|  |  |
| --- | --- |
| 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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

**Отчет**

|  |  |
| --- | --- |
| **по лабораторной работе №** | 5 |

**Название:** Взаимодействие параллельных процессов.

**Дисциплина:** Операционные системы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ7-53Б |  |  | А.С.Саркисов |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | Н.Ю. Рязанова |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2020

**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.

**Код программы**

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

const int totalProducers = 5;

const int totalConsumers = 5;

const int bufferSize = 10;

const int perm = S\_IRWXU | S\_IRWXG | S\_IRWXO;

char\* sharedBuffer;

char\* posToWrite;

char\* posToRead;

#define SB 0

#define SE 1

#define SF 2

#define P -1

#define V 1

struct sembuf producerStart[2] = { {SE, P, 0}, {SB, P, 0} };

struct sembuf producerStop[2] = { {SB, V, 0}, {SF, V, 0} };

struct sembuf consumerStart[2] = { {SF, P, 0}, {SB, P, 0} };

struct sembuf consumerStop[2] = { {SB, V, 0}, {SE, V, 0} };

void producer(const int semid, const int consumerNum)

{

for (int i = 0; i < 26; i++) {

char symb = i + 65;

sleep(rand() % 4);

int semP = semop(semid, producerStart, 2);

if (semP == -1)

{

perror("Can't make operation on semaphors.");

exit(1);

}

sharedBuffer[(\*posToWrite) % bufferSize] = symb;

printf("Producer #%d ----> %c\n", consumerNum, sharedBuffer[(\*posToWrite) % bufferSize]);

(\*posToWrite)++;

int semV = semop(semid, producerStop, 2);

if (semV == -1)

{

perror("Can't make operation on semaphors.");

exit(1);

}

}

}

void consumer(const int semid, const int consumerNum)

{

for (int i = 0; i < 26; i++) {

char symb = i + 65;

sleep(rand() % 2);

int semP = semop(semid, consumerStart, 2);

if (semP == -1)

{

perror("Can't make operation on semaphors.");

exit(1);

}

printf("Consumer #%d <---- %c\n", consumerNum, sharedBuffer[(\*posToRead) % bufferSize]);

(\*posToRead)++;

int semV = semop(semid, consumerStop, 2);

if (semV == -1)

{

perror("Can't make operation on semaphors.");

exit(1);

}

}

}

int main()

{

int shmid, semid;

int parent\_pid = getpid();

printf("Parent pid: %d\n", parent\_pid);

if ((shmid = shmget(IPC\_PRIVATE, (bufferSize + 2) \* sizeof(char), IPC\_CREAT | perm)) == -1)

{

perror("Unable to create a shared area.");

exit(1);

}

sharedBuffer = shmat(shmid, 0, 0);

if (\*sharedBuffer == -1)

{

perror("Can't attach memory");

exit(1);

}

posToWrite = sharedBuffer;

posToRead = sharedBuffer + sizeof(char);

sharedBuffer = sharedBuffer + sizeof(char) \* 2;

\*posToWrite = 0;

\*posToRead = 0;

if ((semid = semget(IPC\_PRIVATE, 3, IPC\_CREAT | perm)) == -1)

{

perror("Unable to create a semaphore.");

exit( 1 );

}

int ret\_sb = semctl(semid, SB, SETVAL, 1);

int ret\_se = semctl(semid, SE, SETVAL, bufferSize);

int ret\_sf = semctl(semid, SF, SETVAL, 0);

if ( ret\_se == -1 || ret\_sf == -1 || ret\_sb == -1)

{

perror("Can't set control semaphors.");

exit(1);

}

pid\_t pid;

for (size\_t i = 0; i < totalProducers; i++)

{

switch (pid = fork())

{

case -1:

printf("Can't fork.");

exit(1);

case 0:

producer(semid, i + 1);

exit(0);

}

}

for (size\_t i = 0; i < totalConsumers; i++)

{

switch (pid = fork())

{

case -1:

printf("Can't fork.");

exit(1);

case 0:

consumer(semid, i + 1);

exit(0);

