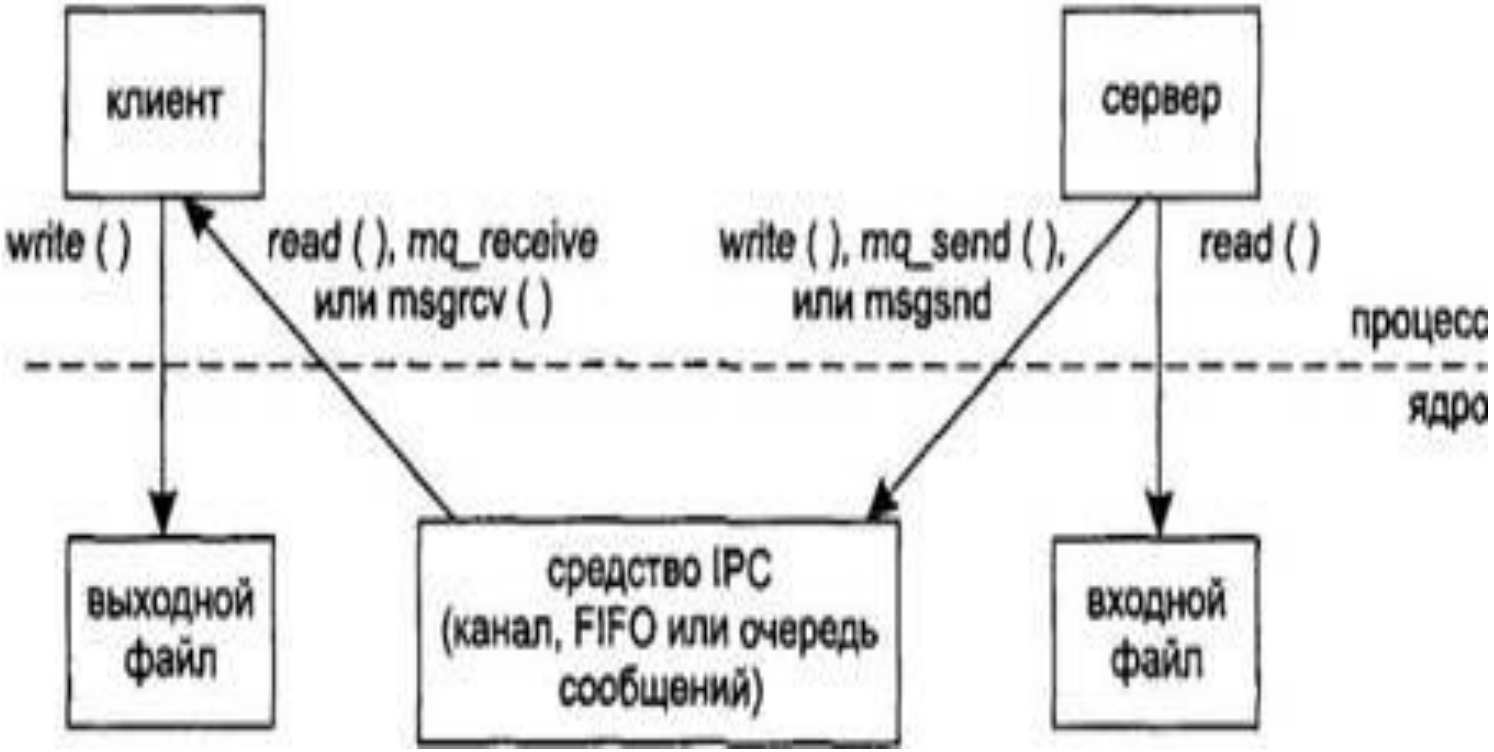
