# 1 Тема 11. Эффективное взаимодействие процессов

Как было отмечено в предыдущей теме, использование семафоров пакета IPC, совместно с базовыми средствами синхронизации процессов, позволяет ре- шать все задачи взаимодействия процессов, естественно при наличии корректного алгоритма устранения взаимных блокировок.

К сожалению, указанных инструментов недостаточно для *эффективной син- хронизации процессов*, которые запускаются и взаимодействуют в системе *асинх- ронно*.

Причина состоит в том, что ОС не знает о возможности и деталях такого асинхронного взаимодействия, поэтому процессы сами должны позаботиться об алгоритмах и средствах такого взаимодействия.

В общем случае, любое взаимодействие обеспечивается передачей между процессами некоторого количества структурированных данных, которое можно ре- ализовать через файловую систему ОС. Но такой подход, во многих случаях, является неэффективным, поскольку требует значительных затрат времени на пере- мещение данных на внешние носители ЭВМ и обратно.

Пакет IPC предоставляет программисту два варианта эффективной реализа- ции асинхронного взаимодействия многих процессов:

* *разделяемая память*;
* *очереди сообщений*.

С этой целью, учебный материал данной темы разделен на четыре части:

* *подраздел 1.1* - раскрывает идейный аспект изучаемых средств;
* *подраздел 1.2* - посвящен описанию системных вызовов, обеспечивающих ра- боту с разделяемой памятью;
* *подраздел 1.3* - демонстрирует решение типовой проблемы синхронизации, известной как «*Задача о читателях и писателях*»;
* *подраздел 1.4* - посвящен описанию системных вызовов, обеспечивающих ра- боту с очередями сообщений.

## Прикладные средства пакета IPC

Поскольку ПО ЭВМ предназначено для решения прикладных задач, то разра- батывая и реализовывая алгоритм решения конкретной задачи, прикладной про- граммист стремится сделать ее наиболее надежной, привлекательной и быстро- действующей. Соответственно, на этом пути, он стремится использовать наиболее эффективные системные средства ОС, которые бы помогли ему достичь желаемой цели.

Естественно, что наиболее быстродействующими и, во многом, наиболее эф- фективными средством являются системные вызовы ОС, которые обеспечивают максимальные возможности по обеспечению реализации любых алгоритмов. Тем не менее, как мы убедились ранее, системные вызовы требуют излишней детали- зации приложения, например, при взаимодействии процессов через полудуплекс-

ные каналы или детального знания работы ядра ОС, например, при использовании сигналов. Поэтому, чтобы обеспечить эффективную разработку приложений, осо- бенно в плане взаимодействия асинхронно выполняющихся процессов, был разра- ботан пакет *System V IPC*, включающий единые средства идентификации (адреса- ции), а также универсальное средство синхронизации, поддерживающее функции семафоров, первоначально разработанные Дейкстрой.

Дополнительно, чтобы освободить программиста от рутинной работы, свя- занной с «изобретением» индивидуальных средств передачи сообщений, а также для создания инструментов работы с данными, которые бы не уступали по простоте использования модели потоков процессов (нитей, *threads*), были разработаны сред- ства: *разделяемой памяти* и *очередей сообщений*.

**Разделяемая память** — *информационный объект данных*, создаваемый и хранящийся в ядре ОС, который процесс может:

* *создать* или *удалить*;
* *подключить* к своему пространству данных или *отсоедениться* от него;
* *работать с ним* как с собственной структурой данных.

**Очередь сообщений** — универсальный *«механизм» временного хранения* в ядре ОС *последовательности типизированных данных*, которые процессы могут помещать и извлекать для своих нужд.

В совокупности с универсальным «механизмом» адресации (идентификации) и «механизмом» семафоров, разделяемая память и очереди сообщений образуют *набор прикладных средств* системного пакета *System V IPC*.

## Разделяемые сегменты памяти

**В стандарте POSIX-2001** *разделяемый объект памяти* определяется как ***объект***, представляющий собой память ЭВМ, который может быть *параллельно отображен* в адресное пространство более чем одного процесса.

Таким образом, процессы могут *иметь* общие области виртуальной памяти и

*разделять* содержащиеся в них данные.

**Единицей** разделяемой памяти является *сегмент*, который может быть соз- дан одним из асинхронно взаимодействующих процессов.