}

}

for (size\_t i = 0; i < totalConsumers + totalProducers; i++)

{

int status;

pid\_t ret\_value = wait(&status);

printf("Child process has finished: PID = %d, status = %d\n", ret\_value, status);

if (WIFEXITED(status))

printf("Child %d finished with %d code.\n\n", ret\_value, WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("Child %d finished from signal with %d code.\n\n", ret\_value, WTERMSIG(status));

else if (WIFSTOPPED(status))

printf("Child %d finished from signal with %d code.\n\n", ret\_value, WSTOPSIG(status));

}

printf("%d\n", \*posToWrite);

if (shmdt(posToWrite) == -1)

{

perror("Can't detach shared memory");

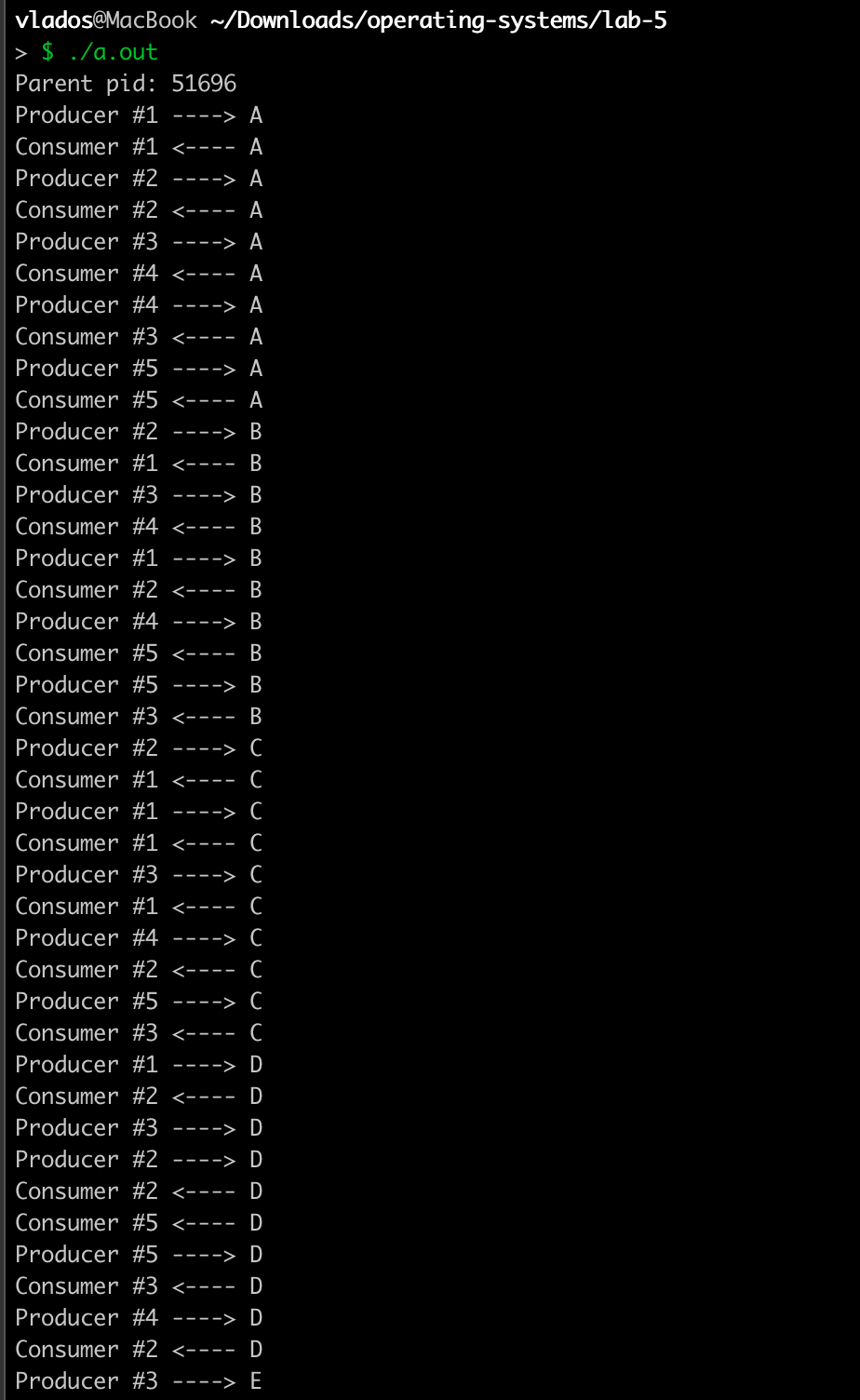
exit(1);

