программе.

Исходный код хранится в файле main.c. В данном файле используются заголовочные файлы stdio.h, stdlib.h, string.h, sys/types.h, sys/wait.h, и unistd.h. В программе используются следующие системные вызовы:

1. pipe — создание канала для обмена данными между процессами. Системный вызов возвращает два дескриптора. Один для записи в канал, другой для чтения из канала.
2. fork — создание дочернего процесса.
3. read — чтение из потока в буфер некоторого количества байт.
4. write — запись в поток из буфера некоторого количества байт.
5. wait — ожидание завершения дочернего процесса.
6. close — закрытие потока.

Файлы stack.h и stack.c хранят структуру стека и функции для работы с ним (st_new, st_rm, push, top, pop).

## 3. Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Используя системный вызов pipe, создать два канала для общения между процессами.
2. С помощью системного вызова fork создать дочерний процесс, где создаётся стек.
3. В зависимости от введённых команд выполнятся определенное действие
    1. 0 — очистка стека;
    2. 1— добавить число в стек. В этом случае вводится число, которое будет добавлено в стек;
    3. 2 — метод pop (вывод последнего добавленного элемента и его удаление);
    4. 3 — вывод количества элементов в стеке;

5. 4 — метод top (вывод последнего добавленного элемента без его удаления);
4. Родительский процесс вступает в состояние ожидания завершения дочернего процесса.
5. Как только родительский процесс записал данные в первый канал, дочерний процесс считывает их, производит поиск образца в строке и записывает результат во второй канал.
6. Как только дочерний процесс завершился, родительский считывает результат из второго канала и выводит результат команды в стандартный поток вывода.

## 4. Основные файлы программы.

Файл main.c:

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

#include "stack.h"

enum {
   cmd_rm,
   cmd_push,
   cmd_pop,
   cmd_count,
   cmd_top
};

void help(void) {
    if (isatty(0) && isatty(1)) {
        printf("Commands: \n");
        printf("0 - delete all elements from the stack;\n");
        printf("1 - push a number into stack; \n");
        printf("2 - pop a number from stack; \n");
        printf("3 - show a number of items in stack; \n");
        printf("4 - show a top element in stack. \n");
    }
}

void bye(void) {
```

```
    if (isatty(0) && isatty(1))
        printf("\n");
}

void greet(void) {
    if (isatty(0) && isatty(1))
        printf(">>> ");
}

int parent(int ind, int outd) {
     int command;
   int value;
   int error;

   help();

     greet();
   while (scanf("%d", &command) == 1) {
        write(outd, &command, sizeof(command));

      if (command == cmd_push) {
            scanf("%d", &value);
            write(outd, &value, sizeof(value));
        }

      read(ind, &error, sizeof(error));
      read(ind, &value, sizeof(value));

      switch (command) {
            case cmd_top:
                if (!error)
                    printf("At top:\t%d\n", value);
                else
                    printf("Stack is empty\n");

                break;

            case cmd_pop:
                if (!error)
                    printf("Popped:\t%d\n", value);
                else
                    printf("Stack is empty\n");
```

```
                break;

            case cmd_push:
                if (!error)
                    printf("Pushed:\t%d\n", value);
                else
                    printf("Error occured\n");

                break;

            case cmd_count:
                printf("Count:\t%d\n", value);
                break;

            case cmd_rm:
                printf("Cleared\n");
                break;

            default:
                printf("Unknown command\n");
                break;
        }
        greet();
    }
     bye();

    close(outd);
    close(ind);

     wait(NULL);
}

int child(int ind, int outd) {
    int    c_num;
     stack *st = st_new();

     if (st) {
        while (read(ind, &c_num, sizeof(c_num)) > 0) {
            int value = 0;
            int error = 0;

            switch (c_num) {
                case cmd_rm:
```

```
                    st_rm(st);
                    break;

                case cmd_push:
                    read(ind, &value, sizeof(value));
                    error = push(st, value);
                    break;

                case cmd_pop:
                    error = pop(st, &value);
                    break;

                case cmd_count:
                    value = st->count;
                    break;

                case cmd_top:
                    error = top(st, &value);
                    break;