Такой сегмент продолжает существовать, пока один из процессов не удалит его или ядро ОС не будет перезапущено.

**Создаваемый сегмент памяти** имеет *атрибуты идентификации* (адреса- ции), которые являются общими для всех средств пакета IPC, а также *индивиду- альные права доступа*, которые обеспечиваются каждым процессом, подключаю- щим этот сегмент.

Общий список системных вызовов, обеспечивающих работу с разделяемыми сегментами памяти, имеет вид:

#include <sys/ipc.h> #include <sys/shm.h>

int shmget(key\_t key,

size\_t

size,

int shmflg);

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

int int

shmdt (const void \*

shmctl(int

shmid, int

shmaddr);

cmd, *struct shmid\_ds* \*buf);

Рассмотрим каждую из указанных функций по отдельности.

**Создание нового** или **получение идентификатора** уже существующего раз- деляемого сегмента памяти осуществляется системным вызовом *shmget(...)*, имею- щего вид:

int shmget(key\_t key, size\_t size, int shmflg);

где *key* — ключ доступа к средствам пакета IPC, полученный с помощью функции *ftok(...)*, или - специальное значение *IPC\_PRIVATE*; причем, в последнем случае, создается приватный разделяемый сегмент памяти (см. замечание ниже);

*size* — задает требуемый *размер сегмента в байтах*, который выравнивается до размера, кратного ***PAGE\_SIZE***;

*shmflg* – флаги, которые играют роль *только при создании* нового сегмента разде- ляемой памяти и определяют права различных пользователей при доступе к этому сегменту; сами флаги являются побитной комбинацией значений, выполняемых с помощью побитной операции " или " – " | ", что задает права доступа:

* IPC\_CREAT – если сегмента для указанного ключа не существует, он должен быть создан;
* IPC\_EXCL – применяется совместно с флагом *IPC\_CREAT*. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом

переменная *errno*, описанная в файле *<errno.h>*, примет значение *EEXIST*;

* 0400 – разрешено чтение для пользователя, создавшего сегмент;
* 0200 – разрешена запись для пользователя, создавшего сегмент;
* 0040 – разрешено чтение для группы пользователя, создавшего сегмент;
* 0020 – разрешена запись для группы пользователя, создавшего сегмент;
* 0004 – разрешено чтение для всех остальных пользователей;
* 0002 – разрешена запись для всех остальных пользователей.

### Замечание

Существует два варианта создания ключа *key* для вызова *shmget(...)*:

* *Стандартный способ*. В качестве значения ключа для системного вызова исполь- зуется значение, сформированное функцией *ftok(...)* для некоторого *имени файла* и *номера экземпляра области разделяемой памяти*. В качестве флагов постав-

ляется комбинация прав доступа к создаваемому сегменту и флага *IPC\_CREAT*. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же он уже существовал, то мы просто получим его дескриптор. Возможно добавление к этой комбинации флагов флага *IPC\_EXCL*, гарантирующий нормальное завершение системного вызова

только в том случае, если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной *errno*, описанной в файле *errno.h*, будет установлено в *EEXIST*.

* *Нестандартный способ*. В качестве значения ключа указывается специальное значение *IPC\_PRIVATE*. Использование значения *IPC\_PRIVATE* всегда приводит к

попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции *ftok(...)* ни при одной комбинации ее параметров. Наличие флагов *IPC\_CREAT* и *IPC\_EXCL*, в этом случае, *игнорируется*.

При удачном завершении вызова *shmget(...)* возвращается идентификатор сег- мента *shmid*, и *-1* при ошибке, причем переменная *errno* приобретает одно из следующих значений:

* *EINVAL* - если создается новый сегмент, а *size* < *SHMMIN* или *size* >

*SHMMAX*, либо новый сегмент не был создан; второй вариант - сегмент с

данным ключем существует, но *size* больше чем размер этого сегмента;

* *EEXIST* - если значение *IPC\_CREAT | IPC\_EXCL* было указано, а сегмент уже существует;
* *ENOSPC* - если все возможные идентификаторы сегментов уже распределены (*SHMMNI*) или если размер выделяемого сегмента превысит системные ли- миты (*SHMALL*);
* *ENOENT* - если не существует сегмента для ключа *key*, а значение

*IPC\_CREAT* не указано;

* *EACCES* - если у пользователя нет прав доступа к сегменту разделяемой памяти;
* *ENOMEM* - если в памяти нет свободного для сегмента пространства.

