**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение**
**высшего образования**
**«Московский государственный технический университет**
**имени Н.Э. Баумана**
**(национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

# Лабораторная работа № 5

**Тема**
Взаимодействие параллельных процессов

**Студент** Жигалкин Д.Р

**Группа** ИУ7-55Б

**Оценка (баллы)** _____

**Преподаватель** Рязанова Н.Ю

Москва.
2020 г.

## Задание на лабораторную работу:

1. Написать программу, реализующую задачу «Производство-потребление» по алгоритму Э. Дейкстры с тремя семафорами: двумя считающими и одним бинарным. В программе должно создаваться не менее 3х процессов - производителей и 3х процессов – потребителей. В программе надо обеспечить случайные задержки выполнения созданных процессов. В программе для взаимодействия производителей и потребителей буфер создается в разделяемом сегменте. Обратите внимание на то, чтобы не работать с одиночной переменной, а работать именно с буфером, состоящим их N ячеек по алгоритму. Производители в ячейки буфера записывают буквы алфавита по порядку. Потребители считывают символы из доступной ячейки. После считывания буквы из ячейки следующий потребитель может взять букву из следующей ячейки.

2. Написать программу, реализующую задачу «Читатели – писатели» по монитору Хоара с четырьмя функциями: Начать_чтение, Закончить_чтение, Начать_запись, Закончить_запись. В программе всеми процессами разделяется одно единственное значение в разделяемой памяти. Писатели ее только инкрементируют, читатели могут только читать значение.

Для реализации взаимоисключения используются семафоры

**Общее требование к обеим программам.**

В программах осуществляется консольный вывод, т.е. никакого интерфейса не нужно. Работающая программа должна выводить на экран какой процесс что записал, какой процесс что считал.

**Задача «Производство-потребление»**

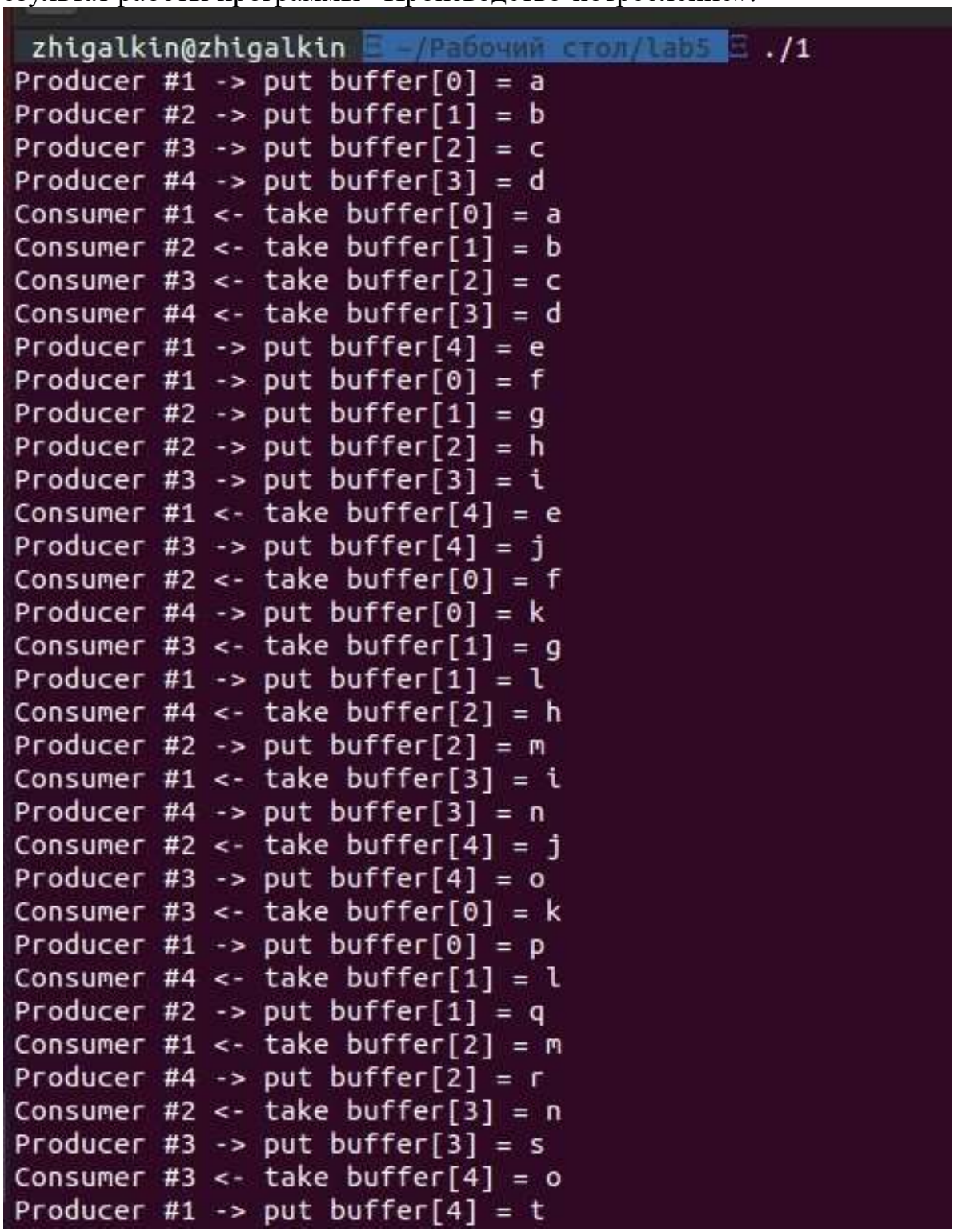
Листинг 1.1 — Программа, реализующая алгоритм задачи «Производство-потребление»