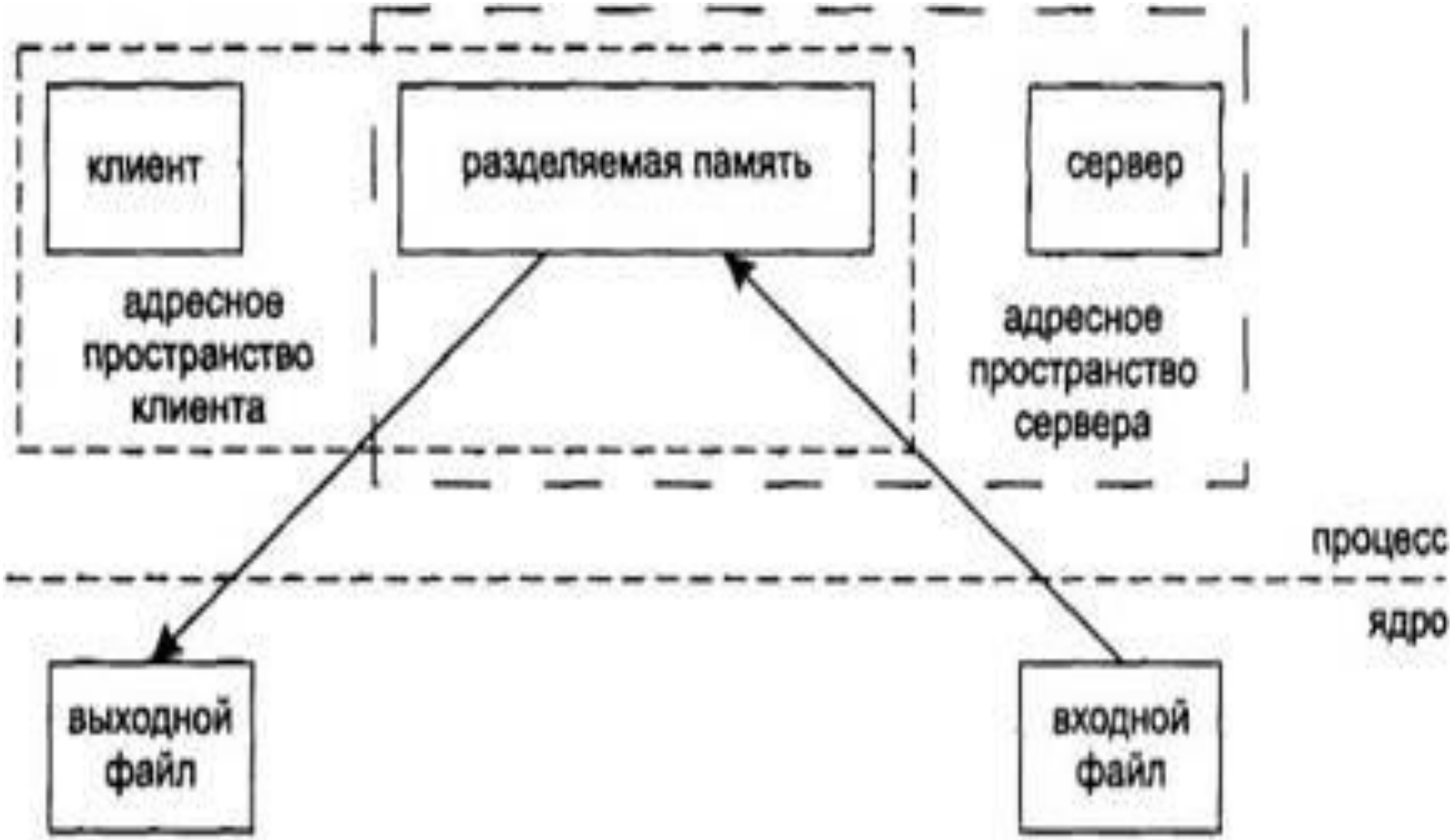
Разделяемая память