}

exit(0);

}

**Работа 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. Писателей д.б. не меньше 3х, а читателей не меньше 5ти. Для реализации взаимоисключения используются семафоры.

**Код программы**

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <signal.h>

#include <unistd.h>

#define TOTAL\_WRITERS 5

#define TOTAL\_READERS 5

#define ACTIVE\_READER 0

#define ACTIVE\_WRITER 1

#define BIN\_ACTIVE\_WRITER 2

#define WAITING\_WRITER 3

struct sembuf startRead[] = { { WAITING\_WRITER, 0, 0 },

{ ACTIVE\_WRITER, 0, 0 },

{ ACTIVE\_READER, 1, 0 } };

struct sembuf stopRead[] = { {ACTIVE\_READER, -1, 0} };

struct sembuf startWrite[] = { { WAITING\_WRITER, 1, 0 },

{ ACTIVE\_READER, 0, 0 },

{ BIN\_ACTIVE\_WRITER, -1, 0 },

{ ACTIVE\_WRITER, 1, 0 },

{ WAITING\_WRITER, -1, 0 } };

struct sembuf stopWrite[] = { { ACTIVE\_WRITER, -1, 0 },

{ BIN\_ACTIVE\_WRITER, 1, 0 }};

const int perm = S\_IRWXU | S\_IRWXG | S\_IRWXO;

void writer(int semid, int\* buffer, int num) {

while (1)

{

semop(semid, startWrite, 5);

(\*buffer)++;

printf("Writer #%d ----> %d\n", num, \*buffer);

semop(semid, stopWrite, 2);

sleep(rand() % 4);

}

}

void reader(int semid, int\* buffer, int num) {

while (1)

{

semop(semid, startRead, 3);

printf("Reader #%d <---- %d\n", num, \*buffer);

semop(semid, stopRead, 1);

sleep(rand() % 2);

}

}

int main()

{

srand(time(NULL));

int parent\_pid = getpid();

printf("Parent pid: %d\n", parent\_pid);

int shm\_id;

if ((shm\_id = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | perm)) == -1)

{

perror("Unable to create a shared area.");

exit(1);

}

int \*sharedBuffer = shmat(shm\_id, 0, 0);

if (sharedBuffer == (void\*) -1)

{

perror("Can't attach memory");

exit(1);

}

(\*sharedBuffer) = 0;

int sem\_id;

if ((sem\_id = semget(IPC\_PRIVATE, 4, IPC\_CREAT | perm)) == -1)

{

perror("Unable to create a semaphore.");

exit(1);

}

int retValue = semctl(sem\_id, BIN\_ACTIVE\_WRITER, SETVAL, 1);

if (retValue == -1)

{

perror( "Can't set control semaphors.");

exit(1);

}

pid\_t pid = -1;

for (int i = 0; i < TOTAL\_WRITERS && pid != 0; i++) {

pid = fork();

if (pid == -1) {

perror("Writer's fork error.");

exit(1);

}

if (pid == 0) {

writer(sem\_id, sharedBuffer, i);

}

}

for (int i = 0; i < TOTAL\_READERS && pid != 0; i++) {

pid = fork();

if (pid == -1) {

perror("Reader's fork error.");

exit(1);

}

if (pid == 0) {

reader(sem\_id, sharedBuffer, i);

}

}

if (shmdt(sharedBuffer) == -1) {

perror("Can't detach shared memory");

exit(1);

}

if (pid != 0) {

int \*status;

for (int i = 0; i < TOTAL\_READERS + TOTAL\_WRITERS; ++i) {

wait(status);