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <sys/wait.h>

#define COUNT_PRODUCERS 4
#define COUNT_CONSUMERS 4

#define BIN_SEM 0
#define BUFFER_EMPTY 1
#define BUFFER_FULL 2

#define BUF_SIZE 5

struct sembuf producer_P[] =
{
    {BUFFER_EMPTY, -1, 0},
    {BIN_SEM, -1, 0}
};

struct sembuf producer_V[] =
{
    {BIN_SEM, 1, 0},
    {BUFFER_FULL, 1, 0}
};

struct sembuf consumer_P[] =
{
    {BUFFER_FULL, -1, 0},
    {BIN_SEM, -1, 0}
};
struct sembuf consumer_V[] =
{
    {BIN_SEM, 1, 0},
    {BUFFER_EMPTY, 1, 0}
};

void Producer(int nomer, int sem_id, char* buf, int* pos, int* character)
{
    while (1)
    {
        // производит единичный объект

        // ждет, когда освободится хотя бы одна ячейка буфера
        // и когда или другой производитель,
        // или потребитель выйдет из критической секции
        semop(sem_id, producer_P, 2);

        // положить в буфер
        buf[*pos] = 'a' + *character;

        printf("Producer #%d -> ", nomer);
        printf("put buffer[%d] = %c\n", *pos, buf[*pos]);

        *pos = *pos == BUF_SIZE - 1 ? 0 : *pos + 1;
        *character = *character == 25 ? 0 : *character + 1;

        // освобождение критической секции и инкремент количества заполненных ячеек
        semop(sem_id, producer_V, 2);

