МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Системы параллельной обработки данных»

Тема: СОЗДАНИЕ МАСШТАБИРУЕМЫХ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Студент гр. 5304 _____ Лянгузов А.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

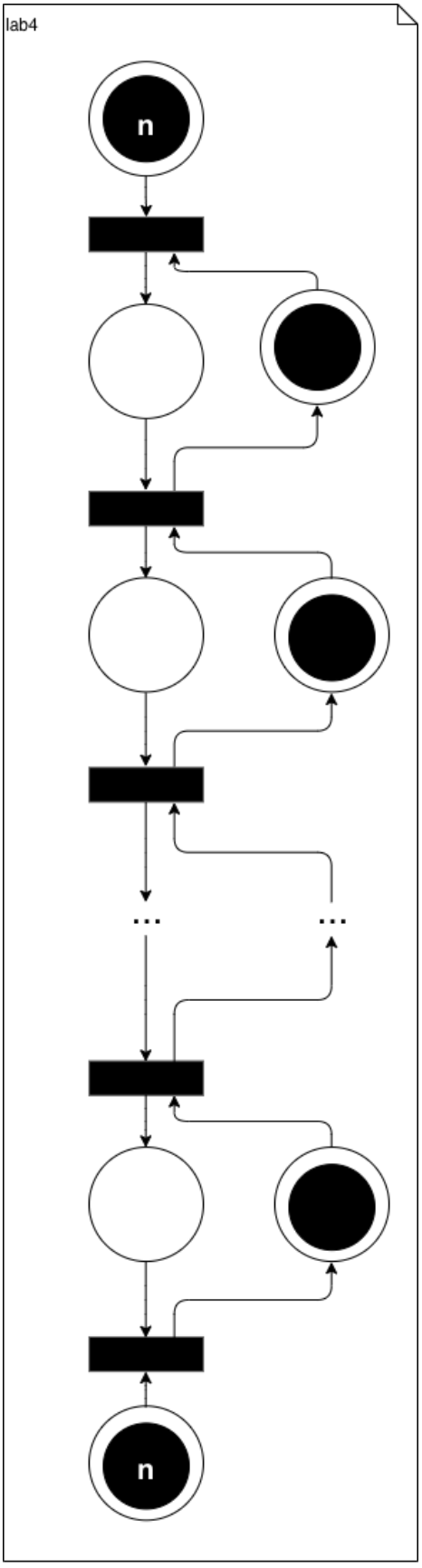
2019

Задание:

1. Написать масштабируемую программу, моделирующую буфер типа FIFO на N сообщений, где N – число запущенных параллельных процессов. Таким образом, каждый процесс должен моделировать один регистр (ячейку памяти) буфера FIFO, хранящую одно сообщение. Программа должна моделировать управляющие сигналы буфера FIFO: буфер пуст и буфер полон.

3. Предложить методику тестирования программы. Используя управляющие сигналы буфера FIFO (см. п.1), рассмотреть несколько режимов работы моделирующей программы: запись информации в пустой буфер до его заполнения (операция чтения не выполняется), чтение информации из полностью заполненного буфера (операция записи не выполняется), одновременная запись и чтение информации из буфера.

Сеть Петри для описания работы очереди.



Выполнение работы

За создание очереди отвечает нулевой процесс.

Ненулевые процессы ждут указаний от нулевого процесса. Они ловят тип операции и в соответствие с ним производят операцию над буфером памяти.

Рассмотрим пример одновременной записи и чтения.

