Лабораторная работа по ОС UNIX
Взаимодействие параллельных процессов

В лабораторной работе исследуются вопросы и формируются навыки использования в приложения таких средств меж процессного взаимодействия, как семафоры и разделяемая память. Работа выполняется на примере двух задач, характерных для взаимодействия асинхронных параллельных процессов: «производство-потребление» и «читатели-писатели».

Решение Э. Дейкстры задачи «производство-потребление»

Постановка задачи - имеется буфер фиксированного размера. Производитель (producer) может произвести единичный объект и поместить его в буфер. Потребитель (consumer) может выбирать объекты из буфера по одному и «потреблять» их. Необходимо обеспечить монопольный доступ производителей и потребителей к буферу: когда производитель помещает элемент в буфер, ни другой производитель или потребитель не должен иметь доступ к буферу; аналогично, когда потребитель берет элемент из буфера, ни другой потребитель или какой-нибудь производитель не могут получить доступ к буферу. В этой задаче буфер является критическим ресурсом (рис.7).

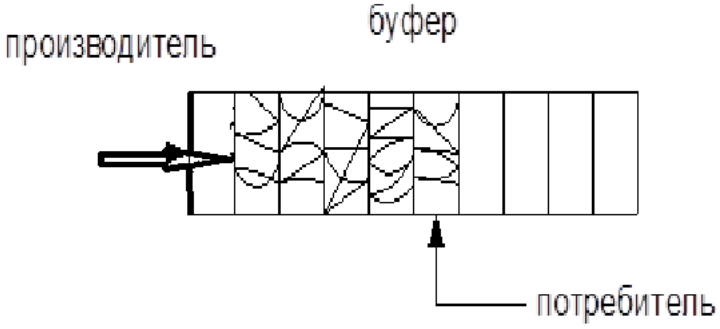

Рис.7

Для решения этой задачи используется два считающих семафора – «БуферПолон» (buffer_full) и «БуферПуст» (buffer_empty) и один бинарный (bin_sem), отслеживающий доступ процессов к буферу. Семафор «БуферПолон» отслеживает количество элементов в буфере в любой момент времени, а семафор «БуферПуст» отслеживает количество пустых элементов.

```
integer N = 24; /*размер буфера*/
semaphore buffer_full, buffer_empty, bin_sem;
Инициализация семафоров:
bin_sem = 1
buffer_full = 0 /* Изначально все ячейки буфера пусты и, таким образом, количество
                   заполненных ячеек   равно 0*/
buffer_empty = N /*Все ячейки буфера изначально пусты */

Producer:
{
/* производит единичный объект*/
P(buffer_empty); /*ждет, когда освободится хотя бы одна ячейка буфера*/
P(bin_sem); /*ждет, когда или другой производитель или потребитель выйдет из
              критической секции*/
/* положить в буфер */
V(bin_sem); /* освобождение критической секции*/
```

```
V(buffer_full); /* инкремент количества заполненных ячеек */
}
```

Когда производитель производит объект, значение семафора buffer_empty уменьшается на 1, если buffer_empty>0, иначе производитель блокируется в ожидании освобождения потребителем хотя бы одной ячейки буфера. Значение bin_sen также декрементируется, чтобы обеспечить монопольный доступ к буферу. Если производитель поместил элемент в ячейку буфера, то значение «buffer_full» инкрементируется. Значение mutex также увеличивается на 1, поскольку задача производителя выполнена.

```
Consumer:
{
/* выбирает из буфера единичный объект*/
 P(buffer_full); /*ждет, когда будет заполнена хотя бы одна ячейка буфера*/
P(bin_sem); /*ждет, когда или потребитель, или другой производитель выйдет из
                критической секции*/
/* взять из буфера */
V(bin_sem); /* освобождение критической секции*/
V(buffer_empty); /* инкремент количества пустых ячеек */
}
```

Поскольку потребитель удаляет элемент из буфера, значение «buffer_full» уменьшается на 1, если это возможно, иначе при значении buffer_full равным нулю потребитель блокируется на этом семафоре, ожидая, когда производитель заполнит хотя бы одну ячейку буфера. Значение bin_sem также уменьшается, так что ни другой производитель, ни производитель не могут получить доступ к буферу в данный момент.

