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»

Отчет по лабораторной работе № 5 по курсу «Операционные системы»

Тема “Взаимодействие параллельных процессов”

Студент Садулаева Т. Р.

Группа ИУ7-54Б

Оценка (баллы) _______________

Преподаватель Рязанова Н. Ю.

Москва.
2020 г.

Задание 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.

Исходный код:

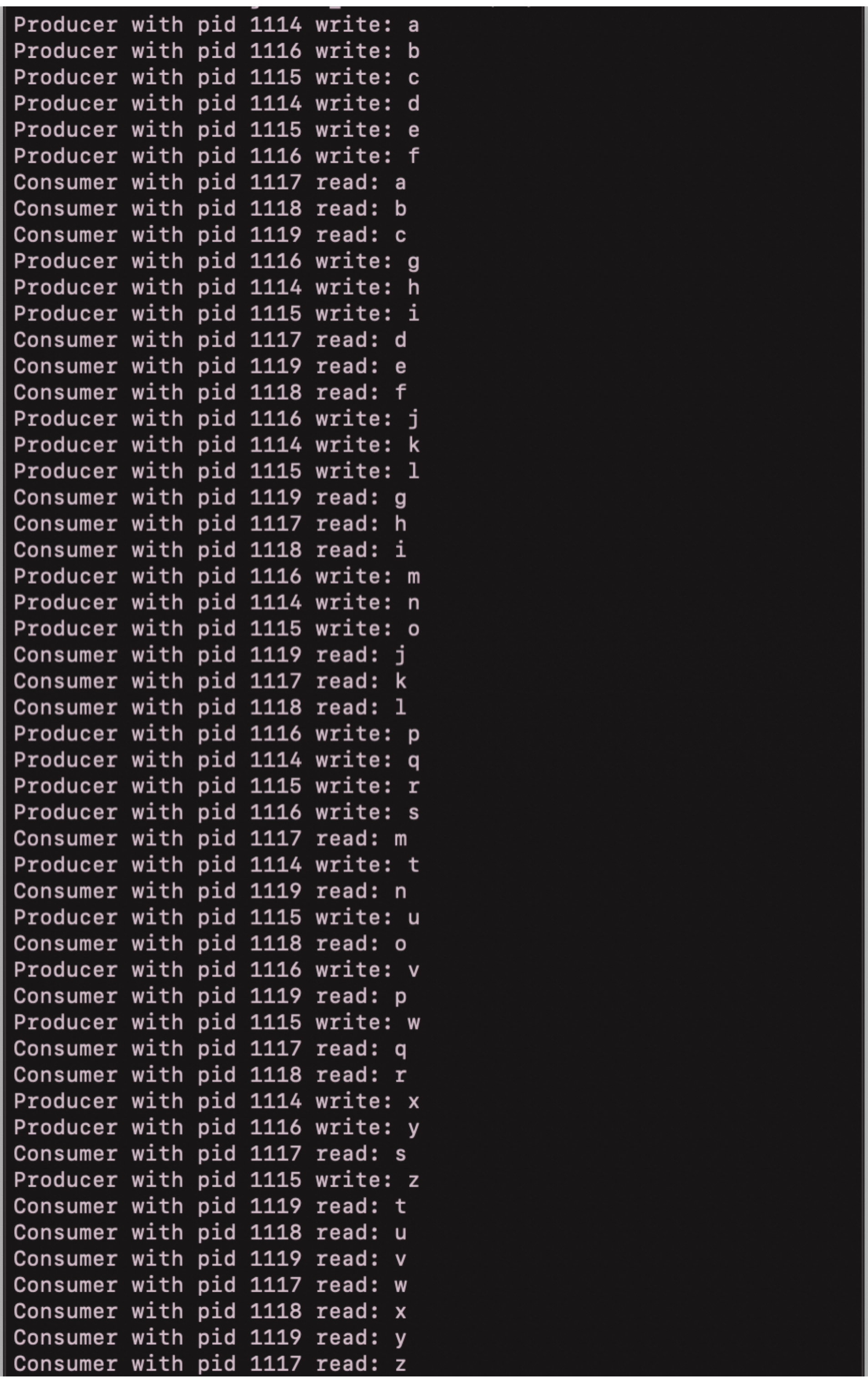
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <sys/sem.h>

#define SB 0
#define SE 1
#define SF 2
#define DECR -1
#define INCR 1
#define ALPHABET_LENGTH 26
#define GET_RANDOM_DELAY_PRODUCER (random() % 2) + 1
#define GET_RANDOM_DELAY_CONSUMER (random() % 5) + 1