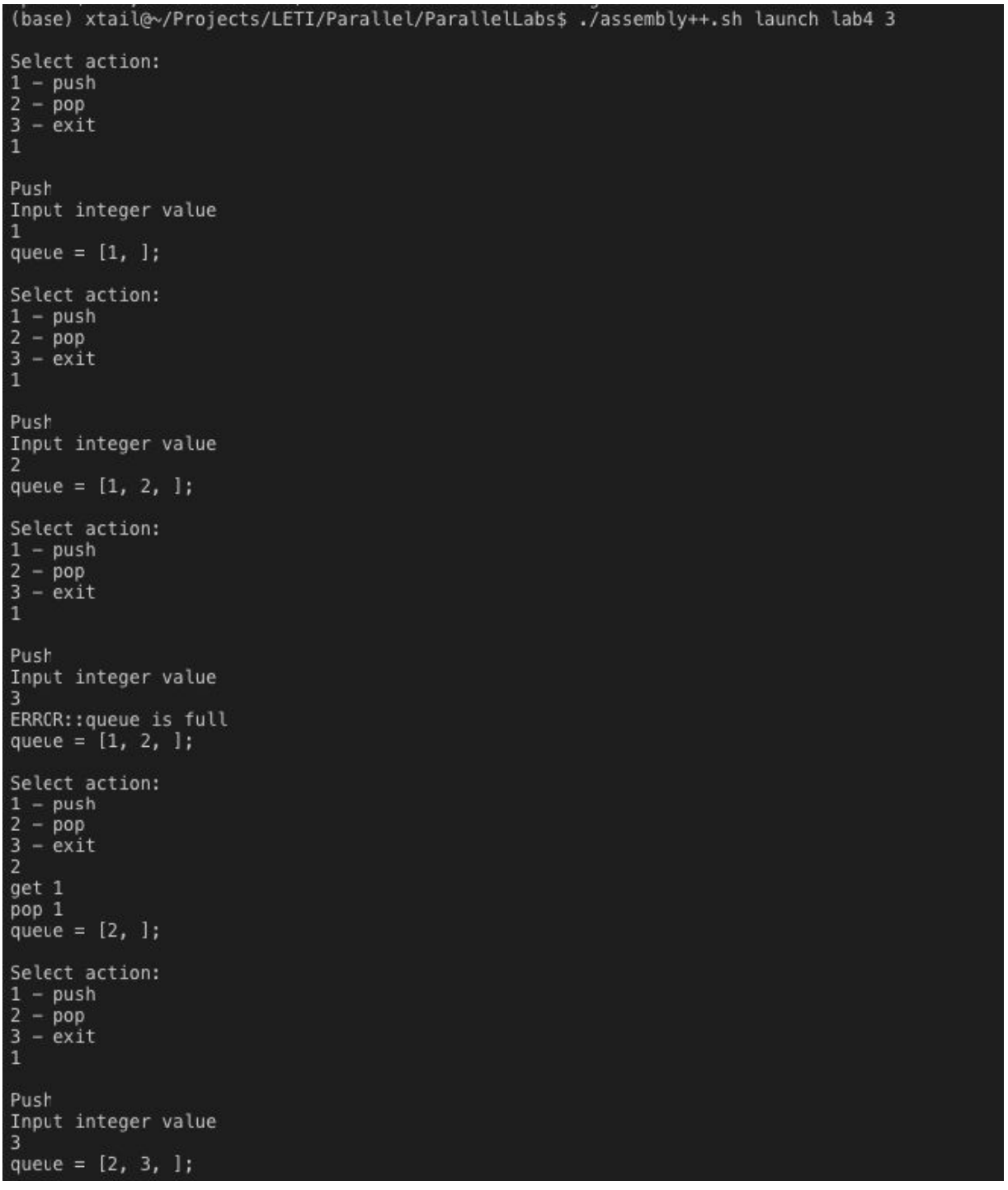


Рис.1. Результат работы программы

Выводы

В ходе выполнения лабораторной работы была написана масштабируемая программа, моделирующая буфер типа FIFO.