**Замечание** Ядро ОС, для каждого созданного разделяемого сегмента памяти, создает управляю- щий набор данных, который определяется структурами:

struct shmid\_ds {

struct ipc\_perm shm\_perm;

int time\_t time\_t time\_t

unsigned short unsigned short short

shm\_segsz;

shm\_atime; shm\_dtime; shm\_ctime; shm\_cpid; shm\_lpid; shm\_nattch;

// права операции

// размер сегмента (в байтах)

// время последнего подключения

// время последнего отключения

// время последнего изменения

// идентификатор процесса создателя

// идентификатор последнего пользователя

// 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

};

В дальнейшем, программист может использовать эту информацию, например, с по- мощью системного вызова *shmctl(...)*.

Уже созданный сегмент разделяемой памяти *подстыковывается* к адресному пространству процесса с помощью системного вызова:

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

где *shmid* - задает идентификатор разделяемого сегмента, полученный с помощью функции *shmget(...)*;

*shmaddr* - адрес, по которому сегмент должен быть присоединен, причем, если:

* *shmaddr* == ***NULL***, то система выбирает для подстыкованного сегмента под- ходящий (неиспользованный) адрес;
* *shmaddr* != ***NULL***, то подстыковка производится по адресу *shmaddr*; *shmflg* — флаг, определяющий свойства разделяемого сегмента памяти:
* по умолчанию, *значение=0* и присоединяемый сегмент будет доступен *и на чтение, и на запись*, если процесс обладает необходимыми правами;
* флаг *SHM\_RDONLY* предписывает присоединить сегмент *только для чтения*.

**Замечание**

Поскольку свойства сегментов зависят от аппаратных особенностей управления па- мятью, не всякий адрес является приемлемым:

* если установлен флаг *SHM\_RND*, адрес присоединения округляется до величи- ны, кратной константе *SHMLBA*;
* если *shmaddr* задан как пустой указатель *(NULL)*, то система выбирает адрес при- соединения *по своему усмотрению*.

**Другим важным вопросом** является поведение сегментов разделяемой памяти при выполнении процессом системных вызовов *fork()*, *exec()* и функции *exit()*:

* при выполнении системного вызова *fork(...)*, все области разделяемой памяти, раз- мещенные в адресном пространстве процесса, *наследуются порожденным про- цессом*;
* при выполнении системных вызовов *exec(...)* и функции *exit(...)*, все области раз- деляемой памяти, размещенные в адресном пространстве процесса, *исключают- ся из его адресного пространства, но продолжают существовать в операци-*

*онной системе*.

При удачном выполнении, системный вызов *shmat(...)* обновляет содержимое структуры *shmid\_ds*, связанной с разделяемым сегментом памяти, следующим об- разом:

* *shm\_atime* устанавливается в текущее время;
* *shm\_lpid* устанавливается в идентификатор вызывающего процесса;
* *shm\_nattch* увеличивается на 1.

В случае ошибки функция *shmat(...)* возвращает значение *-1*, а переменной

*errno* присваивается номер ошибки:

* *EACCES* - вызывающий процесс не имеет прав для подстыковки заданного типа;
* *EINVAL* - неправильное значение *shmid*, не выравненное по границе страницы или неправильное значение *shmaddr*, или ошибка подключения к ***brk***, или *SHM\_REMAP* было определено, но *shmaddr* равно ***NULL***;
* *ENOMEM* - не хватает памяти для описателя или таблиц страниц.

**Чтобы закрепить изученный материал**, рассмотрим *взаимодействие двух программ*, каждая из которых фиксирует:

* *собственное число* запусков;
* а также *общее число* всех запусков.

Для решения подобной задачи, достаточно использовать *разделяемый массив*

памяти, состоящий из трех целых чисел:

* *первый элемент массива* используется как счетчик запуска программы 1;
* *второй элемент* – счетчик запуска программы 2;
* *третий элемент* – счетчик запуска обеих программ суммарно.

**Замечание**

Программа, которая запущенна первой, должна проинициализировать разделяемую па- мять.

**Замечание**

После запуска, каждая из программ изменяет свою часть разделяемого сегмента памяти и завершает свою работу, но созданный сегмент остается в ОС и любая запущенная программа может получить к нему доступ. В частности, можно воспользоваться утилитой ***ipcs***.