#define OPERATIONS_AMOUNT 2
#define BUFFER_SIZE 7


typedef struct shared_struct {
   int producer_index;
   int consumer_index;
   char buffer[BUFFER_SIZE];
   char letter_to_write;
   int letters_read;
} shared_state_t;
shared_state_t* shared_state = NULL;
```

```
struct sembuf producer_start[2] =
{{SE, DECR, 0},
 {SB, DECR, 0}};

struct sembuf producer_stop[2] =
{{SB, INCR, 0},
 {SF, INCR, 0}};


struct sembuf consumer_start[2] =
{{SF, DECR, 0},
 {SB, DECR, 0}};

struct sembuf consumer_stop[2] =
{{SB, INCR, 0},
 {SE, INCR, 0}};

#define FLAGS S_IRWXU | S_IRWXG | S_IRWXO

void producer(int semid, pid_t pid) {
   int isOver = 0;
   while (!isOver) {
       sleep(GET_RANDOM_DELAY_PRODUCER);
       if (semop(semid, producer_start, OPERATIONS_AMOUNT) == -1) {
           fprintf(stderr, "producer::semop::producer_start error!\n");
           exit(EXIT_FAILURE);
       }

       if (shared_state->letter_to_write <= 'z') {
           shared_state->buffer[shared_state->producer_index % BUFFER_SIZE]
= shared_state->letter_to_write;
           printf("Producer with pid %d write: %c\n", pid,
                  shared_state->buffer[shared_state->producer_index %
BUFFER_SIZE]);
           shared_state->producer_index++;
           shared_state->letter_to_write++;
       } else { isOver++; }

       if (semop(semid, producer_stop, OPERATIONS_AMOUNT) == -1) {
           fprintf(stderr, "producer::semop::producer_stop error!\n");
           exit(EXIT_FAILURE);
       }
   }
}
```

```
void consumer(int semid, pid_t pid) {
   int isOver = 0;
   while (!isOver) {
       sleep(GET_RANDOM_DELAY_CONSUMER);
       if (semop(semid, consumer_start, OPERATIONS_AMOUNT) == -1) {
           fprintf(stderr, "producer::consumer::consumer_start error!\n");
           exit(EXIT_FAILURE);
       }

       if (shared_state->letters_read < ALPHABET_LENGTH) {
           printf("Consumer with pid %d read: %c\n", pid,
                  shared_state->buffer[shared_state->consumer_index %
BUFFER_SIZE]);
           shared_state->consumer_index++;
           shared_state->letters_read++;
       } else { isOver++; }

       if (semop(semid, consumer_stop, OPERATIONS_AMOUNT) == -1) {
           fprintf(stderr, "producer::consumer::consumer_stop error!\n");
           exit(EXIT_FAILURE);
       }
   }
}

int main() {
   int shm_id;
   if ((shm_id = shmget(IPC_PRIVATE, sizeof(shared_state_t),
                        IPC_CREAT | FLAGS)) == -1) {
       fprintf(stderr, "Main::shmget buffer error!");
       return EXIT_FAILURE;
   }

   shared_state = shmat(shm_id, 0, 0);

   if (shared_state == (void*)-1) {
       fprintf(stderr, "Main::shmat error!");
       return EXIT_FAILURE;
   }

   shared_state->producer_index = 0;
   shared_state->consumer_index = 0;
   shared_state->letter_to_write = 'a';
   shared_state->letters_read = 0;
```

```
int sem_id;
if ((sem_id = semget(IPC_PRIVATE, 3, IPC_CREAT | FLAGS)) == -1) {
    fprintf(stderr, "Main::semget semaphores error!");
    return EXIT_FAILURE;
}

int ctrl_sb = semctl(sem_id, SB, SETVAL, 1);
if (ctrl_sb == -1) {
    fprintf(stderr, "Main::semctl ctrl_sb semaphores error!");
    return EXIT_FAILURE;
}

int ctrl_se = semctl(sem_id, SE, SETVAL, BUFFER_SIZE);
if (ctrl_se == -1) {
    fprintf(stderr, "Main::semctl ctrl_se semaphores error!");
    return EXIT_FAILURE;
}

int ctrl_sf = semctl(sem_id, SF, SETVAL, 0);
if (ctrl_sf == -1) {
    fprintf(stderr, "Main::semctl ctrl_sf semaphores error!");
    return EXIT_FAILURE;
}