Обмен данными через ядро



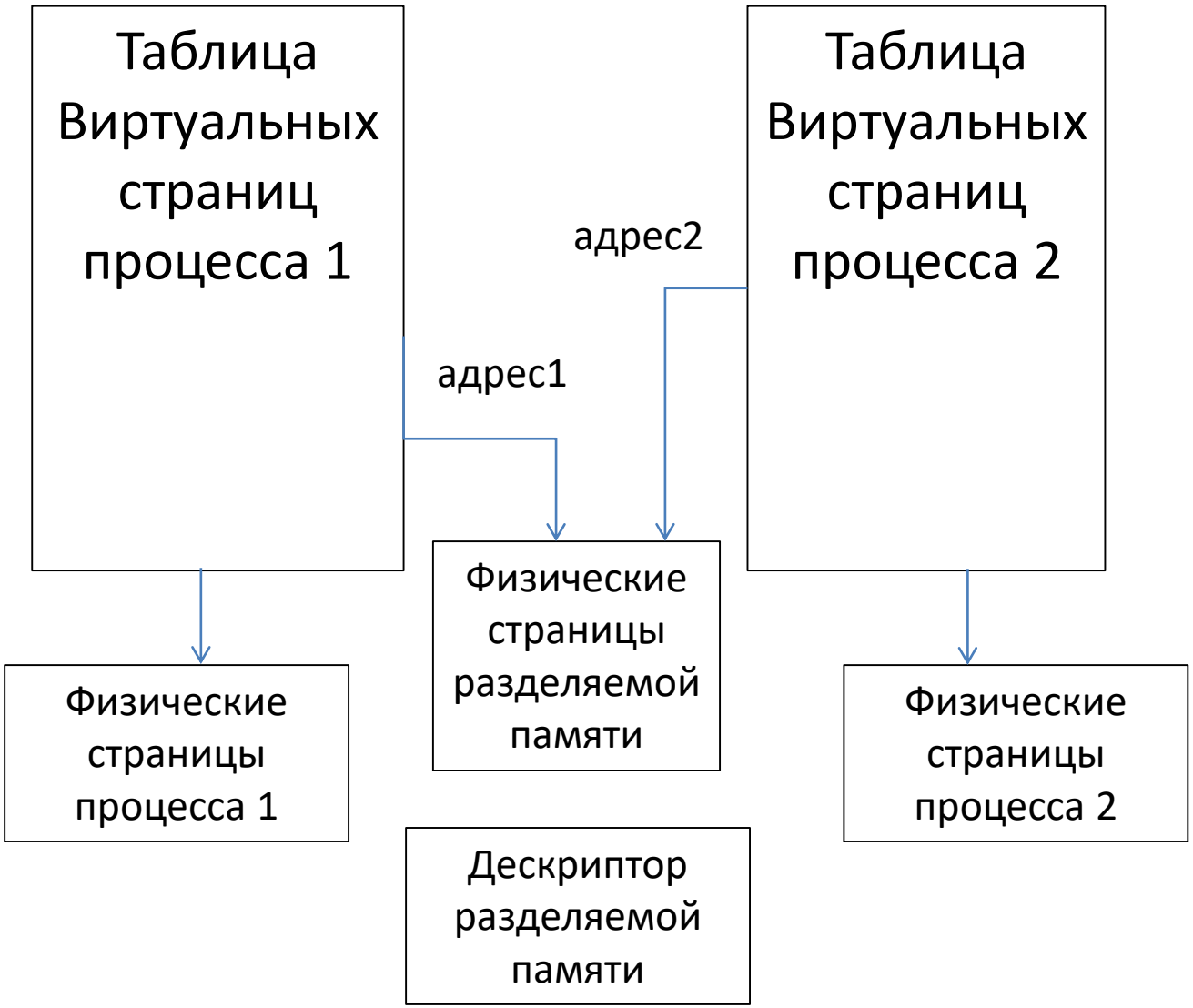
Обмен данными через разделяемую память



Разделяемая память

- Под разделяемой памятью подразумевается область основной памяти, которую операционная система отображает в адресное пространство нескольких процессов.
- Обработка данных, находящихся в разделяемой памяти, выполняется обычными средствами языков программирования, которые используются для работы с памятью.
- Поддерживается произвольный порядок доступа к данным в режимах чтения и записи, что позволяет организовать обмен данными в двух направлениях.
- Обеспечивается наивысшая скорость обмена данными и наибольшие объемы передачи данных.
- Считается надежным средством передачи данных.
- Разделяемую память могут использовать только процессы работающие в рамках одной вычислительной системы.

Организация разделяемой памяти



Дескриптор разделяемой памяти

```
struct shmid_ds {
    struct ipc_perm     shm_perm;     /* создатель и права доступа */
    int     shm_segsz;    /* размер сегмента (bytes) */
__kernel_time_t     shm_atime; /* Время последнего присоединения к сегменту*/
__kernel_time_t     shm_dtime; /* Время последнего отсоединения процесса от
    сегмента*/
__kernel_time_t     shm_ctime; /* Время последнего изменения этой структуры*/
__kernel_ipc_pid_t  shm_cpid; /* pid создателя */
__kernel_ipc_pid_t  shm_lpid; /* pid последнего процесса, обратившегося к сегменту */
unsigned short      shm_nattch; /* Число процессов, присоединивших  сегмент*/
     /* следующее носит частный характер */
    unsigned short      shm_unused;     /* compatibility */
    void                *shm_unused2;   /* ditto - used by DIPC */
    void                *shm_unused3;   /* unused */
};
```

