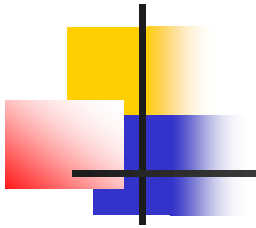
Синхронизация процессов

Проблема синхронизации процессов возникает в основном в следствие необходимости совместного использования ресурсов вычислительной системы. Для правильности работы процессов, совместно использующих одни и те же ресурсы, требуется согласованность действий и координация.

Синхронизация процессов

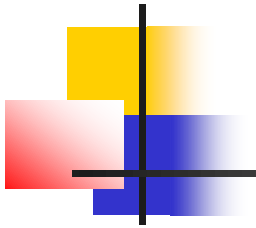
В некоторых случаях операционная система производит принудительную координацию процессов, пользующихся критическими (имеющимися в недостаточном количестве) ресурсами. Например, процесс должен ждать, когда канал ввода-вывода, к которому он обращается, занят. В других случаях координация необходима нескольким процессам, порожденным для выполнения одного задания.

Синхронизация процессов



Существуют две проблемы синхронизации, связанные с распределением процессоров и общением между процессами, - состязания (или гонки) и клинчи (или тупики).

Команда ПРОВЕРИТЬ и УСТАНОВИТЬ

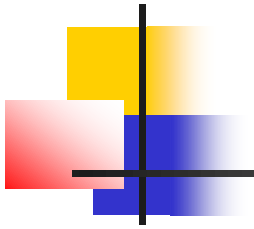


В большинстве случаев для целей синхронизации с ресурсом связывают некоторый объект, являющийся как бы замком к ресурсу. Такой объект чаще всего называют ***байтом блокировки*** или ***семафором.***

Команда ПРОВЕРИТЬ и УСТАНОВИТЬ

Для каждого ресурса (будь то устройство или данные) должен существовать свой байт блокировки. Договоримся, что значение байта блокировки 0 соответствует доступности ресурса; если же он равен 1, то ресурс занят.

Команда ПРОВЕРИТЬ и УСТАНОВИТЬ



Перед обращением к ресурсу процесс должен выполнить следующие шаги:

1) Проверить значение байта блокировки (0 или 1).

2) Установить байт блокировки в 1.

3) Если первоначальное значение байта блокировки было равно 1, вернуться к шагу 1.

Команда ПРОВЕРИТЬ и УСТАНОВИТЬ

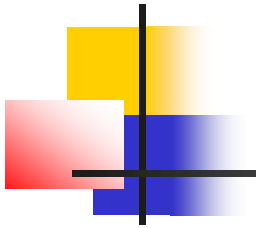
После завершения использования ресурса процесс должен установить байт блокировки в 0.

Ни один из других процессов не должен иметь возможности изменить значения байта блокировки в промежутке между шагами 1 и 2.

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ

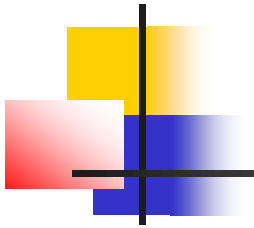
Рассмотренный выше метод обеспечивает синхронизацию процессов, но является весьма неэффективным с точки зрения использования процессоров. Для того, чтобы не пропустить момента снятия блокировки, процессор вынужден циклиться на проверке байта блокировки до тех пор, пока этот байт не будет переведен в нулевое состояние, и тем самым не может использоваться для выполнения какой бы то ни было "полезной" работы.

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ



Неразумно тратить такой важный ресурс, как процессор, на обслуживание заблокированного процесса. Можно модифицировать механизмы захвата и освобождения следующим образом:

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ



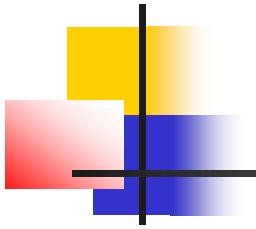
ЗАНЯТЬ(X):

1) Проверить значение байта блокировки (0 или 1).

2) Установить байт блокировки в 1.

3) Если первоначально значение байта блокировки было равно 1, выполнить функцию ОЖИДАНИЕ(X).

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ

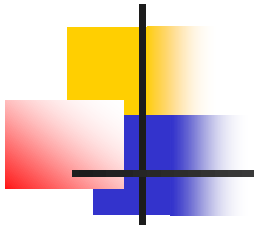


ОСВОБОДИТЬ(X):

1) Установить байт блокировки в 0.

2) Выполнить функцию ОПОВЕЩЕНИЕ(X).

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ



Операции ОЖИДАНИЕ (WAIT) и ОПОВЕЩЕНИЕ (SIGNAL) являются примитивами регулировщика операционной системы. **ОЖИДАНИЕ(Х) блокирует обслуживание процесса, делает отметку об этом в РСВ и связывает его с байтом блокировки(Х).**

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ

Планировщик процессов находит другой готовый процесс и переводит освободившийся процессор на обслуживание этого нового процесса. ОПОВЕЩЕНИЕ(X) просматривает связанный с байтом X список блокированных процессов.

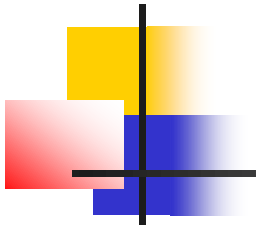
Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ

Если в списке есть процессы, ожидающие освобождения ресурса X, один из них переводится в состояние готовности, а в его PCB делается соответствующая отметка. С этого момента "пробудившийся" процесс вновь становится доступным для выборки его планировщиком процессов, т. е. равноправным претендентом на получение процессорного времени.

Механизмы ОЖИДАНИЕ и ОПОВЕЩЕНИЕ

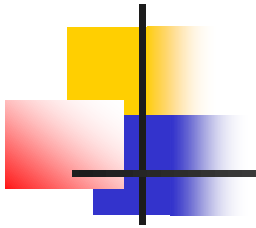
ОЖИДАНИЕ и ОПОВЕЩЕНИЕ могут служить и для иных целей, таких, как синхронизация с процессами ввода-вывода. ***После выдачи запроса на ввод-вывод процесс может "подождать" завершения обмена, используя операцию ОЖИДАНИЕ(X), где X - байт состояния устройства*** Прерывание, свидетельствующее о завершении ввода-вывода, вызывает выполнение системой операции ОПОВЕЩЕНИЕ(X).

События. Управление событиями.



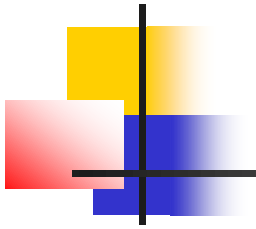
Событие **– это не зависящая от времени операция, производимая внутри ОС.**

События. Управление событиями.



События могут непосредственно или косвенно инициироваться пользователем. Н-р, при запросе на чтение событие инициируется непосредственно. Но запрос на чтение может сам косвенно породить и другие события, включая генерирование прерываний. После окончания события инициатор события должен быть информирован о результате завершения операции (успешном или неуспешном***).***

События. Управление событиями.