ПРИЛОЖЕНИЕ А.
КОД ПРОГРАММЫ

```
#include <iostream>
#include <array>
#include <vector>
#include <numeric>
#include <random>
#include <algorithm>

#include "mpi.h"

#define DataType MPI_INT
#define ControlType MPI_UNSIGNED

using namespace std;

//Id процесса с рангом 0
constexpr int ROOT_RANK = 0;

/*
* Идентификаторы сообщений для MPI_Send и MPI_Recv
*/
enum Tags
{
   CONTROL, // получена команда
   DATA, // получены данные
};



// Управляющие команды/сигналы для дочерних процессов, работающих с FIFO
enum Command
{
   PUSH, // Положить в буфер
   GET, // Получить из буфера
   POP, // Удалить из буфера
   STOP, // Остановить работу с буфером. Завершить работу цикла
};
```

```
using value_type = int;

/*
* Реализация FIFO по типу очереди
*/
class MPIQueue {
   private:
       unsigned m_start_index = 0; // С какого места начинаем писать в
очередь
       unsigned m_already_written = 0; // Размер уже записанного
       const unsigned m_size; // Размер FIFO. Зависит от количества
процессов или процессоров
   public:
       /*
       * Конструктор класса.
       * capacity - общее количество процессов.
       */
       MPIQueue(unsigned capacity) : m_size(capacity) {}

       /*
       * Производит запись данных.
       * Посылает сначала команду записи, затем данные для записи.
       */
       bool push(const value_type &val)
       {
           if (m_already_written == m_size)
               return false;

           unsigned pos = (m_start_index + m_already_written) % m_size;

           Command cmd(Command::PUSH);
           MPI_Send(&cmd, 1, ControlType, pos + 1, int(Tags::CONTROL),
MPI_COMM_WORLD);
           MPI_Send(&val, 1, DataType, pos + 1, int(Tags::DATA),
MPI_COMM_WORLD);

           ++m_already_written;
           return true;
```

```
        }

        /*
        * Получает данные из очереди.
        * val - переменная для хранения результата.
        */
        bool get(value_type &val) const
        {
            if (m_already_written == 0)
                return false;

            Command cmd(Command::GET);
            MPI_Send(&cmd, 1, ControlType, m_start_index + 1,
int(Tags::CONTROL), MPI_COMM_WORLD);
            MPI_Recv(&val, 1, DataType, m_start_index + 1, int(Tags::DATA),
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            return true;
        }

        /*
        * Удаляет данные из FIFO
        */
        bool pop()
        {
            if (m_already_written == 0)
                return false;

            Command cmd(Command::POP);
            MPI_Send(&cmd, 1, ControlType, m_start_index + 1,
int(Tags::CONTROL), MPI_COMM_WORLD);

            m_start_index = (m_start_index + 1) % m_size;
            --m_already_written;
            return true;
        }

        void print()
        {
```

```
            vector<value_type> currentState;

            int currentQueueSize = this->m_already_written;
            for(int i = 0; i < currentQueueSize; i++)
            {
                value_type item;
                if(!this->get(item))
                {
                    cout << "print queue error " << i << endl;
                    break;
                }

                this->pop();

                currentState.push_back(item);
            }

            cout << "queue = [";
            for(auto item : currentState)
            {
                cout << item << ", ";
                this->push(item);
            }
            cout << "];" << endl;
        }
};

// Основная функция для дочерних процессов
void slave(MPI_Status& status)
{
   value_type storage;

   // очередь полна
   bool full = false;

   // ожидание команды
   for (;;)
   {
```

```
        Command cmd;

        // ожидание сигнала на прерывание цикла
        if (MPI_Recv(&cmd, 1, ControlType, ROOT_RANK, int(Tags::CONTROL),
MPI_COMM_WORLD, &status))
        {
            return;
        }

        // обработка пришедшей команды
        switch (cmd)
        {
            // сохраняет данные в ячейку очереди
            case Command::PUSH:
                if (MPI_Recv(&storage, 1, DataType, ROOT_RANK,
int(Tags::DATA), MPI_COMM_WORLD, &status))
                    return;
                full = true;
                break;

            // возвращает данные из ячейки очереди
            case Command::GET:
                if (MPI_Send(&storage, 1, DataType, ROOT_RANK,
int(Tags::DATA), MPI_COMM_WORLD)) {
                    return;
                }
                break;

            // очищает очередь
            case Command::POP:
                full = false;
                break;

            // прерывает цикл выполенния
            case Command::STOP:
                return;
        }
    }
```

```
}

void printMainMenu()
{
   cout << endl << "Select action:" << endl;
   cout << "1 - push" << endl;
   cout << "2 - pop" << endl;
   cout << "3 - exit"<< endl;
}

value_type pushDialogResult()
{
   cout << endl << "Push" << endl;
   cout << "Input integer value" << endl;
   value_type value;
   cin >> value;
   return value;
}

// точка входа в программу
int main(int argc, char* argv[])
{
   int rank;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   int size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   if (rank == 0)
   {
       MPIQueue queue(size - 1);

       for(;;)
       {
           printMainMenu();
```

```
            int action;
            cin >> action;

            if(action == 1)
            {
                value_type value = pushDialogResult();
                if(!queue.push(value))
                {
                    cout << "ERROR::queue is full" << endl;
                }
                queue.print();
                continue;
            }

            if(action == 2)
            {
                value_type value;
                if(!queue.get(value))
                {
                    cout << "ERROR::queue is empty" << endl;
                }
                cout << "get " << value << endl;


                queue.pop();
                cout << "pop " << value << endl;

                queue.print();
                continue;
            }

            Command cmd(Command::STOP);
            for(int i = 1; i < size; ++i)
            {
                MPI_Send(&cmd, 1, ControlType, i, int(Tags::CONTROL),
MPI_COMM_WORLD);
            }
            break;
```

```
        }
    }
    else
    {
        slave(status);
    }

    MPI_Finalize();

    return 0;
}
```