Предельные характеристики разделяемой памяти

ipcs -m -l

------ Пределы совм. исп. памяти --------

максимальное количество сегментов = 4096

максимальный размер сегмента (кбайт) = 32768

max total shared memory (kbytes) = 8388608

минимальный размер сегмента (байт) = 1

Подходы к работе с разделяемой памяти

- в стиле UNIX System V, используя функции расширения POSIX:XSI;
- через функции POSIX (стандарт POSIX.1-2001)

Создание разделяемого сегмента

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

Возвращает: идентификатор разделяемого сегмента памяти в случае успеха -1 в случае ошибки:

errno =

EINVAL (Ошибочно заданы размеры сегмента)

EEXIST (Сегмент существует, нельзя создать)

EIDRM (Сегмент отмечен для удаления, или был удален)

ENOENT (Сегмент не существует)

EACCESS (Доступ отклонен)

ENOMEM (Недостаточно памяти для создания сегмента)

Параметры операции создания разделяемого сегмента

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

key – ключ сегмента или флаг IPC_PRIVATE;

size – размер сегмента в байтах;

#define SHMMAX 0x2000000 /* максимальный размер сегмента (32Мб)*/

#define SHMMIN 1 /* минимальный размер сегмента (байт) */

#define SHMMNI 4096 /* максимальное число сегментов */

shmflg – флаги и права доступа к сегменту

IPC_CREAT – если сегмента для указанного ключа не существует, он должен быть создан;

IPC_EXCL – применяется совместно с флагом IPC_CREAT (если существует сегмент с указанным ключом, то доступ к сегменту не производится)

Права доступа к сегменту

0400 - разрешено чтение пользователю, владеющему разделяемой памятью;

0200 - разрешена запись пользователю, владеющему разделяемой памятью;

0040 - разрешено чтение пользователям, входящим в ту же группу, что и владелец разделяемой памяти;

0020 - разрешена запись пользователям, входящим в ту же группу, что и владелец разделяемой памяти;

0004 - разрешено чтение всем остальным пользователям;

0002 - разрешена запись всем остальным пользователям.

id_shm_1=shmget(1200,32,0666 | IPC_CREAT);

Присоединение сегмента

void* shmat (int shmid, char *shmaddr, int shmflg);
Возвращает: адрес, по которому сегмент был привязан к процессу, в случае успеха -1 в случае ошибки:

errno =

EINVAL (Ошибочно значение IPC ID или адрес привязки)

ENOMEM (Недостаточно памяти для привязки сегмента)

EACCES (Права отклонены)

Параметры операции присоединения сегмента

int shmat (int shmid, char *shmaddr, int shmflg);

shmid – идентификатор сегмента;

shmaddr – 0 или виртуальный адрес по которому должен быть присоединен сегмент;

shmflg – флаги сегмента

SHM_RND – округление адреса

SHM_REMAP – проверка на пересечение с другими областями памяти

SHM_RDONLY – только для чтения

char * addr_shm_1=shmat(id_shm_1,0,0);

Отсоединение сегмента

int shmdt (char *shmaddr);

Возвращает: 0 или -1 в случае ошибки:

errno =

EINVAL (ошибочно указан адрес привязки)

shmdt(addr_shm_1);

Контроль и управление сегментом

int shmctl (int shmqid, int cmd, struct shmid_ds *buf);

Возвращает: 0 в случае успеха -1 в случае ошибки: errno =

EACCESS (Нет прав на чтение при cmd, равном IPC_STAT)

EFAULT (Адрес, на который указывает буфер, ошибочен при cmd, равном IPC_SET или IPC_STAT) EIDRM (Сегмент был удален во время вызова) EINVAL (ошибочный shmqid)

EPERM (попытка выполнить команду IPC_SET или IPC_RMID, но вызывающий процесс не имеет прав на запись, измените права доступа)

Параметры операции контроля и управления сегментом

int shmctl (int shmqid, int cmd, struct shmid_ds *buf);

shmqid – идентификатор сегмента;

cmd – режим работы функции (IPC_STAT, IPC_SET, IPC_RMID);

buf – информация о состоянии сегмента

shmctl(id_shm_1,IPC_RMID,0);

Уничтожение разделяемого сегмента

Изменение характеристик и уничтожение разделяемого сегмента памяти допускается только процессу, у которого эффективный идентификатор пользователя принадлежит либо root, либо создателю или владельцу сегмента. Он может изменить права доступа и передать разделяемый сегмент на владение другому пользователю.