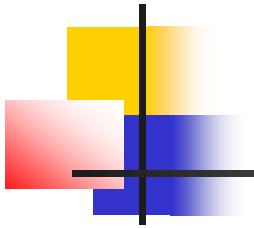


В ОС допускается одновременное возникновение нескольких событий. Каждому событию присваивается числовой идентификатор, называемый флагом события.

События. Управление событиями.

Число флагов события задается проектировщиком системы. Каждому флагу необходимо определенное пространство памяти для хранения статуса и данных. Число флагов зависит от величины управляющих блоков и максимально возможного числа одновременно реализующихся событий.

События. Управление событиями.

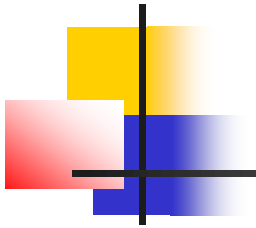


В приведенном примере флаги представляют собой целые числа в интервале от 0 до 16; они интерпретируются следующим образом:

События. Управление событиями.

Флаг события	Примечание
0	Созданный процесс будет выполняться независимо от своего прародителя; синхронизация невозможна
1,…15	Возможна нормальная параллельная синхронизация двух процессов, при которой процесс-родитель получает в соответствующее время информацию от порожденного им процесса.
16	Процесс родитель приостановлен до момента завершения операции процессом-потомком, обеспечивается полная синхронизация

События. Управление событиями.

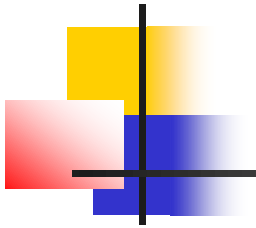


С каждым событием (за исключением нулевого флага) связано статусное слово и числовое значение. Статусное слово в любой момент времени доступно для чтения. Тем самым постоянно поддерживается информация о ходе события. Статусное слово может принимать три значения:

События. Управление событиями.

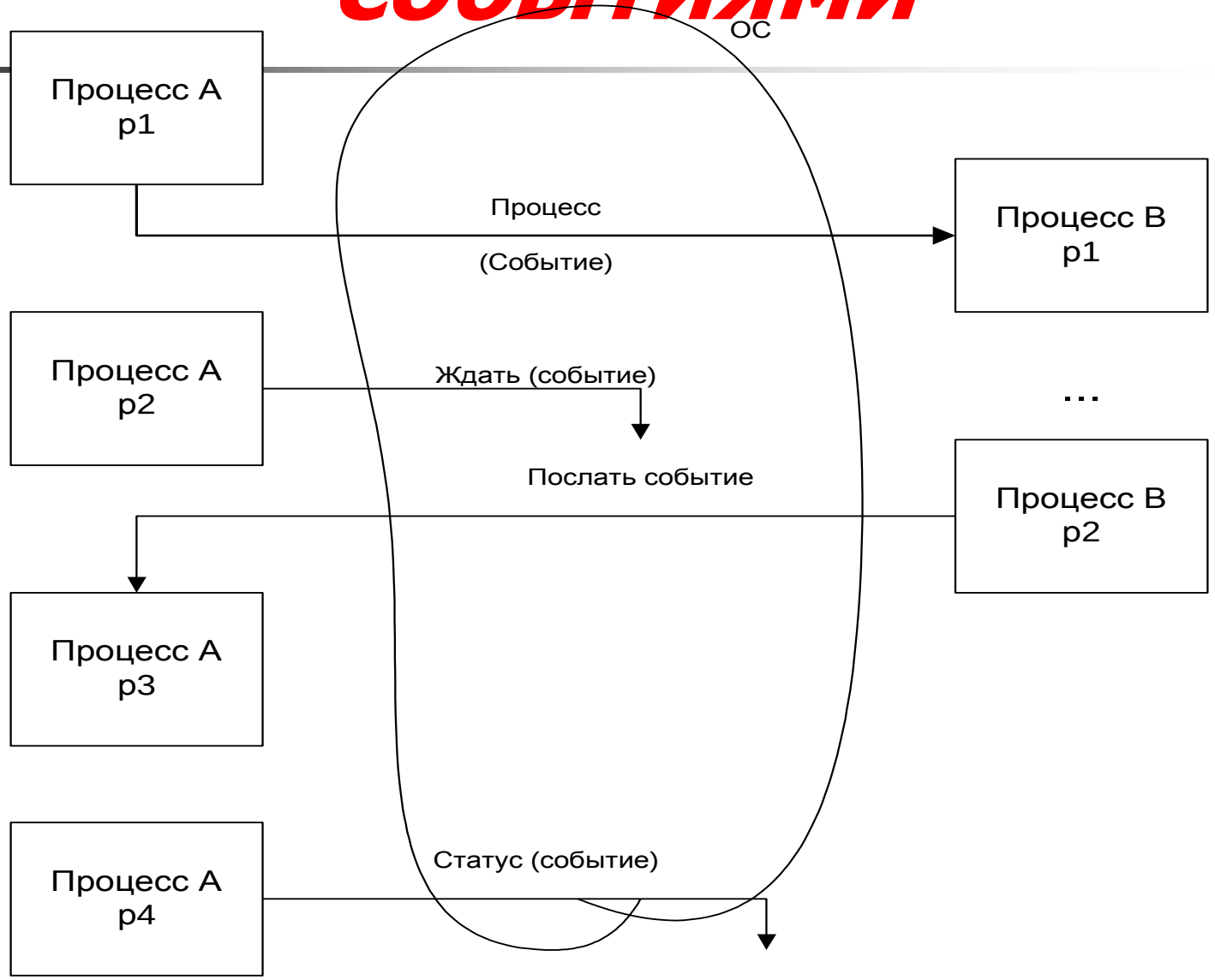
Статус	Обозначение
<0	Событие прекращено из-за ошибки; статусное слово содержит системный код ошибки
0	Событие продолжается
>0	Событие благополучно завершилось; статусное слово может при этом содержать дополнительную информацию

События. Управление событиями.



В дополнение к сигналу «выполняться- не выполняться» подчиненный процесс может возвратить одно или два слова данных.

Операции управления событиями



Операции управления событиями

Для мотивирования операций управления событиями опишем последовательность действий по синхронизации двух процессов:

1 Процесс А инициирует процесс В с помощью события 1.

2 Процесс затем выполняет ЖДАТЬ(1) до тех пор, пока В не закончит работу.

3 Процесс В выполняет назначенные ему действия.

Операции управления событиями

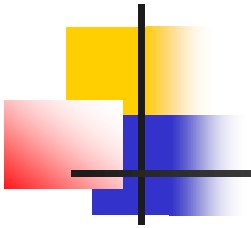
4 После завершения операции процесс В посылает статус и значение для события 1.

5 Система возобновляет работу процесса А после пересылки информации о завершении события.