        sleep(rand() % 2);
    }
}

void Consumer(int nomer, int sem_id, char* buf, int* pos)
{
```

```
70:     while (1)
71:     {
72:         // ждет, когда будет заполнена хотя бы одна ячейка буфера
73:         // и когда или потребитель,
74:         // или другой производитель выйдет из критической секции
75:         semop(sem_id, consumer_P, 2);
76:
77:         // взять из буфера
78:         printf("Consumer #%d <- ", nomer);
79:         printf("take buffer[%d] = %c\n", *pos, buf[*pos]);
80:
81:         *pos = *pos == BUF_SIZE - 1 ? 0 : *pos + 1;
82:
83:         // освобождение критической секции и инкремент количества пустых ячеек
84:         semop(sem_id, consumer_V, 2);
85:
86:         sleep(rand() % 4);
87:     }
88: }
89:
90: int main()
91: {
92:     int perms = S_IRWXU | S_IRWXG | S_IRWXO;
93:     int shm_id;
94:     int sem_id;
95:     char *mem_ptr = -1;
96:
97:     // возвращает идентификатор разделяемому сегменту памяти
98:     if ((shm_id = shmget(IPC_PRIVATE, BUF_SIZE * sizeof(char) + 3 * sizeof(int),
    IPC_CREAT | perms)) == -1)
99:     {
100:         perror("Unable to create a shared area.\n");
101:         return 1;
102:     }
103:
104:     // возвращает указатель на сегмент разделяемой памяти
105:     if ((mem_ptr = shmat(shm_id, NULL, 0)) == -1)
106:     {
107:         perror("Can't attach memory.\n");
108:         return 1;
109:     }
110:
111:     // создание набора семафоров
112:     if ((sem_id = semget(IPC_PRIVATE, 3, perms)) == -1)
113:     {
114:         perror("Can’t semget.\n");
115:         return 1;
116:     }
117:
118:     // изменение управляющих параметров набора семафоров
119:     semctl(sem_id, BIN_SEM, SETVAL, 1);
120:     semctl(sem_id, BUFFER_EMPTY, SETVAL, BUF_SIZE);
121:     semctl(sem_id, BUFFER_FULL, SETVAL, 0);
122:
123:     // создание процессов
124:     int count_processes = 0;
125:     pid_t pid;
126:
127:     int* producer_pos = mem_ptr + BUF_SIZE;
128:     int* character = producer_pos + 1;
129:     int* consumer_pos = producer_pos + 2;
130:
131:     *producer_pos = 0;
132:     *consumer_pos = 0;
133:     *character = 0;
134:
135:     for (int i = 0; i < COUNT_PRODUCERS; i++)
136:     {
137:         if ((pid = fork())== -1)
138:         {
```