Монитор Хоара «Читатели-писатели»

Задача «Читатели-писатели» является одной из известнейших в ОС. Для этой задачи характерно наличие двух типов процессов: процессов «читателей», которые могут только читать данные, и процессов «писателей», которые могут только изменять данные. Читатели могут работать параллельно, поскольку они друг другу не мешают, а писатели могут работать только в режиме монопольного доступа: только один писатель может получить доступ к разделяемой переменной, причем, когда работает писатель, то другие писатели и читатели не могут получить доступ к этой переменной. Рассмотрим монитор Хоара «Читатели-писатели», для которого характерно наличие четырех процедур: start_read(), stop_read(), start_write(), stop_write() (листинг 10).

```
RESOURCE MONITOR;
var
    active_readers : integer;
     active_writer : logical;
    can_read, can_write : conditional;
procedure star_read
    begin
        if (active_writer or turn(can_write)) then wait(can_read);
        active_readers++; //инкремент читатетей
        signal(can_read);
    end;
procedure stop_read
     begin
         active_readers--; //декремент читателей
```

```
          if (active_readers = 0) then signal(can_write);
      end;
procedure start_write
       begin
            if ((active_readers > 0) or active_writer) then wait(can_write);
            active_writer:= true;
       end;
procedure stop_write
        begin
             active_writer:= false;
             if (turn(can_read) then signal(can_read)
               else signal(can_write);
end;
begin
    active_readers:=0;
    active_writer:=false;
end.
```

Листинг 10

Когда число читателей равно 0, процесс писатель получает возможность начать работу. Новый процесс читатель не сможет начать свою работу пока работает процесс писатель и не появится истинное значение условия can_read.

Писатель может начать свою работу, когда условие can_write станет равно истине (true).

Когда процессу читателю нужно выполнить чтение, он вызывает процедуру start_read. Если читатель заканчивает читать, то он вызывает процедуру stop_read. При входе в процедуру star_read новый процесс читатель сможет начать работать, если нет процесса писателя, изменяющего данные, в которых заинтересован читатель, и нет писателей, ждущих свою очередь (turn(can_write)), чтобы изменить эти данные. Второе условие нужно для предотвращения бесконечного откладывания процессов писателей в очереди писателей.

Процедура start_read завершается выдачей сигнала signal(can_read), чтобы следующий читатель в очереди читателей смог начать чтение. Каждый следующий читатель, начав чтение выдает signal(can_read), активизирует следующего читателя в очереди читателей. В результате возникает цепная реакция активизации читателей и она будет идти до тех пор, пока не активизируются все ожидающие читатели.

«Цепная реакция» читателей является отличительной особенностью данного решения, которое эффективно «запускает» параллельное выполнение читателей.

Процедура stop_read уменьшает количество активных читателей: читателей, начавших чтение. После ее многократного выполнения количество читателей может стать равным нулю. Если число читателей равно нулю, выполняется signal(can_write), активизирующий писателя из очереди писателей.

Когда писателю необходимо выполнить запить, он вызывает процедуру start_write. Для обеспечения монопольного доступа писателя к разделяемым данным, если есть читающие процессы или другой активный писатель, то писателю придется подождать, когда будет установлено значение «истина» в переменной типа условие can_write. Когда писатель получает возможность работать логической переменной can_write присваивается значение «истина», что заблокирует доступ других процессов писателей к разделяемым данным.

Когда писатель заканчивает работу, предпочтение отдается читателям при условии, что очередь ждущих читателей не пуста. Иначе для писателей устанавливается переменная can_write. Таким образом исключается бесконечное откладывание читателей.

Семафоры UNIX

ОС Unix/Linux поддерживают наборы считающих семафоров. Семантически такие наборы считающих семафоров представлены в системе массивами и доступ к отдельному семафору набора осуществляется по индексу. В ядре ОС имеется таблица семафоров, в которой отслеживаются все созданные наборы семафоров. Основным свойством набора семафоров является возможность одной неделимой операцией изменить значения всех или части семафоров набора.