6 Процесс А читает статус и значение события.

Каждая операция инициируется запросом к системной службе.

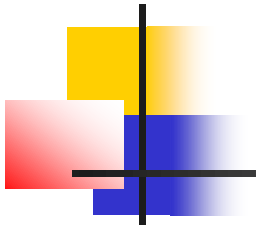
Средства взаимодействия между процессами



Для обеспечения взаимодействия при выполнении общих задач в микропроцессорной системе процессы должны иметь возможность обмениваться данными.

Рассмотрим простой пример с двумя процессами П1 и П2.

Средства взаимодействия между процессами



П1 создает и пересылает процессу П2 последовательность данных, называемую сообщением. П2 получает и обрабатывает сообщение.

Средства взаимодействия между процессами

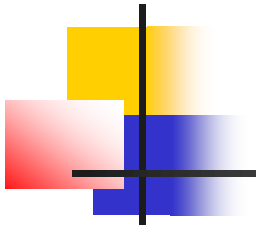
Асинхронность выполнения процессов усложняет их взаимодействие.

Асинхронность предполагает выполнение процесса со скоростью, не зависящей от других процессов.

Средства взаимодействия между процессами

Предположим П1 выдал сообщение до того момента, как процесс П2 оказался в состоянии их обработать. Задержка П1 до момента готовности П2 предполагает информированность П1 о статусе П2. Последнее нарушает принцип асинхронности. Чтобы не задерживать П1, создается временный буфер для хранения сообщений до того момента, когда П2 будет в состоянии их получить.

Средства взаимодействия между процессами



Обмен между процессами может быть разделен на два класса:

1) разделяемые (совместно используемые) переменные и

2) сообщения.

Средства взаимодействия между процессами

Подход, использующий разделяемые переменные, поясняется примерами с семафорами и событиями.

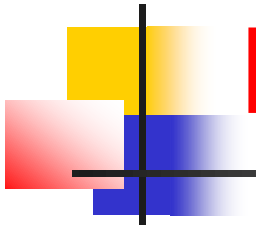
Его серьезным недостатком является ограниченный объем передаваемой информации – одно или несколько слов.

Средства взаимодействия между процессами

Обмен информацией между процессами имеет два ограничения со стороны ресурсов:

1) Объем передаваемых процессом сообщений не должен превышать емкости отведенного под них буфера

2) Принимающий процесс не может обрабатывать сообщения быстрее, чем они создаются передающим процессом.

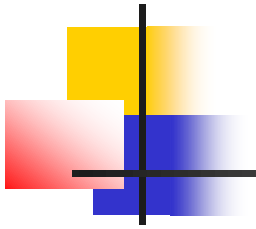


Базовые принципы

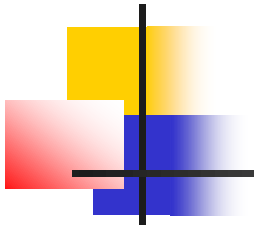
Очереди сообщений представляют собой связный список в адресном пространстве ядра.

Сообщения могут посылаться в очередь по порядку и доставляться из очереди несколькими разными путями. Каждая очередь сообщений однозначно определена идентификатором IPC.

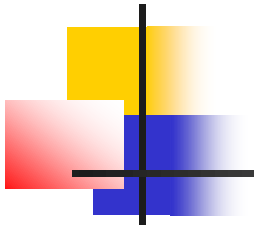
Буфер сообщения



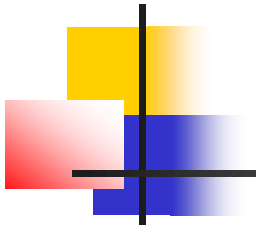
Первая структура, которую мы рассмотрим, msgbuf . Его описание находится в <linux/msg.h>:



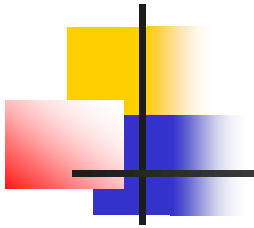
```
/* буфер сообщения для вызовов msgsnd и msgrcv*/
    struct msgbuf {
            long mtype; /* тип сообщения */
            char mtext[1]; /* текст сообщения */
};
```



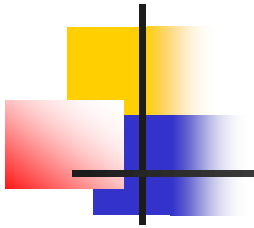
- mtype - Тип сообщения, представленный натуральным числом. Он обязан быть натуральным!
- mtext - Собственно сообщение.



Ядро хранит сообщение в очереди структуры msg. Она определена в <linux/msg.h> следующим образом:

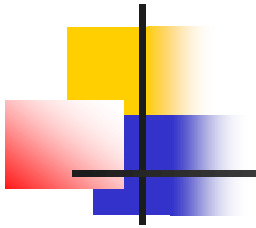


```
struct msg {
    struct msg *msg_next;
                /* следующее сообщение в очереди */
    long msg_type;
    char *msg_spot; /* адрес текста сообщения */
    short msg_ts; /* размер текста */
};
```



- msg_next - Указатель на следующее сообщение в очереди. Сообщения объединены в односвязный список и находятся в адресном пространстве ядра.
- msg_type - Тип сообщения, каким он был объявлен в msgbuf .
- msg_spot - Указатель на начало тела сообщения.
- msg_ts - Длина текста (или тела) сообщения.

Структура msqid_ds ядра

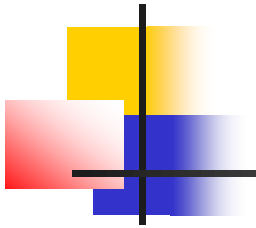


Каждый из трех типов IPC-объектов имеет внутреннее представление, которое поддерживается ядром. Для очередей сообщений это структура msqid_ds . **Ядро создает, хранит и сопровождает образец такой структуры для каждой очереди сообщений в системе. Она определена в <linux/msg.h> следующим образом:**

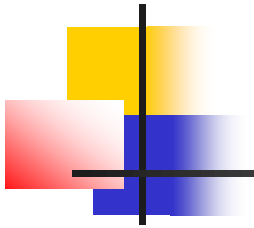
```
/* структура msqid для каждой очереди в системе */
struct msqid_ds {
   struct ipc_perm msg_perm;
   struct msg *msg_first;     /* первое сообщение в очереди */
   struct msg *msg_last;      /* последнее сообщение в очереди */
   time_t msg_stime;          /* время последнего вызова msgsnd */
   time_t msg_rtime;          /* время последнего вызова msgrcv */
   time_t msg_ctime;          /* время последнего изменения */
   struct wait_queue *wwait;
   struct wait_queue *rwait;
   ushort msg_cbytes;
   ushort msg_qnum;
   ushort msg_qbytes;         /* максимальное число байтов на очередь */
   ushort msg_lspid;          /* pid последнего испустившего msgsnd */
   ushort msg_lrpid;          /* последний полученный pid */
};
```