```
139:             perror("Can’t fork.\n");
140:             return 1;
141:         }
142:
143:         if (!pid)
144:             Producer(i + 1, sem_id, mem_ptr, producer_pos, character);
145:         else
146:             count_processes++;
147:     }
148:
149:     for (int i = 0; i < COUNT_CONSUMERS; i++)
150:     {
151:         if ((pid = fork())== -1)
152:         {
153:             perror("Can’t fork.\n");
154:             return 1;
155:         }
156:
157:         if (!pid)
158:             Consumer(i + 1, sem_id, mem_ptr, consumer_pos);
159:         else
160:             count_processes++;
161:     }
162:
163:     // ожидание завершения процессов
164:     int status;
165:     for (int i = 0; i < count_processes; i++)
166:     {
167:         wait(&status);
168:         if (!WIFEXITED(status))
169:             printf("exit-error, code = %d\n", status);
170:     }
171:
172:     // освобождение ресурсов
173:     shmdt(mem_ptr);
174:     semctl(sem_id, 0, IPC_RMID);
175:     shmctl(shm_id, IPC_RMID, 0);
176:
177:     return 0;
178: }
```

Результат работы программы «Производство-потребление»:




**Задача «Читатели-писатели»**

Листинг 1.2 — Программа, реализующая алгоритм задачи «Читатели-писатели»

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <sys/wait.h>

#define COUNT_READERS 4
#define COUNT_WRITERS 4

#define ACTIVE_READERS 0
#define ACTIVE_WRITER 1
#define WAITING_WRITERS 2
#define WAITING_READERS 3


struct sembuf start_writer[] =
{
    {WAITING_WRITERS,   1, 0},
    {ACTIVE_READERS,    0, 0},
    {ACTIVE_WRITER,     0, 0},
    {ACTIVE_WRITER,     1, 0},
    {WAITING_WRITERS,  -1, 0},
};

struct sembuf stop_writer[] =
{
    {ACTIVE_WRITER, -1, 0}
};

void start_write(int sem_id)
{
    semop(sem_id, start_writer, 5);
}

void stop_write(int sem_id)
{
    semop(sem_id, stop_writer, 1);
}

struct sembuf start_reader[] =
{
    {WAITING_READERS,   1, 0},
    {WAITING_WRITERS,   0, 0},
    {ACTIVE_WRITER,     0, 0},
    {ACTIVE_READERS,    1, 0},
    {WAITING_READERS,  -1, 0},
};

struct sembuf stop_reader[] =
{
    {ACTIVE_READERS, -1, 0}
};

void start_read(int sem_id)
{
    semop(sem_id, start_reader, 5);
}

void stop_read(int sem_id)
{
    semop(sem_id, stop_reader, 1);
}

void Reader(int nomer, int sem_id, int* buf)
{
    while (1)
    {
```

```
70:          start_read(sem_id);
71:
72:          printf("Reader #%d -> ", nomer);
73:          printf("read %d\n", *buf);
74:
75:          stop_read(sem_id);
76:
77:          sleep(rand() % 2);
78:      }
79:  }
80:
81:  void Writer(int nomer, int sem_id, int* buf)
82:  {
83:      while (1)
84:      {
85:          start_write(sem_id);
86:
87:          printf("Writer #%d -> ", nomer);
88:          printf("write %d\n", ++(*buf));
89:
90:          stop_write(sem_id);
91:
92:          sleep(rand() % 3);
93:      }
94:  }
95:
96:  int main()
97:  {
98:      int perms = S_IRWXU | S_IRWXG | S_IRWXO;
99:      int shm_id;
100:     int sem_id;
101:     int *mem_ptr;
102:
103:     // возвращает идентификатор разделяемому сегменту памяти
104:     if ((shm_id = shmget(IPC_PRIVATE, sizeof(int), IPC_CREAT | perms)) == -1)
105:     {
106:         perror("Unable to create a shared area.\n");
107:         return 1;
108:     }
109:
110:     // возвращает указатель на сегмент разделяемой памяти
111:     if ((mem_ptr = shmat(shm_id, NULL, 0)) == -1)
112:     {
113:         perror("Can't attach memory.\n");
114:         return 1;
115:     }
116:
117:     // создание набора семафоров
118:     if ((sem_id = semget(IPC_PRIVATE, 4, perms)) == -1)
119:     {
120:         perror("Can’t semget.\n");
121:         return 1;
122:     }
123:
124:     // изменение управляющих параметров набора семафоров
125:     semctl(sem_id, ACTIVE_READERS,  SETVAL, 0);
126:     semctl(sem_id, ACTIVE_WRITER,   SETVAL, 0);
127:     semctl(sem_id, WAITING_READERS, SETVAL, 0);
128:     semctl(sem_id, WAITING_WRITERS, SETVAL, 0);
129:
130:     // создание процессов
131:     int count_processes = 0;
132:     pid_t pid;
133:
134:     *mem_ptr = 0;
135:
136:     for (int i = 0; i < COUNT_WRITERS; i++)
137:     {
138:         if ((pid = fork())== -1)
139:         {
```

```
            perror("Can’t fork.\n");
            return 1;
        }

        if (!pid)
            Writer(i + 1, sem_id, mem_ptr);
        else
            count_processes++;
    }

    for (int i = 0; i < COUNT_READERS; i++)
    {
        if ((pid = fork())== -1)
        {
            perror("Can’t fork.\n");
            return 1;
        }

        if (!pid)
            Reader(i + 1, sem_id, mem_ptr);
        else
            count_processes++;
    }

    // ожидание завершение процессов
    int status;
    for (int i = 0; i < count_processes; i++)
    {
        wait(&status);
        if (!WIFEXITED(status))
            printf("exit-error, code = %d\n", status);
    }

    // освобождение ресурсов
    shmdt(mem_ptr);
    semctl(sem_id, 0, IPC_RMID);
    shmctl(shm_id, IPC_RMID, 0);

    return 0;
}
```

Результат работы программы «Читатели-писатели»:

```
zhigalkin@zhigalkin ~/Рабочий стол/lab5
Writer #1 -> write 1
Writer #2 -> write 2
Writer #4 -> write 3
Writer #3 -> write 4
Reader #1 -> read 4
Reader #4 -> read 4
Reader #2 -> read 4
Reader #3 -> read 4
Writer #1 -> write 5
Writer #2 -> write 6
Writer #4 -> write 7
Reader #1 -> read 7
Reader #1 -> read 7
Reader #4 -> read 7
Reader #4 -> read 7
Reader #2 -> read 7
Reader #2 -> read 7
Writer #3 -> write 8
Reader #3 -> read 8
Reader #3 -> read 8
Writer #1 -> write 9
Writer #1 -> write 10
Writer #2 -> write 11
Writer #2 -> write 12
Writer #4 -> write 13
Writer #4 -> write 14
Reader #1 -> read 14
Reader #4 -> read 14
Reader #2 -> read 14
Writer #3 -> write 15
Writer #3 -> write 16
Reader #3 -> read 16
Writer #1 -> write 17
Writer #2 -> write 18
Writer #4 -> write 19
Reader #1 -> read 19
Reader #4 -> read 19
Reader #2 -> read 19
Writer #3 -> write 20
Reader #3 -> read 20
```