|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **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** Блохин Д. М.  **Группа** ИУ7-52Б  **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 **SEMAPHORE\_BIN** 0

#define **SEMAPHORE\_EMPTY** 1

#define **SEMAPHORE\_FULL** 2

#define **DECREMENT** -1

#define **INCREMENT** 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] =

{{**SEMAPHORE\_EMPTY**, **DECREMENT**, 0},

{**SEMAPHORE\_BIN**, **DECREMENT**, 0}};

struct sembuf producer\_stop[2] =

{{**SEMAPHORE\_BIN**, **INCREMENT**, 0},

{**SEMAPHORE\_FULL**, **INCREMENT**, 0}};

struct sembuf consumer\_start[2] =

{{**SEMAPHORE\_FULL**, **DECREMENT**, 0},

{**SEMAPHORE\_BIN**, **DECREMENT**, 0}};

struct sembuf consumer\_stop[2] =

{{**SEMAPHORE\_BIN**, **INCREMENT**, 0},

{**SEMAPHORE\_EMPTY**, **INCREMENT**, 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, **SEMAPHORE\_BIN**, **SETVAL**, 1);

if (ctrl\_sb == -1) {

fprintf(**stderr**, "Main::semctl ctrl\_sb semaphores error!");

return **EXIT\_FAILURE**;

}

int ctrl\_se = semctl(sem\_id, **SEMAPHORE\_EMPTY**, **SETVAL**, **BUFFER\_SIZE**);

if (ctrl\_se == -1) {

fprintf(**stderr**, "Main::semctl ctrl\_se semaphores error!");

return **EXIT\_FAILURE**;

}

int ctrl\_sf = semctl(sem\_id, **SEMAPHORE\_FULL**, **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");

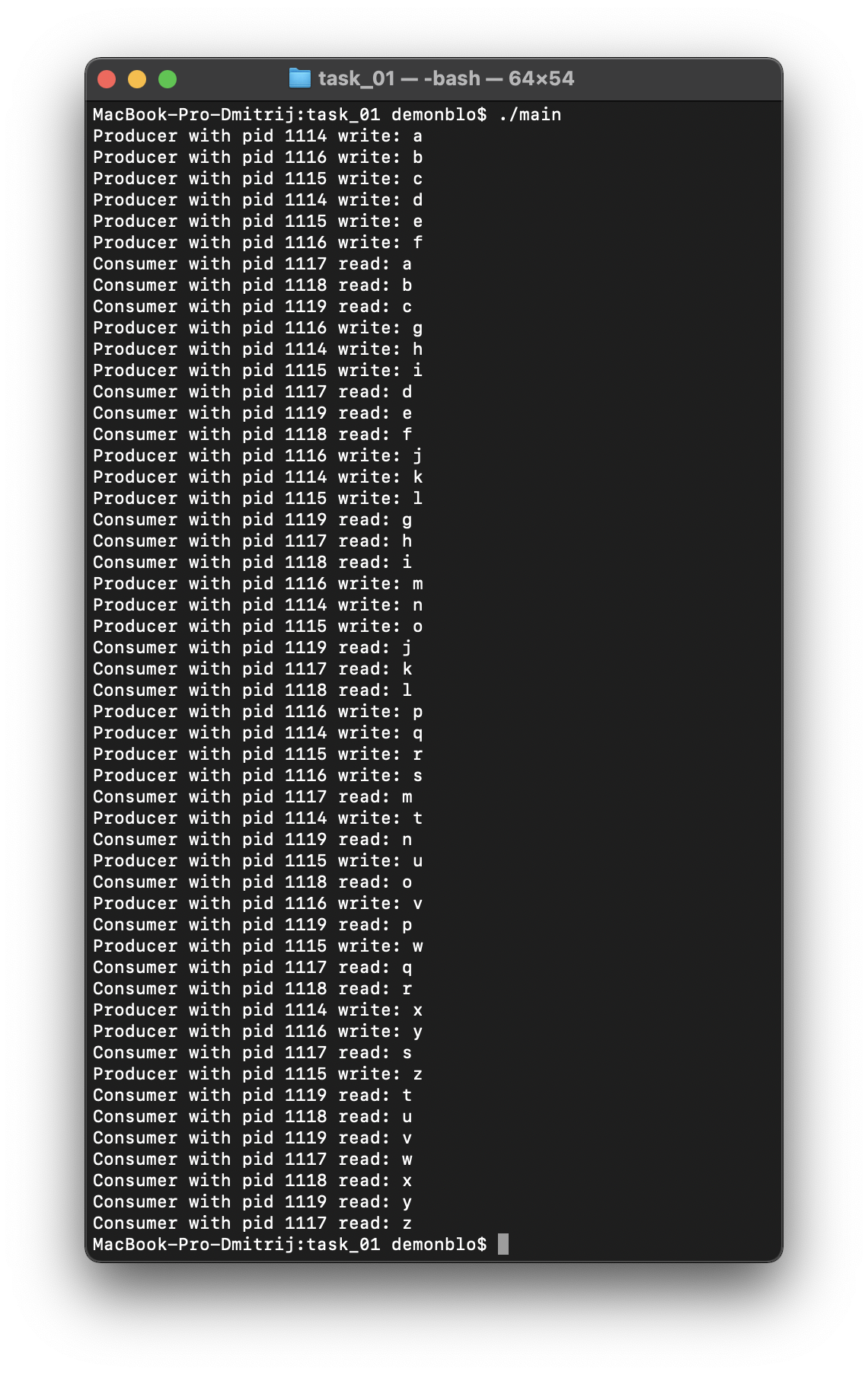
return **EXIT\_FAILURE**;

