Очереди сообщений Posix IPC

- Структура очереди сообщений Posix IPC
- Функции mq_open, mq_close, mq_unlink
- Функции mq_getattr и mq_setattr
- Функции mqsend и mqreceive
- Ограничения очередей сообщений
- Функция mq_notify

Структура очереди

- Очередь сообщений можно рассматривать как связный список. Процессы с соответствующими разрешениями могут помещать сообщения в очередь, а процессы с другими соответствующими разрешениями могут извлекать их оттуда. Каждое сообщение представляет собой запись, и каждому сообщению его отправителем присваивается приоритет. Для записи сообщения в очередь не требуется наличия ожидающего его процесса. Это отличает очереди сообщений от программных каналов и FIFO, в которые нельзя произвести запись, пока не появится считывающий данные процесс.

Структура очереди

- Процесс может записать в очередь какие-то сообщения, после чего они могут быть получены другим процессом в любое время, даже если первый завершит свою работу. Говорят, что очереди сообщений обладают живучестью ядра. Это также отличает их от программных каналов и FIFO.
- Главные отличия очереди сообщений стандарта Posix от стандарта System V заключаются в следующем:

Структура очереди

- операция считывания из очереди сообщений Posix всегда возвращает самое старое сообщение с наивысшим приоритетом, тогда как из очереди System V можно считать сообщение с произвольно указанным типом;
- очереди сообщений Posix позволяют отправить сигнал или запустить программный процесс при помещении сообщения в пустую очередь, тогда как для очередей System V ничего подобного не предусматривается.

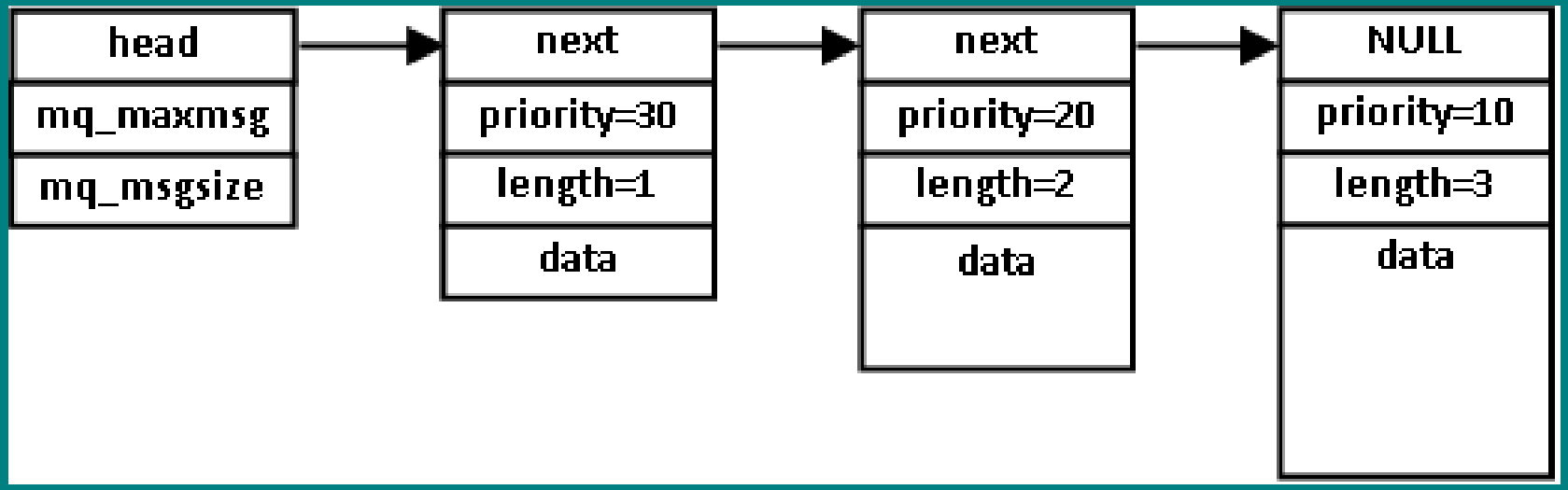
Структура очереди

- Каждое сообщение в очереди состоит из следующих частей:
- приоритет (беззнаковое целое, Posix) либо тип сообщения (длинное целое, System V);
- длина полезной части сообщения, которая может быть нулевой;
- собственно данные (если длина сообщения отлична от 0).
- Этим очереди сообщений отличаются от программных каналов и FIFO.

Структура очереди

- Последние две части сообщения представляют собой байтовые потоки, в которых отсутствуют границы между сообщениями и никак не указывается их тип. На рис. ниже показан возможный вид очереди сообщений.
- Реализация очередей сообщений Posix через связный список, предполагает, что его заголовок содержит два атрибута очереди: максимально допустимое количество сообщений в ней и максимальный размер сообщения.