- msg_perm - Экземпляр структуры ipc_perm , определенной в <linux/ipc.h>. Она содержит информацию о доступе для очереди сообщений, включая права доступа и информацию о создателе сообщения (uid и т.п.).
- msg_first - Ссылка на первое сообщение в очереди (голова списка).
- msg_last - Ссылка на последний элемент списка (хвост списка).
- msg_stime - Момент времени посылки последнего сообщения из очереди.
- msg_rtime - Момент времени последнего изъятия элемента из очереди.
- msg_ctime - Момент времени последнего изменения, проделанного в очереди (подробнее об этом позже).
- wwait и rwait - Указатели в очередь ожидания ядра. Они используются, когда операция над очередью сообщений переводит процесс в состояние спячки (то есть очередь переполнена, и процесс ждет открытия).
- msg_cbytes - Число байт, стоящих в очереди (суммарный размер всех сообщений).
- msg_qnum - Количество сообщений в очереди на настоящий момент.
- msg_qbytes - Максимальный размер очереди.
- msg_lspid - PID процесса, пославшего последнее в очереди сообщение.
- msg_lrpid - PID последнего процесса, взявшего из очереди сообщение.

Структура ipc_perm ядра



Информацию о доступе к IPC-объектам ядро хранит в структуре ipc_perm . Например, описанная выше структура очереди сообщений содержит одну структуру типа ipc_perm в качестве элемента. Следующее ее определение дано в <linux/ipc.h>.



```
struct ipc_perm {
   key_t  key;
   ushort uid;    /* euid и egid владельца */
   ushort gid;   ushort cuid;   /* euid и egid создателя */
   ushort cgid;
   ushort mode;   /* режим доступа, см. режимные флаги ниже */
   ushort seq;    /* порядковый номер использования гнезда */
};
```

Средства взаимодействия между процессами

С сообщениями работают четыре системные функции:

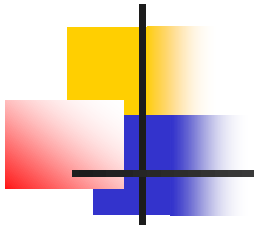
1) msgget, которая возвращает и может создавать дескриптор сообщения, определяющий очередь сообщений и используемый другими системными функциями

Средства взаимодействия между процессами

2) msgctl, которая устанавливает и возвращает связанные с дескриптором сообщений параметры или удаляет дескрипторы,

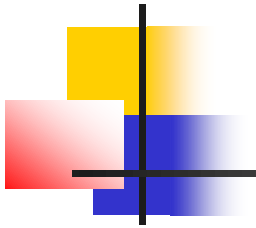
3) msgsnd, которая посылает сообщение

Средства взаимодействия между процессами



4) msgrcv, которая получает сообщение.

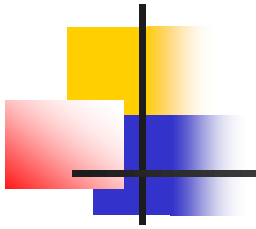
Средства взаимодействия между процессами



Синтаксис вызова системной функции msgget:

msgqid=msgget(key, flag);

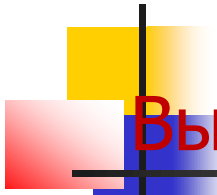
msgqid – возвращаемый функцией дескриптор



RETURNS: идентификатор очереди сообщений в случае успеха; -1 в случае ошибки.

Первый аргумент msgget() значение ключа (мы его получаем при помощи ftok()). Этот ключ сравнивается с ключами уже существующих в ядре очередей. При этом операция открытия или доступа к очереди зависит от содержимого аргумента msgflg :

- IPC_CREAT - Создает очередь, если она не была создана ранее.
- IPC_EXCL - При использовании совместно с IPC_CREAT , приводит к неудаче если очередь уже существует.



Вызов msgget() с IPC_CREAT , но без IPC_EXCL всегда выдает идентификатор (существующей с таким ключом или созданной) очереди.

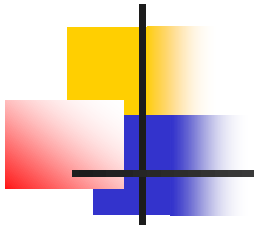
Использование IPC_EXCL вместе с IPC_CREAT либо создает новую очередь, либо, если очередь уже существует, заканчивается неудачей.

Самостоятельно IPC_EXCL бесполезен, но вместе с IPC_CREAT он дает гарантию, что ни одна из существующих очередей не открывается для доступа.

Средства взаимодействия между процессами

Когда процесс вызывает функцию msgget для того, чтобы создать новый дескриптор, ядро просматривает массив очередей сообщений в поисках существующей очереди с указанным идентификатором. Если такой очереди нет, ядро выделяет новую очередь, инициализирует ее и возвращает идентификатор процессу.

Средства взаимодействия между процессами



Ядро хранит сообщения в очереди сообщений, определяемой значением дескриптора, и использует значение msgqid в качестве указателя на массив заголовков очередей.

Средства взаимодействия между процессами

Выводы:

заголовок очереди содержит следующие поля:

- Указатели на первое и последнее сообщение в списке;

- Количество сообщений и общий объем информации в списке в байтах;

Средства взаимодействия между процессами

- Максимальная емкость списка в байтах;
- Идентификаторы процессов, пославших и принявших сообщения последними;
- Поля, указывающие время последнего выполнения функций msgsnd, msgrcv, msgctl.

Средства взаимодействия между процессами

Для отправки сообщения процесс использует системную функцию **msgsnd:**

msgsnd(msgqid, msg, count, flag);

Где **msgqid** – дескриптор очереди сообщений, обычно возвращаемый функцией msgget,

Средства взаимодействия между процессами

- msg – указатель на структуру, состоящую из типа в виде назначаемого пользователем целого числа и массива символов,

- count – размер информационного массива,

- flag – действие, предпринимаемое ядром в случае переполнения внутреннего буферного пространства.

Средства взаимодействия между процессами

Ядро проверяет:

1) имеется ли у посылающего сообщение процесса разрешение на запись по указанному дескриптору,

2) не выходит ли размер сообщения за установленную системой границу,

3) не содержится ли в очереди слишком большого объема информации,

4) а также является ли тип сообщения положительным целым числом.

Средства взаимодействия между процессами

Если условия все соблюдены, ядро выделяет сообщению место, используя карту сообщений, и копирует в это место данные. К сообщению присоединяется заголовок, после чего оно помещается в конец связного списка заголовков сообщений.

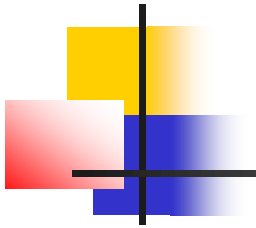
Средства взаимодействия между процессами

В заголовке сообщения записывается тип и размер сообщения, устанавливается указатель на текст сообщения и производится корректировка содержимого различных полей заголовков в очереди, содержащих статистическую информацию (количество сообщений в очереди и их суммарный объем в байтах, время последнего выполнения операций и идентификатор процесса пославшего сообщение).