Отсоединение, ранее подключенного сегмента разделяемой памяти, осущест- вляется системным вызовом:

int shmdt (const void \* shmaddr);

где *shmaddr* - задает начальный адрес отсоединяемого сегмента.

При успешном завершении функции возвращется результат равеный *0* и об- новляется содержимое структуры *shmid\_ds*:

* + *shm\_dtime* устанавливается в текущее время;
  + *shm\_lpid* устанавливается в идентификатор вызывающего процесса;
  + *shm\_nattch* уменьшается на 1; если это значение становится равным 0, а сег- мент помечен на удаление, то сегмент удаляется из памяти; в любом случае,

освобождается занятая ранее этим сегментом область памяти в адресном пространстве процесса.

В случае неудачи, возвращается ***-1*** и в переменную *errno* устанавливается код ошибки:

* + *EACCES* — вызывающий процесс не имеет нужных прав доступа;
  + *EIDRM* - удален идентификатор *shmid*;
  + *EINVAL* - неправильный идентификатор *shmid*;
  + *ENOMEM* - невозможно выделить память для дескриптора или таблицы стра- ниц.

**Замечание** Перед использованием функции *shmdt()*, отсоединяемый сегмент разделяемой памяти должен быть присоединен с помощью функции *shmget()*.

Общее управление сегментами разделяемой памяти, с использованием струк- туры типа *shmid\_ds*, осуществляется посредством системного вызова:

int shmctl(int shmid, int cmd, *struct shmid\_ds* \*buf);

где *shmid* - является дескриптором System V IPC для сегмента разделяемой памяти или - значением, которое вернул системный вызов *shmget(...)*, при создании сегмен- та или при его поиске по ключу.;

*cmd* - определяет управляющие команды:

* + *IPC\_STAT* - получить информацию о состоянии разделяемого сегмента, которая копируется в буфер *buf*;
  + *IPC\_SET* - переустановить характеристики разделяемого сегмента памяти по информации содержащейся в буфере *buf*;
  + *IPC\_RMID* - удалить разделяемый сегмент, причем пользователь должен быть владельцем, создателем или суперпользователем процесса;

*buf* — *NULL* или указатель на структуру типа *shmid\_ds*.

При удачном выполнении возвращается *0*, а при ошибке *-1*, причем перемен- ной *errno* присваиваются следующие значения:

* + *EACCES* - возникает, если запрашивается *IPC\_STAT*, а *shm\_perm.modes* не дает доступа *msqid* к чтению;
  + *EFAULT* - аргумент *cmd* равен *IPC\_SET* или *IPC\_STAT*, а адрес, указываемый

*buf*, недоступен;

* + *EINVAL* - эта ошибка происходит, если *shmid* является неверным идентифи- катором сегмента или *cmd* является неправильной командой;
  + *EIDRM* - эта ошибка возвращается, если *shmid* указывает на удаленный идентификатор;
  + *EPERM* - эта ошибка возвращается, если была произведена попытка выпол- нить *IPC\_SET* или *IPC\_RMID*, эффективный идентификатор вызывающего процессы не является идентификатором создателя, данным в *shm\_perm.cuid*,

владельца, в соответствие с *shm\_perm.uid*, или суперпользователя;

* + *EOVERFLOW* - возвращается если запрашивается *IPC\_STAT*, а значения *gid* или *uid* слишком велики для помещения в структуру, на которую указывает *buf*.

## Задача о читателях и писателях

**Аппарат** *разделяемых сегментов памяти* предоставляет нескольким процес- сам возможность одновременного доступа *к общей области памяти*:

* + **Ранее**, мы рассмотрели доступ к разделяемой памяти, который не требовал какого-либо согласования действий между процессами.
  + **Обеспечивая корректность доступа**, процессы тем или иным способом дол- жны синхронизировать свои действия.
  + **В качестве средства** синхронизации удобно использовать *семафоры*.

**Чтобы правильно** использовать *семафоры*, при доступе к разделяемым сег- ментам памяти, необходимо:

* + тщательно проанализировать задачу и выделить в процессах *критические интервалы* (области программы);
  + определить механизмы, обеспечивающие *взаимное исключение* разделяющих общие данные процессов.

**В качестве учебного примера**, демострирующего совместное использование синхронизации и разделяемые сегменты памяти, рассмотрим задачу «*Читатели- писатели*». Общая интерпретация этой задачи — следующая.