Если область не была присоединена ни к одному из процессов, ядро освобождает сегмент и все выделенные физические страницы памяти. Если же сегмент по-прежнему подключен к каким-то процессам, то разрешено продолжать работать с ним и запрещает новым процессам присоединить его.

Информация о состоянии сегмента

```
struct shmid_ds {
    struct ipc_perm     shm_perm;    /* создатель и права доступа */
    size_t              shm_segsz;   /* размер сегмента (bytes) */
    time_t              shm_atime;   /* Время последнего
                                        присоединения к сегменту*/
    time_t              shm_dtime;   /* Время последнего
                                    отсоединения процесса от сегмента.*/
    time_t              shm_ctime;   /* Время последнего
                                        изменения этой структуры*/
    pid_t shm_cpid;     /* pid создателя */
    pid_t shm_lpid;     /* pid последнего процесса, обратившегося к
    сегменту */
    shmatt_t            shm_nattch;  /* Число процессов,
                                        присоединивших  сегмент*/
```

Информация о владельце и правах сегмента

```
struct ipc_perm {
    key_t    __key;      /* ключ сегмента */
    uid_t    uid;        /* эффективный UID владельца */
    gid_t    gid;        /* эффективный GID владельца */
    uid_t    cuid;       /* эффективный UID создателя */
    gid_t    cgid;       /* эффективный GID создателя */
    unsigned short mode;       /* права */
unsigned short __seq; /*внутренний идентификатор*/
        };
```

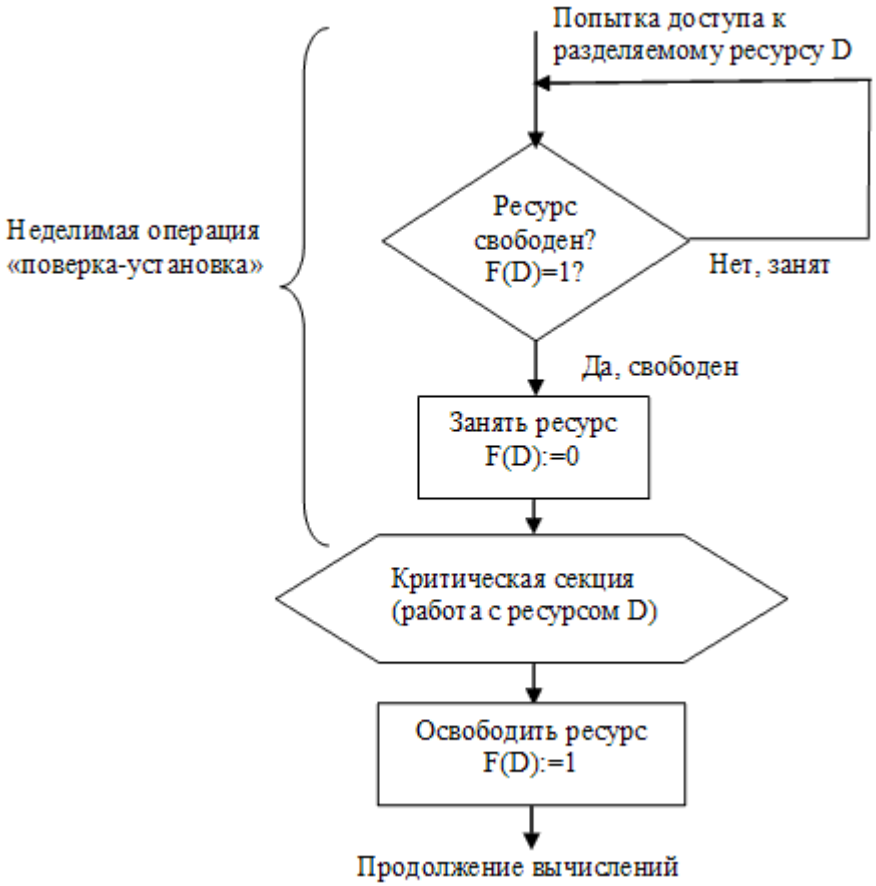
Функции для работы с общей памятью в стиле POSIX

Метод	**Описание**
shm_open	Открывает сегмент в общей памяти
shm_unlink	Удаляет сегмент из общей памяти
ftruncate	Задаёт или изменяет размер разделяемой памяти
mmap	Подключает существующий сегмент разделяемой памяти к адресному пространству процесса
shm_close	Отсоединение сегмента от адресного пространства

Протокол доступа процессов к разделяемой памяти

Нет встроенного способа обеспечения взаимного исключения к разделяемой памяти. Взаимное исключение можно реализовать с использованием общих (блокирующих) переменных.