Средства взаимодействия между процессами

Затем ядро выводит из состояния ожидания все процессы, ожидающие пополнения очереди сообщений. Если размер очереди в байтах превышает границу допустимости, процесс приостанавливается до тех пор, пока другие сообщения не уйдут из очереди. Но если процессу было указание не ждать (соответствующие установки флага), он немедленно возвращает управление с уведомлением об ошибке.

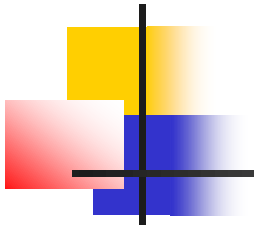
Средства взаимодействия между процессами



Процесс **получает сообщения, вызывая функцию msgrcv** по следующему формату:

count=msgrcv(id, msg, maxcount, type, flag);

Средства взаимодействия между процессами



Где id – дескриптор сообщения,

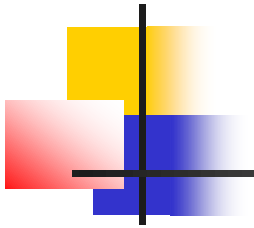
msg – адрес структуры процесса, которая будет содержать полученное сообщение,

maxcount – размер структуры msg,

type - тип считываемого сообщения,

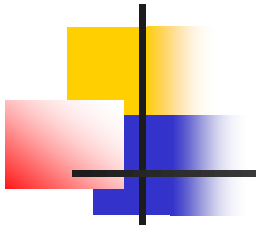
flag – действие, предпринимаемое ядром в том случае, если в очереди сообщений нет.

Средства взаимодействия между процессами

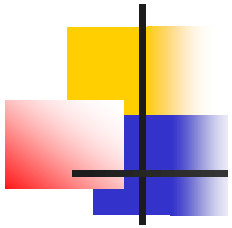


В переменной count процессу возвращается число прочитанных байт сообщения.

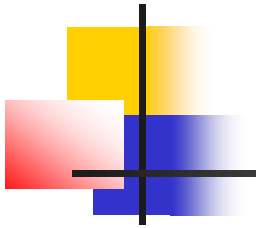
Процесс может получать сообщения определенного типа, если присвоить параметру type соответствующее значение.



Алгоритм взаимодействия между процессами средствами сообщений

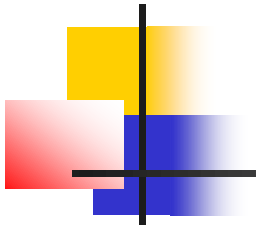


Алгоритм взаимодействия между процессами:
Алгоритм msgsnd /*послать сообщение*/
Входная информация (1) дескриптор очереди сообщений
(2) адрес структуры сообщения
(3) размер сообщения
(4) флаги
выходная информация: количество посланных байт
{
проверить правильность указания дескриптора и наличие соответствующих прав доступа;
выполнить пока (для хранения сообщения не будет выделено место
{
Если (флаги не разрешают ждать)
Вернуться;
Приостановиться (до тех пор, пока место не освободится);
}
Получить заголовок сообщения;
Считать заголовок сообщения из пространства задачи в пространство ядра;
Настроить структуры данных: выстроить очередь заголовков сообщений, установить в заголовке указатель на текст сообщения, заполнить поля, содержащие счетчики, время последнего выполнения операций и идентификатор процесса;
Вывести из состояния приостановки все процессы, ожидающие разрешения считать сообщение из очереди;
}



Семафоры

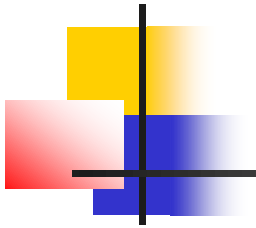
Ключевым принципом, характеризующим современные ОС, является использование концепции семафора для управления синхронизацией взаимодействующих процессов. Эта концепция была впервые предложена в начале 60-х годов Дейкстрой.



Семафоры

Дейкстра описывает семафоры как неотрицательные целые переменные.

Для семафоров определены 2 операции:



Семафоры

P – уменьшение семафора на единицу, если возможно. Если семафор равен нулю, он не может быть уменьшен, и процесс вызвавший операцию, ждет пока уменьшение станет возможным.

Отображаемая память

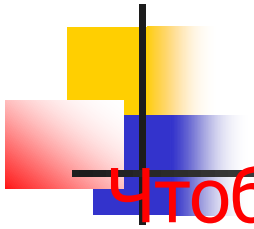
Отображаемая память позволяет различным процессам общаться через общедоступный файл.

Отображаемая память может использоваться для взаимодействия процессов или как простой способ для обращения к содержимому файла. Отображаемая память формирует ассоциацию между файлом и памятью процесса.

Отображаемая память

Linux разбивает файл на фрагменты размером страницы и затем копирует их в страницы виртуальной памяти так, чтобы они могли быть представлены в адресном пространстве процесса.

Таким образом, процесс может читать содержание файла обычным доступом к памяти. Он может также изменить содержимое файла, записывая в память. Что позволяет быстро взаимодействовать с файлом.



Отображаемая память

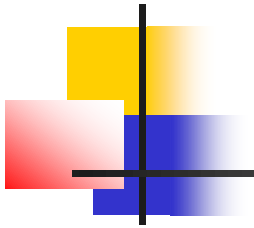
Чтобы отобразить обычный файл в память процесса используйте вызов mmap (Memory MAPped).

Прототип системного вызова

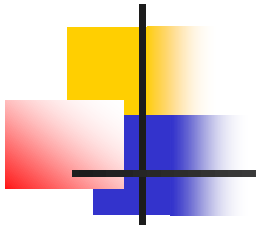
```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h> void *mmap
(void *start, size_t length, int prot, int
flags, int fd, off_t offset);
```

Описание системного вызова

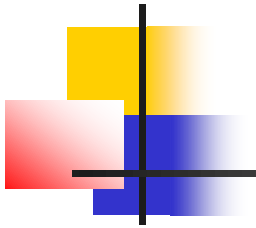
Системный вызов **mmap** служит для отображения предварительно открытого файла (например, с помощью системного вызовава open () в адресное пространство вычислительной системы. После его выполнения файл может быть закрыт (например, системным вызовом close(), что никак не повлияет на дальнейшую работу с отображенным файлом.



Параметр **fd** является файловым дескриптором для файла, который мы хотим отобразить в адресное пространство (т.е. значением, которое вернул системный вызов open()).

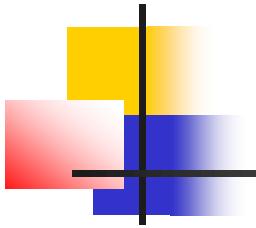


Ненулевое значение параметра addr может использоваться только очень квалифицированными системными программистами, поэтому мы на наших занятиях будем всегда полагать его равным значению **NULL**, позволяя операционной системе самой выбрать начало области адресного пространства, в которую будет отображен файл.