**Писатель**, владея публичным ресурсом, периодически пишет на нем книги:

* + для написания книги, требуется случайный интервал времени *twrite*;
  + для обдумывания новой книги, требуется случайный интервал времени *tsleep*.

**Читатели**, являясь последовательностью процессов, возникающих через слу-

чайные интервалы времени *tsleep*, обращаются к публичному ресурсу писателя:

* + читают, если публикация имеется, на что требуется случайный интервал вре- мени *tread*;
  + завершают работу, если публикация отсутствует.

### Общие требования:

* + процессы-читатели имеют *одновременный доступ* на операцию чтения, но обязаны ждать, пока процесс-писатель не закончит свою работу;
  + процесс-писатель *должен дождаться* завершения процесса чтения читате- лей, но не допускает к чтению процессы, которые пришли во время его напи- сания новой книги.

**Представим** решение данной задачи с помощью одного разделяемого сег- мента памяти *shareseg* и массива из двух семафоров *sembuf[2]*:

* + *sembuf[0]* — число читателей, приступивших к чтению;
  + *sembuf[1]* — значение 0 — можно читать.

**Алгоритм** *процесса-читателя*, представленный на листинге 1.3, :

* + *создаются*, если не созданы, ключи *key1* и *key2*, массив семафоров *sembuf[2]*

и сегмент разделяемой памяти *shareseg*;

* + *процесс-читатель*, в бесконечном цикле, через случайный интервал времени

*tsleep* порождает дочерние процессы;

* + каждый дочерний процесс: ожидает возможности чтения, а затем — завершает работу, если *shareseg=0*, или читает случайное время *tread*, если *shareseg>0*.

**Замечание**

Для правильного запуска процессов-читателей — смотри подраздел 2.2 по выполнению

лабораторной работы.

**Алгоритм** *процесса-писателя*, представленный на листинге 1.4:

* + *создаются*, если не созданы, ключи *key1* и *key2*, массив семафоров *sembuf[2]*

и сегмент разделяемой памяти *shareseg*;

* + процесс-писатель *обнуляет значения семафоров* независимо от того, создал он их или использует уже созданные;
  + *выполняется цикл* по количеству задуманных публикаций;

### в каждом цикле:

* + *блокируется* подключение новых процессов-читателей;
  + *ожидается* завершение чтения, - уже читающих;
  + *объявляется* о написании новой книги и выполняется сам процесс — случай- ный интервал времени *twrite*;
  + после написания книги, *разрешается доступ* на чтение и обдумывание ново- го произведения случайный интервал времени *tsleep*;
  + **после завершения** всех циклов:
  + *ожидается* завершение всех читателей и обнуляется *shareseg*;
  + *печатается сообщение* о завершении работы программы и осуществляется выход.

*Листинг 1.4. Алгоритм, моделирующий действия процесса-писателя*

**Замечание** Программа-писатель работает в паре с программой-читателем, поэтому для правильно- го их запуска — смотри подраздел 2.2 по выполнению лабораторной работы.

## Передача сообщений

**Третьим** и последним средством, входящим в *System V IPC*, являются *очере- ди сообщений*.

**Очереди сообщений** - это наиболее семантически нагруженный способ взаи- модействия процессов через каналы связи, в котором *на передаваемую информа- цию накладывается определенная структура*, так что процесс, принимающий дан- ные, может четко определить, где заканчивается одна порция информации и начи- нается другая.

Такая модель позволяет задействовать один и тот же канал связи для переда- чи данных *в двух направлениях между несколькими процессами*.

**Очереди сообщений**, как семафоры и разделяемая память, *являются сред- ством связи с непрямой адресацией*, что требует:

* + *инициализации их*, для организации взаимодействия процессов;
  + *специальных действий*, для освобождения системных ресурсов по окончании взаимодействия.

**Пространством имен** очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции *ftok()*, а для передачи данных используются **системные примитивы**, в виде функций *send()* и *receive()*, которым

в качестве параметра передаются *IPC-дескрипторы* очередей сообщений, одноз- начно идентифицирующие эти данные во всей вычислительной системе.

**Очереди сообщений** имеют следующие особенности:

* + *располагаются* в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди.
  + *каждый элемент* списка представляет собой отдельное сообщение.
  + *каждое сообщение* имеет атрибут, называемый *типом сообщения*.