В ОС Unix System V имеются функции (API) для создания набора семафоров, изменения управляющих параметров набора и выполнения операций на семафорах.

Функция **semget()** создает новый набор семафоров или открывает уже имеющийся. Прототип функции semget() имеет следующий вид:

# include < sys / types.h >
# include < sys / ipc.h >
# include < sys / sem.h >
int semget(key_t key, int numb_sem, int flag);

```
SYSTEM CALL: semget();

  PROTOTYPE: int semget (key_t key, int nsems, int semflg);
    RETURNS: IPC-идентификатор множества семафоров в случае успеха
             -1 в случае ошибки
                errno: EACCESS (доступ отклонен)
                       EEXIST (существует нельзя создать (IPC_ESCL))
                       EIDRM (множество помечено как удаляемое)
                       ENOENT (множество не существует, не было исполнено
                               ни одного IPC_CREAT)
                       ENOMEM (не хватает памяти для новых семафоров)
                       ENOSPC (превышен лимит на количество множеств
                               семафоров)
```

В случае успешного завершения функция возвращает дескриптор семафора, а в случае неудачи - -1. Параметр numb_sem задает количество семафоров в наборе. Параметр key задает идентификатор семафора. Если значением key является макрос IPC_PRIVATE, то создается набор семафоров, который смогут использовать только процессы, порожденные процессом, создавшим семафор. Параметр flag представляет собой результат побитового сложения прав доступа к семафору и константы IPC_CREATE.

Например, следующий системный вызов создает набор из двух семафоров с идентификатором 100, для которого устанавливаются следующие права доступа: чтение-запись - для владельца, чтение – для членов группы и остальных пользователей.

int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
int isem_descr = semget(100, 2, IRC_CREATE | perms);

Для гарантированного создания нового набора семафоров совместно с флагом IPC_CREATE можно указать флаг IPC_EXCL.

Функция **semctl()** позволяет изменять управляющие параметры набора семафоров.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

```
SYSTEM CALL: semctl();
PROTOTYPE: int semctl ( int semid, int semnum, int cmd, union semun arg );
   RETURNS: натуральное число в случае успеха
            -1 в случае ошибки:
               errno = EACCESS (доступ отклонен)
```

```
                        EFAULT (адрес, указанный аргументом arg, ошибочен)
                        EIDRM (множество семафоров удалено)
                        EINVAL (множество не существует или неправильный semid)
                        EPERM (EUID не имеет привилегий для cmd в arg-е)
                        ERANGE (значение семафора вышло за пределы допустимых
                                значений)
NOTES: Выполняет операции, управляющие множеством семафоров
```

После получения идентификатора дескриптора семафора в системе значение семафора можно изменять с помощью системного вызова **semop**():

int semop(int isem_descr, struct sembuf *op, int nop);

```
SYSTEMCALL: semop();
PROTOTYPE: int semop( int semid, struct sembuf *sop, unsigned nsop);
  RETURNS: 0 в случае успеха (все операции выполнены)
           -1 в случае ошибки
              errno: E2BIG (nsops больше чем максимальное число
                            позволенных операций)
                     EACCESS (доступ отклонен)
                     EAGAIN (при поднятом флаге IPC_NOWAIT операция не может
                             быть выполнена)
                     EFAULT (sops указывает на ошибочный адрес)
                     EIDRM (множество семафоров уничтожено)
                     EINTR (сигнал получен во время сна)
                     EINVAL (множество не существует или неверный semid)
                     ENOMEM (поднят флаг SEM_UNDO, но не хватает памяти
                             для создания необходимой undo-структуры)
                     ERANGE (значение семафора вышло за пределы
                             допустимых значений)
```

В случае неудачи при попытке изменения значения семафора функция возвращает значение –1, при успешном вызове возвращается 0. Параметр op – указатель на массив объектов типа struct sembuf, параметр sop определяет количество семафоров набора, над которыми выполняется операция.

```
struct sembuf{
        short sem_num; // индекс семафора
        short sem_op; // операция: увеличение, уменьшение или проверка значения
        short sem_flg; // флаги
      }
```

На семафорах UNIX определено три типа операций: декремент, инкремент и проверка на 0.