pid_t pid1, pid2, pid3, pid4, pid5, pid6;

if ((pid1 = fork()) == 0) {
    producer(sem_id, getpid());
    return 0;
}

if ((pid2 = fork()) == 0) {
    producer(sem_id, getpid());
    return 0;
}

if ((pid3 = fork()) == 0) {
    producer(sem_id, getpid());
    return 0;
}

if ((pid4 = fork()) == 0) {
    consumer(sem_id, getpid());
    return 0;
    }

    if ((pid5 = fork()) == 0) {
        consumer(sem_id, getpid());
        return 0;
    }

    if ((pid6 = fork()) == 0) {
        consumer(sem_id, getpid());
        return 0;
    }

    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    waitpid(pid3, NULL, 0);
    waitpid(pid4, NULL, 0);
    waitpid(pid5, NULL, 0);
    waitpid(pid6, NULL, 0);

    if (shmdt(shared_state) == -1) {
        fprintf(stderr, "Memory error!\n");
        return EXIT_FAILURE;
    }

    return 0;
}
```

Пример работы программы:



```
Producer with pid 1114 write: a
Producer with pid 1116 write: b
Producer with pid 1115 write: c
Producer with pid 1114 write: d
Producer with pid 1115 write: e
Producer with pid 1116 write: f
Consumer with pid 1117 read: a
Consumer with pid 1118 read: b
Consumer with pid 1119 read: c
Producer with pid 1116 write: g
Producer with pid 1114 write: h
Producer with pid 1115 write: i
Consumer with pid 1117 read: d
Consumer with pid 1119 read: e
Consumer with pid 1118 read: f
Producer with pid 1116 write: j
Producer with pid 1114 write: k
Producer with pid 1115 write: l
Consumer with pid 1119 read: g
Consumer with pid 1117 read: h
Consumer with pid 1118 read: i
Producer with pid 1116 write: m
Producer with pid 1114 write: n
Producer with pid 1115 write: o
Consumer with pid 1119 read: j
Consumer with pid 1117 read: k
Consumer with pid 1118 read: l
Producer with pid 1116 write: p
Producer with pid 1114 write: q
Producer with pid 1115 write: r
Producer with pid 1116 write: s
Consumer with pid 1117 read: m
Producer with pid 1114 write: t
Consumer with pid 1119 read: n
Producer with pid 1115 write: u
Consumer with pid 1118 read: o
Producer with pid 1116 write: v
Consumer with pid 1119 read: p
Producer with pid 1115 write: w
Consumer with pid 1117 read: q
Consumer with pid 1118 read: r
Producer with pid 1114 write: x
Producer with pid 1116 write: y
Consumer with pid 1117 read: s
Producer with pid 1115 write: z
Consumer with pid 1119 read: t
Consumer with pid 1118 read: u
Consumer with pid 1119 read: v
Consumer with pid 1117 read: w
Consumer with pid 1118 read: x
Consumer with pid 1119 read: y
Consumer with pid 1117 read: z
```

Задание 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.

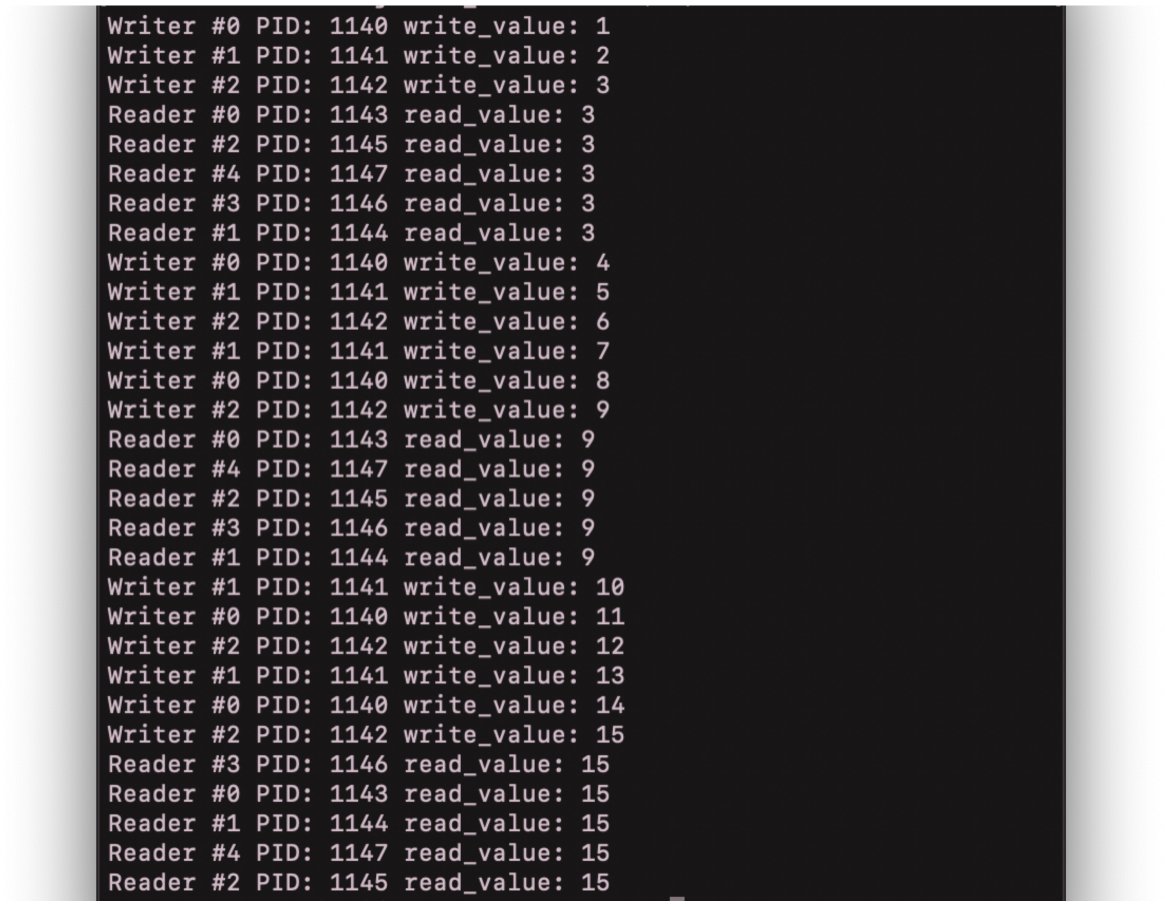
Исходный код:

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

#define INCR 1
#define DECR (-1)
#define CHECK 0

#define MAX_NUMBER 15

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

#define GET_RANDOM_DELAY_WRITER (rand() % 3)
#define GET_RANDOM_DELAY_READER (rand() % 5)
#define FLAGS (S_IRWXG | S_IRWXU | S_IRWXO)


struct sembuf start_write[] = {{WAITING_WRITERS, INCR, 0},
                               {ACTIVE_READERS, CHECK, 0},
                               {ACTIVE_WRITER, CHECK, 0},
                               {ACTIVE_WRITER, INCR, 0},
                               {WAITING_WRITERS, DECR, 0}};
#define START_WRITE_SEM_AMOUNT 5

struct sembuf stop_write[] = {{ACTIVE_WRITER, DECR, 0}};
```

```
#define STOP_WRITE_SEM_AMOUNT 1

void writer(int semid, int* shm, int num) {