            }
            write(outd, &error, sizeof(error));
            write(outd, &value, sizeof(value));
        }
    }

    if (st) {
        st_rm(st);
        free (st);
    }

  close(ind);
  close(outd);

    return 0;
}

int main() {
  // We use two pipes
  // First pipe to send input string from parent
  // Second pipe to send concatenated string from child

  int fd1[2]; // Used to store two ends of first pipe
```

```
    int fd2[2]; // Used to store two ends of second pipe
    int er1 = pipe(fd1);
    int er2 = pipe(fd2);

    if (er1 == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }
    if (er2 == -1) {
        fprintf(stderr, "Pipe Failed");
        return 1;
    }

    pid_t p = fork();

    if (p < 0) {
        fprintf(stderr, "fork Failed");
        er1 = 1;
    } else if (p > 0) {
            close(fd1[0]);
            close(fd2[1]);
            er1 = parent(fd2[0], fd1[1]);
    } else {
            close(fd1[1]);
            close(fd2[0]);
        er1 = child(fd1[0], fd2[1]);
    }

    return er1;
}
```

Файл stack.h:

```
#ifndef D_STACK_H
#define D_STACK_H

typedef struct {
    size_t size;
    size_t count;
    int   *data;
} stack;

stack *st_new(void);
void   st_rm (stack       *st);
```

```
int    push  (stack       *st, int  val);
int    pop   (stack       *st, int *val);
int    top   (const stack *st, int *val);

#endif //D_STACK_H
```

Файл stack.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define MULTIPLIER 2

stack *st_new(void){
   stack *st = (stack *)malloc(sizeof(stack));

   if (st){
      st->size  = 4;
      st->count = 0;
      st->data  = (int *)malloc(sizeof(int) * st->size);
   }

   return st;
}

void st_rm(stack *st) {
     if (st) {
          if (st->data)
               free(st->data);

          st->count = 0;
          st->size  = 0;
          st->data  = NULL;
     }
}

int resize(stack *st) {
     int err = 0;

     if (st && st->count >= st->size) {
          int *tmp;
```

```
        if (!st->size)
            st->size++;

        tmp = (int *)realloc(st->data, st->size * sizeof(int) * MULTIPLIER);

        if (tmp) {
            st->data  = tmp;
            st->size *= MULTIPLIER;
        }
        else
            err = 1;
    }

    return err;
}

int push (stack *st, int val){
    int err = 0;

    if (st) {
        err = resize(st);

        if (!err) {
            st->data[st->count] = val;
            st->count++;
        }
    }

    return err;
}

int pop(stack *st, int *val) {
    int err = top(st, val);

    if (st && st->count)
        st->count--;

    return err;
}

int top(const stack *st, int *val) {
    int err = 0;
```

```
    if (st && st->count) {
        if (val)
            *val = st->data[st->count - 1];
    }
    else
        err = 1;

    return err;
}
```

## 5. Демонстрация работы программы.

```
user@PSB133S01ZFH:~/3sem_projects/os_lab2$ ./lab2
Commands:
0 - delete all elements from the stack;
1 - push a number into stack;
2 - pop a number from stack;
3 - show a number of items in stack;
4 - show a top element in stack.
>>> 1
50
Pushed:     50
>>> 1
20
Pushed:     20
>>> 1
40
Pushed:     40
>>> 4
At top: 40
>>> 3
Count: 3
>>> 2
Popped:     40
>>> 4
At top: 20
>>> 3
Count: 2
>>> 2
Popped:     20
>>> 4
At top: 50
```

```
>>> 3
Count: 1
>>> 2
Popped:     50
>>> 4
Stack is empty
>>> 3
Count: 0
>>> 2
Stack is empty
>>> 2
Stack is empty
>>> 0
Cleared
```

## 6. Вывод.

Процессы — это одна из самых старых и наиболее важных абстракций, присущих операционной системе. Они поддерживают возможность осуществления (псевдо) параллельных операций даже при наличии всего одного процессора. Они превращают один центральный процессор в несколько виртуальных. Без абстракции процессоров современные вычисления просто не могут существовать. Межпроцессное взаимодействие можно осуществлять с помощью канала. В системах UNIX канал создается с помощью системного вызова pipe. Я считаю, что такой подход к общению процессов удобен, поскольку при использовании блокирующих вызовов read и write процессы блокируются, если им нечего считывать или буфер для записи полный. Также одним из плюсов такого способа общения процессов является то, что каналом могут пользоваться только родственные процессы.