}

return 0;

}

**Пример работы программы**:



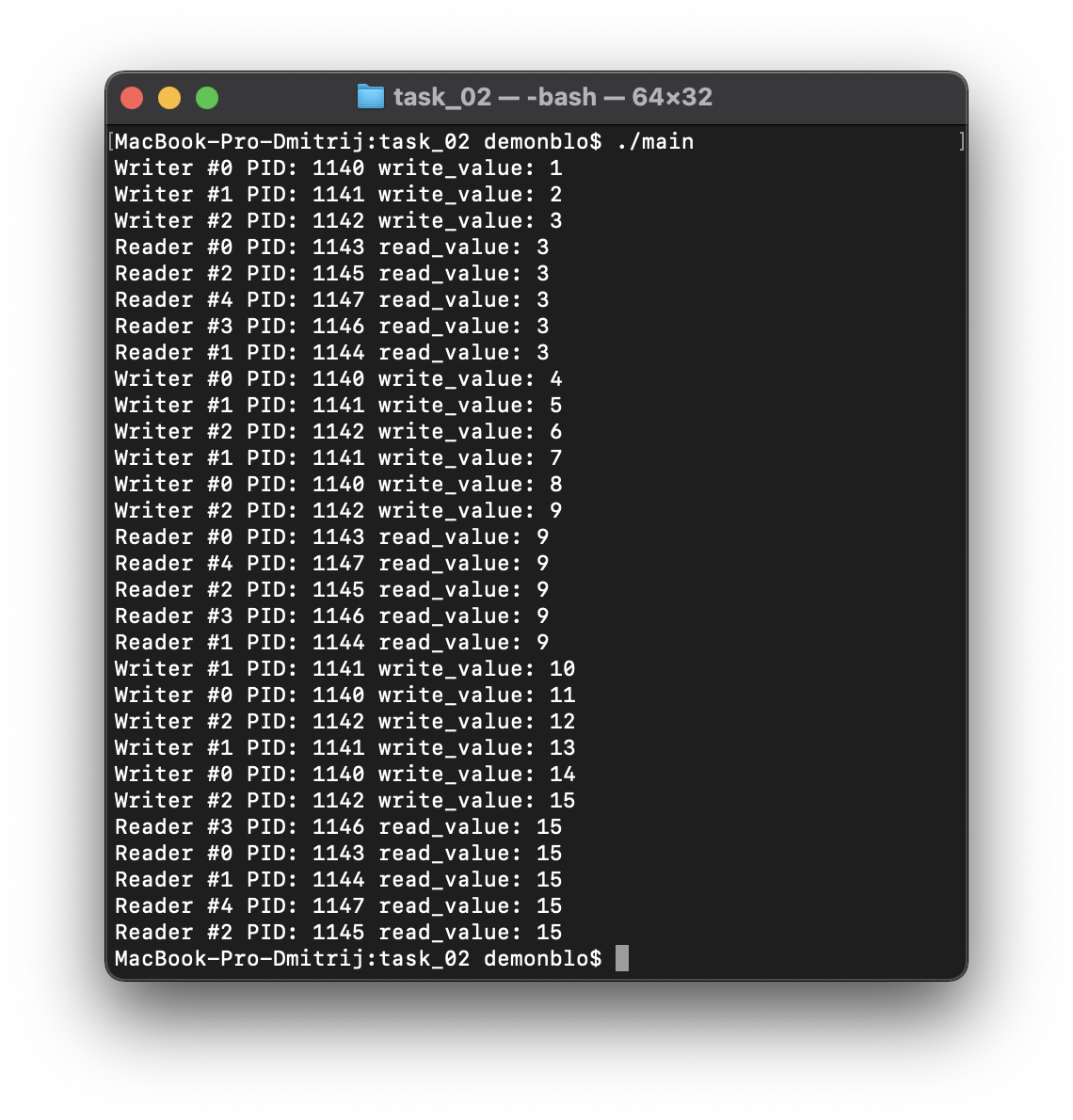
**Задание 2**

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 **INCREMENT** 1  
#define **DECREMENT** (-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**, **INCREMENT**, 0},  
 {**ACTIVE\_READERS**, **CHECK**, 0},  
 {**ACTIVE\_WRITER**, **CHECK**, 0},  
 {**ACTIVE\_WRITER**, **INCREMENT**, 0},  
 {**WAITING\_WRITERS**, **DECREMENT**, 0}};  
#define **START\_WRITE\_SEM\_AMOUNT** 5  
  
struct sembuf stop\_write[] = {{**ACTIVE\_WRITER**, **DECREMENT**, 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**, **INCREMENT**, 0},  
 {**ACTIVE\_WRITER**, **CHECK**, 0},  
 {**WAITING\_WRITERS**, **CHECK**, 0},  
 {**WAITING\_READERS**, **DECREMENT**, 0},  
 {**ACTIVE\_READERS**, **INCREMENT**, 0}};  
#define **START\_READ\_SEM\_AMOUNT** 5  
  
struct sembuf stop\_read[] = {{**ACTIVE\_READERS**, **DECREMENT**, 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**: ****