}

if (shmctl(shm\_id, IPC\_RMID, NULL) == -1) {

perror("Can't free memory!");

exit(1);

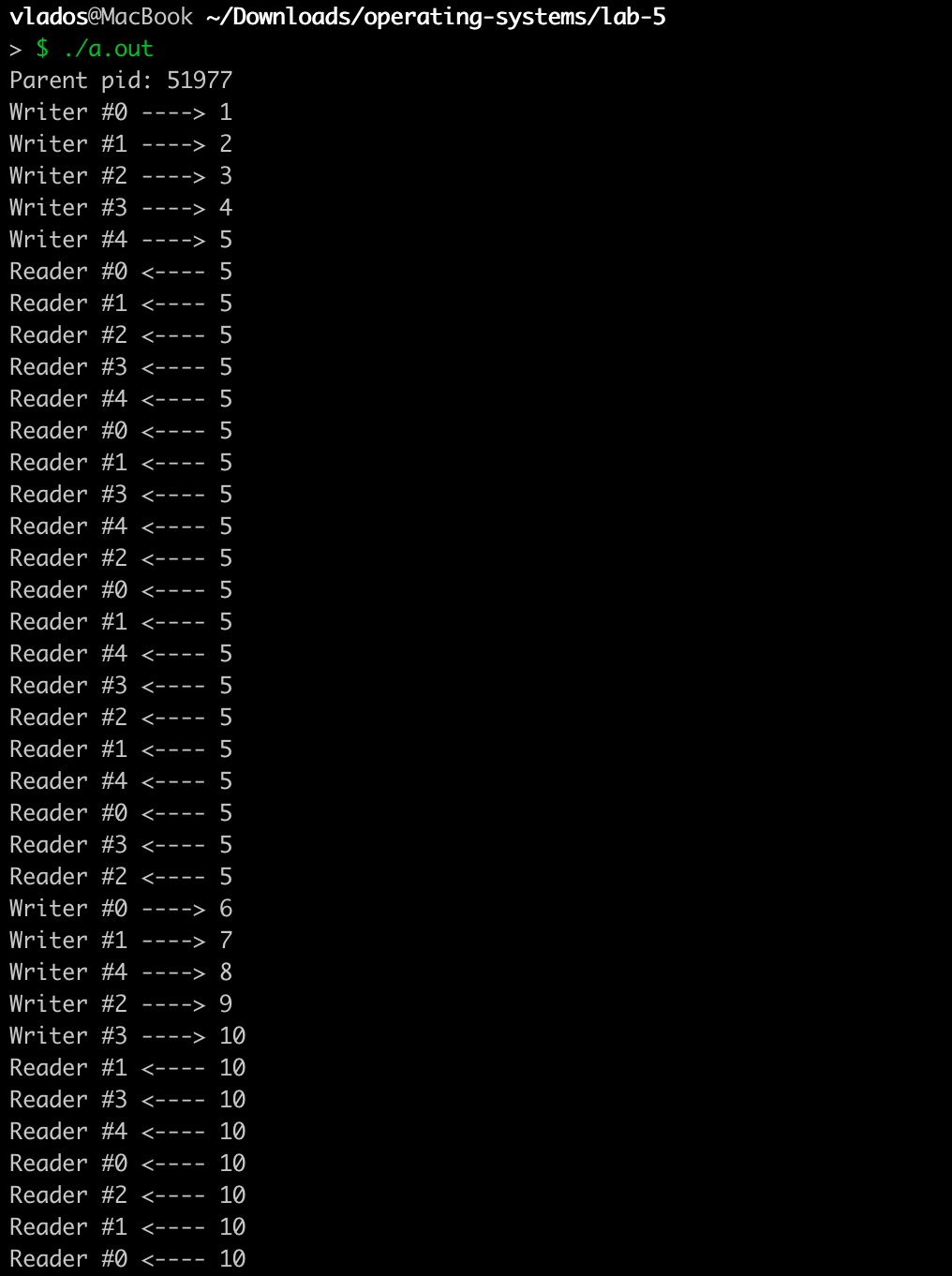
}

}

exit(0);

}

**Работа программы**

****