1. Проверка общей переменной, доступной всем процессам
2. Захват ресурса (блокировка) или цикл активного ожидания
3. Выполнение критических операций с ресурсом
4. Освобождение ресурса (разблокировка)



Свойства протокола синхронизации

1**. Взаимное исключение**

В любой момент времени только один поток может выполнять критическую секцию (КС)

2. **Отсутствие взаимной блокировки**

Если несколько потоков пытаются войти в КС, и она свободна, то хотя бы один поток это осуществит.

3. **Отсутствие излишних задержек**

Если поток пытается войти в КС, а остальные потоки либо выполняют свои некритические секции либо завершены, то первый поток немедленно войдет в критическую секцию.

4. **Возможность входа**

Поток, который пытается войти в критическую секцию, когда-нибудь это сделает.

Проблема программных методов взаимоисключения

Невозможно гарантировать неразрывность выполнения отдельных действий:

- Программа может прерваться в любой момент
- Возможны произвольные последовательности операций в параллельной программе (недетерминизм)

Взаимоисключение с общей переменной

*lock=0; общая переменная в разделяемой памяти

Процесс 1

```
while (true) {
while (*lock!=0);
*lock=1;
 Критическая секция;
*lock=0;
Некритическая секция;
}
```

Процесс 2

```
while (true) {
while (*lock!=0);
 *lock=1;
 Критическая секция;
*lock=0;
Некритическая секция;
}
```

действия while(*lock); *lock = 1;
не является атомарным.
Нарушение условия 1 (оба процесса могут войти в критическую сеуцию).

```
movl (lock),%ebx
L: movl (%ebx),%ecx
 cmp %ecx,0
jne L
movl (%ebx),1
```

Запрет прерываний

Запрещении всех аппаратных прерываний при входе процесса в критическую область. Таким образом становится невозможным прерывание по таймеру, что исключает передачу процессора другому процессу.

```
while (true) {
asm (“CLI”); запрет прерывания
while (*lock!=0);
*lock=1;
asm (“STI”);  разрешение прерывания
 Критическая секция;
*lock=0;
Некритическая секция;
}
```

Недостатки:

- если прерывания отключены пользовательским процессом и, в результате какого-либо сбоя, не включены обратно - это приведет к краху системы;
- в многопроцессорной системе запрет прерываний касается только одного процессора. Остальные процессоры продолжат свою работу в нормальном режиме, сохраняя доступ к разделяемым данным.
- не исключается и бесконечно долгое откладывание процесса, т.е. алгоритм не обеспечивает выполнение условия (4).

Строгое чередование

*lock=0; общая переменная

Процесс 1

```
while (true) {
while (*lock!=0);
 Критическая секция;
*lock=1;
Некритическая секция;
}
```

Процесс 2

```
while (true) {
while (*lock!=1);
 Критическая секция;
*lock=0;
Некритическая секция;
}
```

Процессы попадают в критическую область строго по очереди. Не один из них не может попасть в критическую секцию два раза подряд. Нарушение условия 3.

Синхронизация с использованием флагов готовности

shared int ready [1:n]=([n] 0) флаги готовности в разделяемой памяти

process Woker [i=1 to n] {

while (true) {

Процесс готов войти в критическую секцию

ready[i] = 1;

Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

while (ready[j≠i]==1);

Критическая секция

ready[i] = 0;

}

}

Не выполняется условие 2. Может возникнуть взаимная блокировка (deadlock).

Алгоритм Петерсона

```
shared int ready [N]=([N] 0); флаги готовности в разделяемой памяти
shared int turn [N]; флаги очередности
void process(int i) {      //    номер процесса    от 0 до N-1
   int j;
   for( j = 0; j < N-1; j++ ) {   // перебор этапов
      ready [i] = j;
       turn [j] = i; //хранит идентификатор последнего процесса, добравшегося до этого этапа
       while( turn[j] == i and pos_is_big(i, j) ); // цикл активного ожидание
   }
   // Критическая секция
 ready[i] = 0;
}
// Для прохода на следующий этап соревнуемся со всеми другими процессами,
// Проходит тот, у которого больше идентификатор
bool pos_is_big(int i, int j)
 {int k;
   for( k = 0; k < N-1; k++ )              // опрос остальных процессов
      if( k != i and ready[k] >= j )
         return true;
   }
   return false;
}
```

Перед входом в *критическую секцию* процесс сначала заявляет о своем намерении в нее войти, но затем пытается предоставить право на вход в *критическую секцию* другим процессам и только после того, как другой процесс ее выполнил или не желает в нее войти, входит сам в свою *критическую секцию*.