    int isOver = 0;
    while (!isOver) {
        sleep(GET_RANDOM_DELAY_WRITER);

        if (semop(semid, start_write, START_WRITE_SEM_AMOUNT)) {
            fprintf(stderr, "writer::semop::start_write error!\n");
            exit(1);
        }

        if (*shm < MAX_NUMBER) {
            (*shm)++;
            printf("Writer #%d PID: %d write_value: %d\n", num, getpid(),
*shm);
        } else { isOver++; }

        if (semop(semid, stop_write, STOP_WRITE_SEM_AMOUNT)) {
            fprintf(stderr, "writer::semop::stop_write error!\n");
            exit(1);
        }
    }
}


struct sembuf start_read[] = {{WAITING_READERS, INCR, 0},
                              {ACTIVE_WRITER, CHECK, 0},
                              {WAITING_WRITERS, CHECK, 0},
                              {WAITING_READERS, DECR, 0},
                              {ACTIVE_READERS, INCR, 0}};
#define START_READ_SEM_AMOUNT 5

struct sembuf stop_read[] = {{ACTIVE_READERS, DECR, 0}};
#define STOP_READ_SEM_AMOUNT 1

void reader(int semid, int* shm, int num) {
    int isOver = 0;
    while (!isOver) {
        sleep(GET_RANDOM_DELAY_READER);

        if (semop(semid, start_read, START_READ_SEM_AMOUNT)) {
            fprintf(stderr, "reader::semop::start_read error!\n");
            exit (EXIT_FAILURE);
        }
        printf("Reader #%d PID: %d read_value: %d\n", num, getpid(), *shm);
        if (*shm == MAX_NUMBER) { isOver++; }

        if (semop(semid, stop_read, STOP_READ_SEM_AMOUNT)) {
            fprintf(stderr, "reader::semop::stop_read error!\n");
            exit (EXIT_FAILURE);
        }
    }
}

int main() {
    int shm_id;
    if ((shm_id = shmget(IPC_PRIVATE, 4, IPC_CREAT | FLAGS)) == -1) {
        fprintf(stderr, "main::shmget::shm_id error!\n");
        exit (EXIT_FAILURE);
    }

    int* shared_buffer = (shmat(shm_id, NULL, 0));
    if (shared_buffer == (void*)-1) {
        fprintf(stderr, "main::shmat error!\n");
        exit (EXIT_FAILURE);
    }

    (*shared_buffer) = 0;

    int sem_id;
    if ((sem_id = semget(IPC_PRIVATE, 4, IPC_CREAT | FLAGS)) == -1) {
        fprintf(stderr, "main::shmget::sem_id error!\n");
        exit (EXIT_FAILURE);
    }

    int ctrl = semctl(sem_id, ACTIVE_WRITER, SETVAL, 0);
    if (ctrl == -1) {
        fprintf(stderr, "main::shmget::semctl error!\n");
        exit (EXIT_FAILURE);
    }

    pid_t pid1, pid2, pid3, pid4, pid5, pid6, pid7, pid8;

    if ((pid1 = fork()) == 0) {
        writer(sem_id, shared_buffer, 0);
        return 0;
    }
```

```
if ((pid2 = fork()) == 0) {
    writer(sem_id, shared_buffer, 1);
    return 0;
}

if ((pid3 = fork()) == 0) {
    writer(sem_id, shared_buffer, 2);
    return 0;
}

if ((pid4 = fork()) == 0) {
    reader(sem_id, shared_buffer, 0);
    return 0;
}

if ((pid5 = fork()) == 0) {
    reader(sem_id, shared_buffer, 1);
    return 0;
}

if ((pid6 = fork()) == 0) {
    reader(sem_id, shared_buffer, 2);
    return 0;
}

if ((pid7 = fork()) == 0) {
    reader(sem_id, shared_buffer, 3);
    return 0;
}

if ((pid8 = fork()) == 0) {
    reader(sem_id, shared_buffer, 4);
    return 0;
}

waitpid(pid1, NULL, 0);
waitpid(pid2, NULL, 0);
waitpid(pid3, NULL, 0);
waitpid(pid4, NULL, 0);
waitpid(pid5, NULL, 0);
waitpid(pid6, NULL, 0);
waitpid(pid7, NULL, 0);
waitpid(pid8, NULL, 0);

if (shmdt(shared_buffer) == -1) {
        fprintf(stderr, "Memory error!\n");
        return EXIT_FAILURE;
    }

    return 0;
}
```