**Выборка сообщений** из очереди, соответствующая примитиву *receive()*, мо-

жет быть выполнена тремя способами:

* + *В порядке FIFO*, независимо от типа сообщения.
  + *В порядке FIFO*, для сообщений конкретного типа.
  + *Первым выбирается сообщение с минимальным типом*, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем

же типом.

**Реализация примитивов** *send()* и *receive()* обеспечивает *скрытое от пользо- вателя взаимоисключение*, во время помещения сообщения в очередь или его полу- чения из очереди. Она также обеспечивает:

* + *блокировку процесса*, при попытке выполнить примитив *receive()* над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного

типа;

* + *блокировку процесса*, при попытке выполнить примитив *send()* для очереди, в которой нет свободного места.

**Замечание** Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаи- модействие процессов, не находящихся одновременно в вычислительной системе.

Общий набор примитивов передачи сообщений представлен четырьмя сис- темными вызовами:

#include <types.h> #include <ipc.h> #include <msg.h>

*int* msgget(*key\_t* key, *int* msgflg);

*int* msgsnd(*int* msqid, *struct msgbuf* \*ptr, *int* length, *int* flag);

*int* msgrcv(*int* msqid, *struct msgbuf* \*ptr, *int* length, *long* type, *int* flag);

*int* msgctl(*int* msqid, *int* cmd, *struct msqid\_ds* \*buf);

Системный вызов *msgget()* предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди: *целое неотрицательное число*, однозначно характе- ризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней.

Здесь параметр *key* - ключ System V IPC для очереди сообщений.

**Замечание**

В качестве значения этого параметра может быть использовано значение ключа, полу-

ченное с помощью функции *ftok()*, или специальное значение *IPC\_PRIVATE*, исполь- зование которого всегда приводит к попытке создания новой очереди сообщений с идентификатором, не совпадающим ни с одной из уже существующих очередей.

Параметр *msgflg* – имеет значение только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией, с помощью операции побитовое или – " | ", следующих предопределенных значений и восьмеричных прав доступа:

* + *IPC\_CREAT* — если очереди для указанного ключа не существует, она долж- на быть создана;
  + *IPC\_EXCL* — применяется совместно с флагом IPC\_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к

очереди не производится и констатируется ошибочная ситуация, при этом переменная ***errno***, описанная в файле *<errno.h>*, примет значение *EEXIST*;

* + 0400 — разрешено чтение для пользователя, создавшего очередь;
  + 0200 — разрешена запись для пользователя, создавшего очередь;
  + 0040 — разрешено чтение для группы пользователя, создавшего очередь;
  + 0020 — разрешена запись для группы пользователя, создавшего очередь;
  + 0004 — разрешено чтение для всех остальных пользователей;
  + 0002 — разрешена запись для всех остальных пользователей.

**Замечание**

Очередь сообщений имеет ограничение по общему количеству хранимой информации, которое может быть изменено администратором системы, а текущее значение ограни- чения можно узнать с помощью команды:

**ipcs -l**

Системный вызов *msgget()* возвращает:

* + *значение дескриптора System V IPC* для очереди сообщений, при нормальном завершении;
  + *значение -1*, при возникновении ошибки.

Системный вызов *msgsnd()* предназначен для помещения сообщения в оче- редь сообщений.

*int* msgsnd(*int* msqid, *struct msgbuf* \*ptr, *int* length, *int* flag);

Параметр *msqid* является *дескриптором System V IPC* для очереди, в которую отправляется сообщение.

Непосредственно передаеются данные структуры типа *struct msgbuf*, которая

описана в файле *<sys/msg.h>* как

*struct* msgbuf {

*long* mtype;

*char* mtext[1];

};

где *mtype* - имеет тип *long*, интерпретируемый как *тип сообщения*; должен быть

*строго положительной величиной*.

*mtext[...]* - массив байт передаваемого сообщения; в **Linux**, назмер этого массива ограничен размером *8192 байт* и может быть еще уменьшен системным адми- нистратором.

### Например:

*struct* mymsgbuf {

*long* mtype;

*char* mtext[1024];

} mybuf;

**Более того**, информация вовсе не обязана быть текстовой, например:

*struct* mymsgbuf {

*long* mtype;

*struct* {

*int* iinfo;

*float* finfo;

} info;

} mybuf;

Параметр *length* — действительная длинна полезной информации, которая следует за типом сообщения, в структуре *msgbuf*.