Если sem_op < 0, то значение семафора уменьшается. Это соответствует получению ресурса, который контролирует семафор. Захват ресурса или семафора, контролирующего данный ресурс, исключает возможность его захвата другим процессом.

Если sem_op > 0, то значение семафора увеличивается на соответствующую величину и ожидающий в очереди к семафору процесс разблокируется.

Наконец, если sem_op = 0, то вызывающий процесс будет усыплен (sleep()), пока значение семафора не станет нулем.

Для выполнения первых двух операций у процесса должно быть право на изменение, для выполнения третьей достаточно права на чтение. Чтобы увеличить значение семафора, системному вызову semop следует передать требуемое число. Чтобы уменьшить значение семафора, нужно передать требуемое число, взятое с обратным знаком; если результат получается отрицательным, операция не может быть успешно выполнена. Для третьей операции нужно передать 0; если текущее значение семафора отлично от нуля, операция не может быть успешно выполнена.

Для операции могу быть установлены флаги. Флаг SEM_UNDO означает, что операция выполняется в проверочном режиме, то есть требуется только узнать, можно ли успешно выполнить данную операцию.

При отсутствии флага IPC_NOWAIT системный вызов semop() может быть приостановлен до тех пор, пока значение семафора, благодаря действиям другого процесса, не позволит успешно завершить операцию (ликвидация множества семафоров также приведет к завершению системного вызова). Подобные операции называются ``*операциями с блокировкой*''. С другой стороны, если обработка завершается неудачей и не указано, что выполнение процесса должно быть приостановлено, операция над семафором называется ``*операцией без блокировки*''.

Системный вызов semop оперирует не с отдельным семафором, а с множеством семафоров, применяя к нему ``массив операций". Массив содержит информацию о том, с какими семафорами нужно оперировать и каким образом. Выполнение массива операций с точки зрения пользовательского процесса является неделимым действием. Это значит, во-первых, что если операции выполняются, то только все вместе и, во-вторых, что другой процесс не может получить доступ к промежуточному состоянию множества семафоров, когда часть операций из массива уже выполнилась, а другая часть еще не успела.

Операционная система, разумеется, выполняет операции из массива по очереди, причем порядок не оговаривается. Если очередная операция не может быть выполнена, то эффект предыдущих операций аннулируется. Если таковой оказалась операция с блокировкой, выполнение системного вызова приостанавливается. Если неудачу потерпела операция без блокировки, системный вызов немедленно завершается, возвращая значение -1 как признак ошибки, а внешней переменной errno присваивается код ошибки.

Если вызов *semop* попытается уменьшить значение семафора до отрицательного числа или посчитает, что значение семафора равно нулю, когда на самом деле это не так, то ядро заблокирует вызывающий процесс. Этого не произойдет в том случае, если в полях *sem_flg* элементов массива, где *sem_op* меньше или равно нулю, указан флаг IPC_NOWAIT.

В полях *sem_flg* объектов *struct sembuf* может быть установлен еще один флаг — SEM_UNDO. Этот флаг дает ядру указание отслеживать изменение значения семафора (произведенное вызовом *semop*). При завершении вызывающего процесса ядро ликвидирует сделанные изменения, чтобы процессы, ожидающие изменения семафоров, не были заблокированы навечно, что может произойти в том случае, если вызывающий процесс "забудет" отменить сделанные им изменения.

Пример:

В следующем примере создается набор из двух семафоров с идентификатором 100. Затем значение первого семафора уменьшается на 1, а значение второго семафора проверяется (подробнее см. Тренс Чан стр. 328).

```
# include < sys / types.h >
# include < sys / ipc.h >
# include < sys / sem.h >
struct sem_buf sem_arr[2] = {{0, -1, SEM_UNDO | SEM_NOWAIT}, {1, 0, 1} };
int main(void)
{
  int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
  int isem_descr = semget(100, 2, IRC_CREATE | perms );
  if (isem_descry = = -1) { perror("semget"); return 1;}
  if ( semop (isem_descry, sem_arr, 2) = = -1) { perror("semop"); return 1;}
  return 0;
}
```

Обратите внимание на объявление и инициализацию массива sem_arr[2] структур sem_buf. В результате одной неделимой операцией semop() выполняются действия сразу над всеми, т.е. двумя, семафорами набора.