Пример работы программы:



```
Writer #0 PID: 1140 write_value: 1
Writer #1 PID: 1141 write_value: 2
Writer #2 PID: 1142 write_value: 3
Reader #0 PID: 1143 read_value: 3
Reader #2 PID: 1145 read_value: 3
Reader #4 PID: 1147 read_value: 3
Reader #3 PID: 1146 read_value: 3
Reader #1 PID: 1144 read_value: 3
Writer #0 PID: 1140 write_value: 4
Writer #1 PID: 1141 write_value: 5
Writer #2 PID: 1142 write_value: 6
Writer #1 PID: 1141 write_value: 7
Writer #0 PID: 1140 write_value: 8
Writer #2 PID: 1142 write_value: 9
Reader #0 PID: 1143 read_value: 9
Reader #4 PID: 1147 read_value: 9
Reader #2 PID: 1145 read_value: 9
Reader #3 PID: 1146 read_value: 9
Reader #1 PID: 1144 read_value: 9
Writer #1 PID: 1141 write_value: 10
Writer #0 PID: 1140 write_value: 11
Writer #2 PID: 1142 write_value: 12
Writer #1 PID: 1141 write_value: 13
Writer #0 PID: 1140 write_value: 14
Writer #2 PID: 1142 write_value: 15
Reader #3 PID: 1146 read_value: 15
Reader #0 PID: 1143 read_value: 15
Reader #1 PID: 1144 read_value: 15
Reader #4 PID: 1147 read_value: 15
Reader #2 PID: 1145 read_value: 15
```