Сценарий взаимоблокировки для алгоритма Петерсона

для N = 3:

1. Сначала начинается процесс 0, устанавливает ready[0] = 0 turn[0] = 0, а затем ждет.
2. Далее начинается процесс 2, устанавливает ready[2] = 0 и turn[0] = 2, а затем ждет.
3. Процесс 1 запускается последним, устанавливает ready[1] = 0 и turn[0] = 1, а затем ждет.
4. отсеивание будет происходить в два этапа: на нулевом этапе отсеивается один из потоков, на первом второй;
5. Процесс 2 первым заметил изменение в turn[0] и поэтому устанавливает j = 1, ready[2] = 1 и turn[1] = 2.
6. Процессы 0 и 1 блокируются, потому что ready[2] большой.
7. Процесс 2 не заблокирован, поэтому он устанавливает j = 2, выходит из цикла for и входит в критический раздел. После завершения он устанавливает ready[2] = 0, но сразу начинает снова конкурировать за критический раздел, тем самым устанавливая turn[0] = 2 и ожидая.
8. Процесс 1 первым замечает изменение в turn[0] и продолжает как процесс 2 раньше.
9. ...
10. Процесс 1 и 2 завершает процесс конкурирования 0.
11. "Невезучий" процесс может ждать пока другие процессы O(N2) раз войдут в критическую секцию (квадратичное ожидание).

Взаимное исключение с очередью

Пусть все потоки, которые хотят войти в критическую секцию выстраиваются в очередь:

1. будем выдавать каждому пришедшему наименьший из номеров больший всех номеров в очереди;
2. обслуживать будем в порядке возрастания номеров.
3. Как найти и занять нужный номер?

- мы должны посмотреть на номера в очереди и выбрать себе подходящий; (процессы могут получить одинаковые номера)
- будем вместо номера использовать пару: номер и идентификатор потока; даже если выбранный номер не уникален, пара будет уникальной.
- Гарантирует взаимное исключение, отсутствие блокировки и линейное ожидание.

Алгоритм Лампорта (булочной)

shared enum {false, true} choosing[n];
shared int number[n];
Изначально элементы этих массивов инициируются значениями false и 0 соответственно.
Введем следующие обозначения
(a,b) < (c,d), если a < c или если a == c и b < d
max(a0, a1, ..., an) – это число k такое, что k >= ai для всех i = 0, ...,n
Структура процесса Pi

```
while (условие) {
  choosing[i] = true;
  number[i] = max(number[0], ..., number[n-1]) + 1;
  choosing[i] = false;
  for(j = 0; j < n; j++){
    while(choosing[j]); // Ждём, пока поток j получит свой номер
// Ждём, пока все потоки с меньшим номером или с таким же номером,
 // но с более высоким приоритетом, закончат свою работу:
    while(number[j] != 0 && (number[j],j) < (number[i],i));
  }
    Критическая секция
  number[i] = 0;
Оставшаяся часть
}
```

Задание 1 к лабораторной работе 9

Напишите **3 программы**, которые запускаются в произвольном порядке и построчно записывают свои индивидуальные данные в один файл. Пока не закончит писать строку одна программа, другие две не должны обращаться к файлу. **Интервалы записи в файл и количество записываемых строк определяются входными параметрами**, задаваемыми при запуске каждой программы. При завершении работы одной из программ, другие должны продолжить свою работу. Синхронизация работы программ должна осуществляться с помощью общих переменных, размещенных в разделяемой памяти.

Задание 2 к лабораторной работе 9

Напишите **две программы**, которые работают параллельно и обмениваются массивом целых чисел через две общие разделяемые области. Через первую область первая программа передает массив второй программе. Через вторую область вторая программа возвращает первой программе массив, каждый элемент которого уменьшен на 1. Обе программы должна вывести получаемую последовательность чисел. Синхронизация работы программ должна осуществляться с помощью общих переменных, размещенных в разделяемой памяти.