Флаг SEM_UNDO установленный в поле sem_flg структуры sembuf дает ядру указание отслеживать изменение семафора, выполненное операцией semop. При завершении вызывающего процесса супервизор отменит сделанные изменения для того, чтобы процессы ожидающие освобождения семафоров не были заблокированы навечно. Это может произойти, если вызывающий процесс по каким-либо причинам не отменил произведенные им изменения.

Разделяемая память

Разделяемая память является средством передачи информации от процесса к процессу. Разделяемая память (сегменты разделяемой памяти) была разработана для сокращения времени передачи сообщений за счет исключения необходимости копировать текст сообщения из пространства пользователя в пространство ядра. Это обеспечивается за счет возможности подключения разделяемого сегмента к адресному пространству процесса, а именно за счет возможности получения процессом указателя на разделяемый сегмент. Аналогично программным каналам разделяемые сегменты создаются в разделяемой памяти, которой является область данных ядра системы. В отличие от программных каналов разделяемая память не имеет встроенных средств взаимоисключения и, как правило, используется совместно с семафорами. Аналогично семафорам дескрипторы всех разделяемых сегментов находятся в системой таблице разделяемой памяти ядра системы.

В ОС Unix System V имеются функции (API) для создания разделяемой памяти, изменения управляющих параметров созданного сегмента, подключения сегмента к адресному пространству процесса, т.е. получения указателя на него и отключения сегмента разделяемой памяти от адресного пространства процесса.

После выполнения команды fork() дочерний процесс наследует подключаемые к нему разделяемые сегменты памяти.

После выполнения команды exec() все подключаемые к процессу разделяемые сегменты памяти отключаются от него, но не удаляются.

По завершении exit() все подключенные разделяемые сегменты памяти отключаются (но не удаляются).

Функция **shmget()** создает новый разделяемый сегмент или, если сегмент уже существует, то права доступа подтверждаются. Прототип функции shmget() имеет следующий вид:

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget(key_t *key*, **int** *size*, **int** *shmflg*);

shmget() возвращает идентификатор разделяемому сегменту памяти, соответствующий значению аргумента key. Создается новый разделяемый сегмент памяти с размером size (округленным до размера, кратного PAGE_SIZE), если значение key равно IPC_PRIVATE или если значение key не равно IPC_PRIVATE и нет идентификатора, соответствующего key; причем, выражение shmflg&IPC_CREAT истинно. Поле *shmflg* состоит из:

IPC_CREAT
: служит для создания нового сегмента. Если этого флага нет, то функция **shmget()** будет искать сегмент, соответствующий ключу *key* и затем проверит, имеет ли пользователь права на доступ к сегменту.

IPC_EXCL
: используется совместно с **IPC_CREAT** для того, чтобы не создавать существующий сегмент заново.

mode_flags (младшие 9 битов)
: указывают на права хозяина, группы и др. В данный момент права системой не используются.

Если создается новый сегмент, то права доступа копируются из *shmflg* в *shm_perm*, являющийся членом структуры *shmid_ds*, которая определяет сегмент.
Структура *shmid_ds* имеет такую форму:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* права операции */
    int             shm_segsz;   /* размер сегмента (в байтах) */
    time_t          shm_atime;   /* время последнего подключения */
    time_t          shm_dtime;   /* время последнего отключения */
    time_t          shm_ctime;   /* время последнего изменения */
    unsigned short  shm_cpid;    /* идентификатор процесса создателя
*/
    unsigned short  shm_lpid;    /* идентификатор последнего
пользователя */
    short           shm_nattch;  /* количество подключений */
};
struct ipc_perm {
    key_t  key;
```