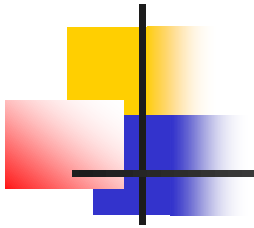


В память будет отображаться часть файла, начиная с позиции внутри его, заданной значением параметра **offset** - смещение от начала файла в байтах, и длиной равной значению параметра **length** (естественно, тоже в байтах).

Значение параметра **length** можно указать и существенно большим, чем реальная длина от позиции **offset** до конца существующего файла. На поведении системного вызова это никак не отразится, но в дальнейшем при попытке доступа к ячейкам памяти, лежащим вне границ реального файла, возникнет сигнал **SIGBUS** (реакция на него по умолчанию - прекращение процесса с образованием core файла).

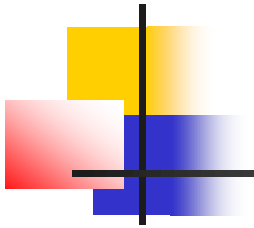


Параметр **flags** определяет способ отображения файла в адресное пространство.

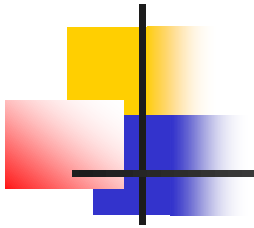


Значение флажка :

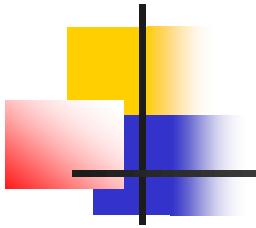
- MAP_FIXED - отображать только начиная с указанного адреса, если не удастся - вернуть ошибку, а не искать подходящий. Этот адрес должен быть выровнен на границу страницы.



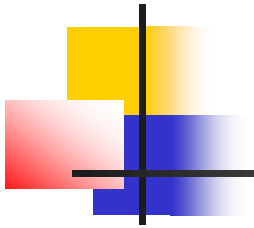
- MAP_PRIVATE - производить запись изменений не в отображенный файл, а в его копию. Никакой другой процесс не увидит, изменений в файле. Этот режим не может использоваться с MAP_SHARED.



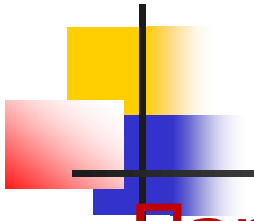
- MAP_SHARED - изменения в памяти немедленно отражаются в основном файле вместо буферизации. Используйте этот режим для межпроцессного взаимодействия. Не должно быть использовано вместе с MAP_PRIVATE.



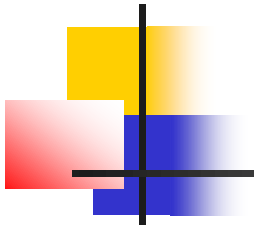
Если в качестве его значения выбрано **MAP_SHARED**, то полученное отображение файла потом будет использоваться и другими процессами, вызвавшими **mmap** для этого файла с аналогичными значениями параметров, а все изменения, сделанные в отображенном файле, будут сохранены во вторичной памяти.



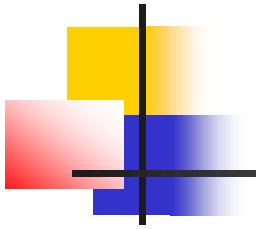
Если в качестве значения параметра **flags** указано **MAP_PRIVATE**, то процесс получает отображение файла в свое монопольное распоряжение, но все изменения в нем не могут быть занесены во вторичную память (т.е., проще говоря, не сохранятся).



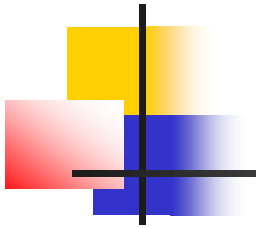
Параметр **prot** определяет разрешенные операции над областью памяти, в которую будет отображен файл. В качестве его значения мы будем использовать значения **PROT_READ** (разрешено чтение), **PROT_WRITE** (разрешена запись) или их комбинацию через операцию "побитовое или" - "|".



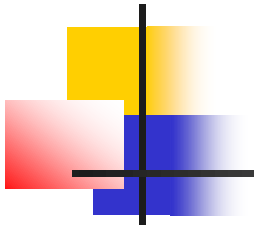
Необходимо отметить особенность системного вызова, связанную с этим параметром:



- Значение параметра prot не может быть шире, чем операции над файлом, заявленные при его открытии в параметре **flags** системного вызова open(). Например, нельзя открыть файл только для чтения, а при его отображении в память использовать значение **prot = PROT_READ | PROT_WRITE**.



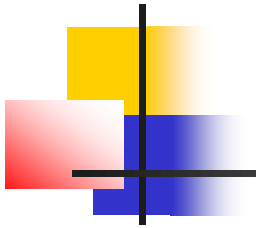
При нормальном завершении системный вызов возвращает начальный адрес области памяти, в которую отображен файл (или его часть), при возникновении ошибки - специальное значение **MAP_FAILED** .



Отображаемая память

Когда вы закончили работу с управлением памятью, освободите ее при помощи **munmap.**

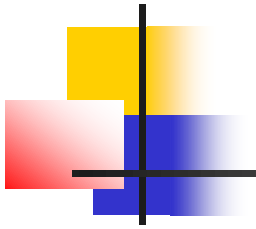
Передайте адрес начала и длину области отображаемой памяти. Linux автоматически освобождает отображаемую память при завершении процесса.