Параметр *flag* может принимать два значения: *0* и *IPC\_NOWAIT*:

* + если значение флага равно *0*, и в очереди не хватает места для того, чтобы поместить сообщение, то системный вызов блокируется до тех пор, пока не освободится место;
  + при значении флага *IPC\_NOWAIT* системный вызов в этой ситуации не блокируется, а констатируется возникновение ошибки с установлением зна- чения переменной *errno*, описанной в файле *<errno.h>*, равным *EAGAIN*.

Системный вызов *msgsnd()* возвращает:

* + *значение 0*, при нормальном завершении;
  + *значение -1*, при возникновении ошибки.

Системный вызов *msgrcv()* предназначен для получения сообщения из очере- ди сообщений.

*int* msgrcv(*int* msqid, *struct msgbuf* \*ptr, *int* length, *long* type, *int* flag);

где параметр *msqid* - дескриптор System V IPC очереди, из которой извлекается сообщение;

*type* - способ выборки сообщения из очереди:

* + ***0*** - в порядке FIFO, независимо от типа сообщения;
  + ***n*** - в порядке FIFO, для сообщений с типом ***n***;
  + ***-n*** - первым выбирается сообщение с минимальным типом, не превышающим значения ***n***, пришедшее ранее всех других сообщений с тем же типом.

*length* - максимальная длина полезной части информации, расположенной в струк- туре после типа сообщения, которая может быть размещена в сообщении.

Параметр **flag** может принимать значение ***0*** или быть какой-либо комбинацией флагов *IPC\_NOWAIT* и *MSG\_NOERROR*:

* + Если флаг *IPC\_NOWAIT* не установлен и очередь сообщений пуста или в ней нет сообщений с заказанным типом, то *системный вызов блокируется до*

*появления запрошенного сообщения*.

* + При установлении флага *IPC\_NOWAIT* системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной ***errno***, описанной в файле *<errno.h>*, равным *EAGAIN*.
  + Если действительная длина полезной части информации в выбранном сообщении превышает значение, указанное в параметре *length* и флаг *MSG\_NOERROR* не установлен, то выборка сообщения не производится, и

фиксируется наличие ошибочной ситуации.

* + Если флаг *MSG\_NOERROR* установлен, то в этом случае ошибки не возника- ет, а сообщение копируется в сокращенном виде.

В случае удачи, системный вызов *msgrcv()* копирует выбранное сообщение из очереди сообщений по адресу, указанному в параметре *ptr*, одновременно удаляя его из очереди сообщений.

Системный вызов *msgrcv()* возвращает:

* + *действительную длину полезной части информации*, при нормальном завер- шении;
  + *значение -1*, при возникновении ошибки.

Системный вызов *msgctl()* предназначен для получения информации об оче- реди сообщений, изменения ее атрибутов и удаления из системы.

*int* msgctl(*int* msqid, *int* cmd, *struct msqid\_ds* \*buf);

где *msqid* - дескриптор System V IPC, которое вернул системный вызов *msgget()* при создании очереди или при ее поиске по ключу;

*cmd* используется как команда (см. руководство ***man***).

**Замечание** *IPC\_RMID* – команда для удаления очереди сообщений с заданным идентификатором. Параметр *buf* для этой команды не используется, поэтому мы всегда будем подставлять туда значение *NULL*.

Системный вызов *msgctl()* возвращает:

* + *значение 0*, при нормальном завершении;
  + *значение -1*, при возникновении ошибки.

**Рассморим** практическое применение очередей сообщений, на примере рабо- ты двух программ:

* + *программа lab11.5*, показанная на листинге 1.5, *пишет в очередь ряд сооб- щений* с типом *0* и заканчивает запись в очередь сообщением с типом 255;
  + *программа lab11.6*, показанная на листинге 1.6, *читает из очереди все сооб- щения и печатает их*; чтение сообщений заканчивается, когда получено сообщение с типом 255;

**Замечание** Программы работы с очередями сообщений имеют только внутренние средства синх- ронизации, связанные с целостностью операций записи в очередь и чтения из нее. Оче- видно, что многие задачи могут потребовать дополнительных средств для синхрониза- ции их критических областей. В таком случае, следует дополнительно воспользоваться инструментом семафоров.