```
    ushort uid;   /* действующие идентификаторы владельца и группы
euid и egid */
    ushort gid;
    ushort cuid;  /* действующие идентификаторы создателя euid и egid
*/
    ushort cgid;
    ushort mode;  /* младшие 9 битов shmflg */
    ushort seq;   /* номер последовательности */
};
```

При создании нового сегмента разделяемой памяти системный вызов инициализирует структуру данных *shmid_ds* следующим образом устанавливаемые значения:

- **shm_perm.cuid** и **shm_perm.uid** становятся равными значению идентификатора эффективного пользователя вызывающего процесса;
- **shm_perm.cgid** и **shm_perm.gid** устанавливаются равными идентификатору эффективной группы пользователей вызывающего процесса.
- Младшим 9-и битам **shm_perm.mode** присваивается значение младших 9-и битов *shmflg*.
- **shm_segsz** присваивается значение *size.*
- Устанавливаемое значение
  shm_lpid, **shm_nattch**, **shm_atime** и **shm_dtime** становится равным **нулю.**
- **shm_ctime** устанавливается на текущее время.

Если сегмент уже существует, то права доступа подтверждаются, а проверка производится для того, чтобы убедиться, что сегмент не помечен на удаление.

Функция **shmctl()** позволяет изменять управляющие параметры сегмента. Прототип функции имеет вид:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Вызов **shmctl**() выполняет управляющую операцию, указанную в *cmd*, над общим сегментом памяти System V, чей идентификатор задан в *shmid*.

Функция **shmat()** возвращает указатель на сегмент и ее прототип имеет следующий вид:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Вызов **shmat**() подключает сегмент общей памяти System V с идентификатором *shmid* к адресному пространству вызывающего процесса. Адрес подключения, указанный в *shmaddr*, учитывается следующим образом:

- Если значение *shmaddr* равно NULL, то система выбирает подходящий (неиспользуемый) адрес для подключения сегмента.
- Если значение *shmaddr* не равно NULL, а в *shmflg* указан флаг **SHM_RND**, то подключение производится по адресу *shmaddr*, округлённому до ближайшего значения кратного **SHMLBA**.
  В противном случае *shmaddr* должно быть выровнено по адресу страницы, к которому производится подключение.

При успешном выполнении **shmat**() возвращается адрес подключённого общего сегмента памяти; при ошибке возвращается *(void *) -1*, а в *errno* содержится код ошибки.

Функция **shmdt()** «отключает» разделяемый сегмент от адресного пространства процесса и ее прототип имеет следующий вид:

#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void ****shmaddr*);**

Вызов **shmdt**() отключает сегмент общей памяти, находящийся по адресу *shmaddr*, от адресного пространства вызывающего процесса. Отключаемый сегмент должен быть подключён по адресу *shmaddr* с помощью вызова **shmat**().

При успешном выполнении **shmdt**() возвращается 0; при ошибке возвращается -1, а в *errno* содержится код ошибки.

Пример:

В примере программа открывает сегмент разделяемой памяти размером 1024 байта с идентификатором 100. Если такая область не существует, то она создается системным вызовом shmget() с полными правами доступа для всех категорий пользователей. Затем сегмент присоединяется к виртуальному адресному пространству процесса. Системный вызов shmat() возвращает адрес сегмента и по этому адресу записывается сообщение «Hello». После этого сегмент отсоединяется.

После этого любой процесс по идентификатору сегмента может присоединить этот сегмент к своему адресному пространству и прочитать записанное сообщение.

```
# include < sys / types.h >
# include < sys / ipc.h >
# include < sys / shm.h >
# include <string.h>
int main(void)
{
  int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
  int fd = shmget(100, 1024, IRC_CREATE | perms );
  if (fd = = -1) { perror(“shmget”); exit(1);}
  char *addr = (char*)shmat(fd,0,0);
  if (*addr == (char*) -1) { perror(“shmat”); return 1;}
  strcpy(addr, “Hello”);
  if (shmdt(addr) == -1) perror(“shmdt”);
  return 0;
}
```

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 обоим 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.