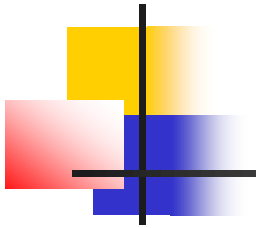
Прототип системного вызова

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h> int
munmap (void *start, size_t
length);
```

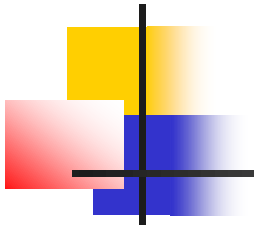
Описание системного вызова



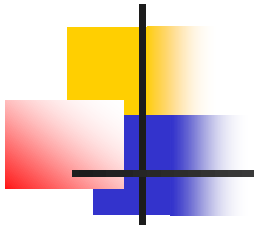
Системный вызов **munmap** служит для прекращения отображения memory mapped файла в адресное пространство вычислительной системы.



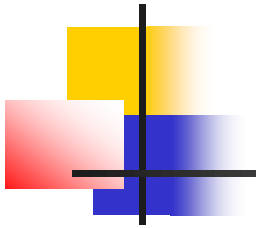
Если при системном вызове mmap() было задано значение параметра **flags** равное **MAP_SHARED** и в отображении файла была разрешена операция записи (в параметре **prot** использовалось значение **PROT_WRITE**), то **munmap** синхронизирует содержимое отображения с содержимым файла во вторичной памяти.



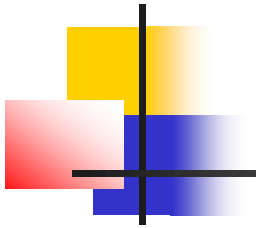
После его выполнения области памяти, использованные для отображения файла, становятся недоступны текущему процессу.



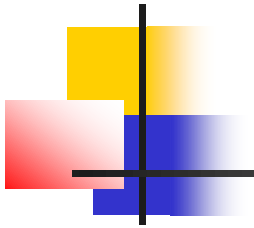
Параметр **addr** является адресом начала области памяти, выделенной для отображения файла, т.е. значением, которое вернул системный вызов mmap().



Параметр **length** определяет ее длину и его значение должно совпадать со значением соответствующего параметра в системном вызове mmap().



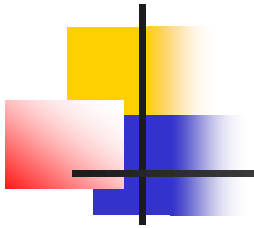
При нормальном завершении системный вызов возвращает значение 0, при возникновении ошибки - значение -1.



Отображаемая память

Совместный доступ к файлу

Различные процессы могут взаимодействовать используя области отображаемой памятью, связанные с одним и тем же файлом. Укажите флажок MAP_SHARED для того, чтобы любые операции записи в область памяти немедленно передаются файлу и видимым другим процессам. Если Вы не определяете этот флажок, Linux может буферизовать операции записи перед передачей их к файлу.

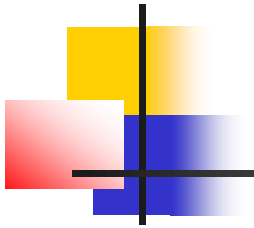


msync - синхронизирует содержимое файла с его отражением в памяти

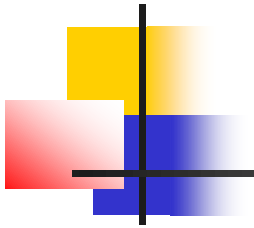
http://www.opennet.ru/man.shtml?topic=msync&category=2&russian=0

СИНТАКСИС

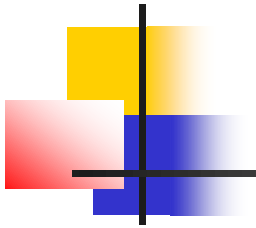
```
#include <unistd.h>
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES
#ifdef _POSIX_SYNCHRONIZED_IO

int msync(void *start, size_t length, int flags);

#endif
#endif
```



msync записывает на диск изменения, внесенные в файл, отраженный в память при помощи функции **mmap**. Если не использовать эту функцию, то нет никакой гарантии, что изменения будут записаны в файл до вызова **munmap**.

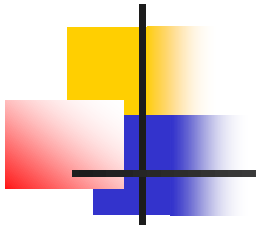


На диск записывается часть файла, начинающаяся в памяти с адреса *start* длиной *length*.



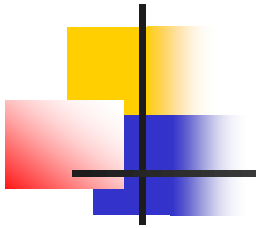
Параметр *flags* состоит из комбинации битов MS_ASYNC, MS_SYNC и MS_INVALIDATE, но не MS_ASYNC и MS_SYNC.

MS_ASYNC дает системе задание на запись и немедленно возвращается в вызывающий процесс.



MS_SYNC дает системе задание на запись и ждет его исполнения.

MS_INVALIDATE приказывает системе обновить другие отражения этого файла так, чтобы они содержали изменения, внесенные этим вызовом.



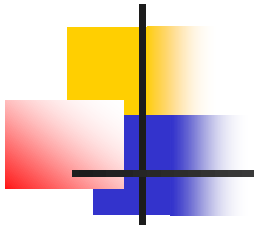
ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

При удачном завершении вызова возвращаемое значение равно нулю. При ошибке оно равно -1

Отображаемая память

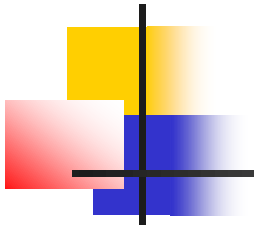
- Например, чтобы сбросить на диск буфер общедоступного файла, отображенного в адресе mem_addr длины mem_length байт:

  msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);



Как и с совместно используемой памятью, пользователи областей отображенной памяти должны следовать протоколу, чтобы избежать условий гонки.

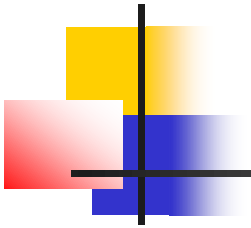
Например, семафор может использоваться, чтобы препятствовать доступу более одного процесса к отображенной памяти.



Отображаемая память

- **Частные отображения**
- Указывая флаг MAP_PRIVATE при вызове mmap создается область копирования при записи. Любая операция записи отражается только в памяти этого процесса; другие процессы, которые отображают тот же самый файл, не будут видеть изменения.

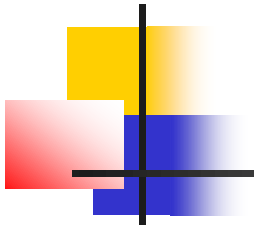
Вместо того, чтобы писать непосредственно странице, разделенной всеми процессами, процесс пишет частной копии этой страницы. Все последующие операции чтения и записи процессом используют эту же страницу.



Отображаемая память

- **Другие применения mmap**
- Вызов mmap может использоваться и в целях не связанных с взаимодействием процессов. Одно из применений - замена операций для чтения и записи. Например, вместо того, чтобы явно читать содержимое файла в память, программа могла бы отобразить файл в память и просмотреть его, используя чтение памяти. Для некоторых программ, это более удобно и может работать быстрее, чем явные операции ввода - вывода.

Пример программы



Первая программа, листинг 5.5, генерирует случайное число и пишет его в файл. Вторая программа, листинг 5.6, читает число, печатает его, и удваивает его в файле. Обе программы считывают имена отображаемых файлов из командной строки.

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
/* Возвратить случайное число из диапазона [low, high]. */
int random_range (unsigned const low, unsigned const high)
{
        unsigned const range = high - low + 1;
        return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}
int main (int argc, char* const argv[])
{
        int fd;
        void* file_memory;
        /* Инициализируем генератор случайных чисел. */
        srand (time (NULL));

        /* Открываем(создаем) файл, достаточно  большой, чтобы хранить целое число без знака. */
        fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
        lseek (fd, FILE_LENGTH+1, SEEK_SET);
        write (fd, "", 1);
        lseek (fd, 0, SEEK_SET);

        /* Создаем отображение в памяти. */
        file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
        close (fd);

        /* Пишем случайное целое число в отображенную память. */
        sprintf((char*) file_memory, "%d\n", random_range (-100, 100));

        /* Освобождаем память. */
        munmap (file_memory, FILE_LENGTH);

        return 0;
}
```

Listing 5.6 (mmap-read.c) Читает число из файла отображенного в памяти и удваивает число.

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
int main (int argc, char* const argv[])
{
        int fd;
        void* file_memory;
        int integer;
        /* Открыть файл. */
        fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
        /* Отобразить файл в память.  */
        file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);
        close (fd);
        /* Чтение целого  числа, распечатка и умножение на 2.  */
        scanf (file_memory, "%d", &nteger);
        printf ("значение: %d\n", integer);
        sprintf ((char*) file_memory, "%d\n", 2 * integer);
        /* Освобождение памяти. */
        munmap (file_memory, FILE_LENGTH);
        return 0;
}
```

Общая (совместно используемая) память

Один из самых простых методов межпроцессного взаимодействия - использовать общую память. Общая память позволяет двум или более процессам обращаться к одной и той же области памяти, как будто они все вызывали malloc и им были возвращены указатели на одну и ту же физическую память. Когда один процесс изменяет память, все другие процессы "видят" модификацию.

Общая (совместно используемая) память

Общая память - самая быстрая форма межпроцессного взаимодействия, потому что все процессы совместно используют одну и ту же часть памяти. Доступ к этой общей памяти осуществляется с той же скоростью, что и при обращении к несовместно используемой памяти, и это не требует системного вызова или входа в ядро. Это также не требует излишнего копирования данных.

Общая (совместно используемая) память

Поскольку ядро не синхронизирует доступы к совместно используемой памяти, вы должны сами обеспечить синхронизацию.

Например, процесс не должен читать из памяти, пока данные не записаны туда, и два процесса не должны написать по одному и тому же адресу памяти в одно и то же время. Общая стратегия избегания условий гонки состоит в том, чтобы использовать семафоры.

Общая (совместно используемая) память

- **Модель памяти**
- Чтобы использовать сегмент общей памяти, один процесс должен выделить сегмент. Тогда каждый процесс, желающий обращаться к сегменту должен подключить сегмент. После окончания его использования сегмента, каждый процесс отключает сегмент. В некоторый момент, один процесс должен освободить сегмент.

Общая (совместно используемая) память

Выделение нового сегмента общей памяти приводит к созданию страницы виртуальной памяти.

Поскольку все процессы желают обратиться к одному и тому же общему сегменту, то только один процесс должен выделить новый общий сегмент.

Выделение существующего сегмента не создает новых страниц, а возвращает идентификатор для существующих.

Чтобы разрешить процессу использовать сегмент общей памяти, процесс подключает сегмент, который добавляет отображение его виртуальной памяти на общедоступные страницы сегмента.

Общая (совместно используемая) память

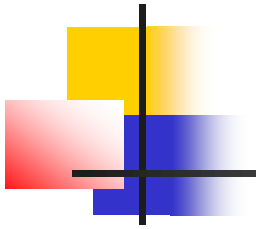
Когда работа с сегментом завершена, эти отображения удаляются. Когда ни один из процессов не хочет обращаться к сегментам общей памяти, какой-то один процесс должен освободить страницы виртуальной памяти. Все сегменты общей памяти выделяются постранично и округляются до размера страницы системы, который является числом байтов в странице памяти. На системах Linux, размер страницы равен 4 КБ, но вы должны получить это значение, вызывая функцию getpagesize.

Общая (совместно используемая) память

Выделение

- Процесс выделяет сегмент общей памяти, используя shmget ("SHared Memory GET"). Его первый параметр - целочисленный ключ, который определяет, какой сегмент создать. Несвязанные процессы могут обращаться к одному и тому же сегменту, используя одно и то же ключевое значение. К сожалению, другие процессы, возможно, также выбрали тот же самый ключ, что может привести к конфликту. Используя специальную константу IPC_PRIVATE как ключевое значение, гарантируется, что создастся совершенно новый сегмент памяти.

Общая (совместно используемая) память



- Его второй параметр определяет число байтов в сегменте. Поскольку сегменты выделяются постранично, число фактически выделенных байт округляется до размера страницы.
- Третий параметр - поразрядное двоичное значение флажка, которые определяют опции к shmget.

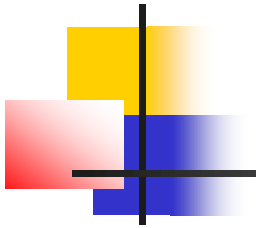
Общая (совместно используемая) память

- **Подключение и отключение**
- Чтобы сделать сегмент общей памяти доступным, процесс должен использовать shmat, "SHared Memory ATtach" Передайте ему идентификатор сегмента общей памяти SHMID, возвращенный shmget. Второй параметр - указатель, который определяет, где в адресном пространстве вашего процесса вы хотите отобразить общую память; если вы передадите NULL, то Linux выберет любой доступный адрес.

Общая (совместно используемая) память

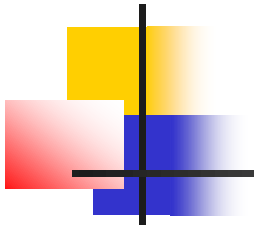
- Если вызов успешен, он вернет адрес подключенного общего сегмента. Потомки, созданные вызовами fork, наследуют подключенные общие сегменты; они могут отключить сегменты общей памяти, если захотят.
- Когда вы закончили работу с сегментом общей памяти, сегмент должен быть отключен, используя shmdt ("SHared Memory DeTach"). Передайте ему адрес, возвращенный shmat. Если сегмент был освобожден, и больше не осталось процессов, использующих его, он будет удален. Вызовы exit и exec автоматически отключают сегменты.

Общая (совместно используемая) память



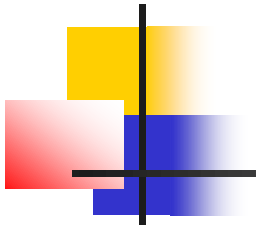
Shmctl ("SHared Memory ConTrol") вызов возвращает информацию об сегменте общей памяти и может изменить его. Первый параметр - идентификатор сегмента общей памяти.

Общая (совместно используемая) память



Каждый сегмент общей памяти должен быть явно освобожден, используя shmctl, когда Вы закончили работу с ним, чтобы избежать нарушение системного предела размера количества сегментов общей памяти. Вызовы exit и exec отключат сегменты памяти, но не освобождят их.

Общая (совместно используемая) память



Команда ipcs предоставляет информацию относительно средств взаимодействия процессов, включая общие сегменты памяти. Используйте флаг -m, чтобы получить информацию об общей памяти.