Структура очереди



Структура очереди

- Поскольку все объекты IPC обладают живучестью ядра, можно написать программу, создающую очередь сообщений Posix, а потом написать другую программу, которая помещает сообщение в такую очередь, а потом еще одну, которая будет считывать сообщения из очереди. Помещая в очередь сообщения с различным приоритетом, можно увидеть, в каком порядке они будут возвращаться функцией **mq_receive**. Но вначале рассмотрим функции открытия, закрытия и удаления очередей сообщений Posix: **mq_open, mq_close, mq_unlink.**

Функции mq_open, mq_close, mq_unlink

- Функция **mq_open** создает новую очередь сообщений либо открывает существующую:
- **#include <mqueue.h>**
- **mqd_t mq_open(char *name, int oflag, /* mode_t mode, struct mq_attr *attr */);**
- Функция возвращает дескриптор очереди в случае успешного завершения или –1 в противном случае.
- Требования к аргументу **name** описаны ранее.
- Аргумент **oflag** может принимать одно из следующих значений: **O_RDONLY**, **O_WRONLY**, **O_RDWR** в сочетании с **O_CREAT**, **O_EXCL**, **O_NONBLOCK**.

Функции mq_open, mq_close, mq_unlink

- При создании новой очереди (указан флаг **O_CREAT** и очередь сообщений еще не существует) требуется указание аргументов **mode** и **attr**. Возможные значения аргумента **mode** приведены в предыдущей презентации. Аргумент **attr** позволяет задать некоторые атрибуты очереди. Если в качестве этого аргумента задать нулевой указатель, очередь будет создана с атрибутами по умолчанию. Сами атрибуты описаны ниже.

Функции mq_open, mq_close, mq_unlink

- Возвращаемое функцией **mq_open** значение называется дескриптором очереди сообщений, но оно не обязательно должно быть (и, скорее всего, не является) небольшим целым числом, как дескриптор файла или программного сокета. В большинстве реализаций дескрипторы трактуются как указатели на структуру. Название «дескриптор» было дано им по ошибке. Это значение используется в качестве первого аргумента оставшихся семи функций для работы с очередями сообщений.

Функции mq_open, mq_close, mq_unlink

- Открытая очередь сообщений закрывается функцией **mq_close**:
- **#include <mqueue.h>**
- **int mq_close(mqd_t mqdes);**
- Функция возвращает 0 в случае успешного завершения или –1 в случае ошибки.
- По действию эта функция аналогична **close** для открытого файла: вызвавший функцию процесс больше не может использовать дескриптор, но очередь сообщений из системы не удаляется.

Функции mq_open, mq_close, mq_unlink

- При завершении процесса все открытые очереди сообщений закрываются, как если бы для каждой был сделан вызов **mq_close**.
- Для удаления из системы имени (**name**), которое использовалось в качестве аргумента при вызове **mq_open**, нужно использовать функцию **mq_unlink**:
- **#include <mqueue.h>**
- **int mq_unlink(char *name);**
- Функция возвращает 0 в случае успешного завершения и –1 в случае ошибки.

Функции mq_open, mq_close, mq_unlink

- Для очереди сообщений (как и для файла) ведется подсчет числа процессов, в которых она открыта в данный момент, и по действию эта функция аналогична **unlink** для файла: имя (**name**) может быть удалено из системы, даже пока число подключений к очереди отлично от нуля, но удаление очереди (в отличие от удаления имени из системы) не будет осуществлено до того, как очередь будет закрыта последним использовавшим ее процессом. Очереди сообщений Posix продолжают существовать, храня все имеющиеся в них сообщения, даже если нет процессов, в которых они были бы открыты.

Функции mq_open, mq_close, mq_unlink

- Очередь существует, пока она не будет удалена явно с помощью **mq_unlink**.
- Если очередь сообщений реализована через отображаемые в память файлы (как будет показано далее), она может обладать живучестью файловой системы, но это не является обязательным и рассчитывать на это нельзя.
- Пример создания очереди сообщений Posix приведен в программе mqcreate1.c, доступной в разделе «PosixMsg» сайта [6].

Функции mq_open, mq_close, mq_unlink

- Сборка программы потребует ключа **gcc –lrt**, поскольку библиотека librt.so содержит функции, обеспечивающие большинство интерфейсов, описанных в Posix.1b Realtime Extension:
- **[gun@CentOS]$ gcc -Wall -o mqcreate1 mqcreate1.c -lrt**
- В командной строке при запуске этой программы можно указать параметр **–e**, требующий создания эксклюзивной очереди. Программа вызывает функцию **mq_open**, указывая ей в качестве имени IPC полученный из командной строки параметр. Рассмотрим результат работы программы в CentOS 6.9.

Функции mq_open, mq_close, mq_unlink

- *очередь успешно создается:*
- **[gun@CentOS]$ ./mqcreate1 /tmp.123**
- **Result:Success**
- **[gun@localhost PosixMsg]$ ls -l /dev/mqueue**
- **-rw-------. 1 gun gun   80 Янв 30 14:10 tmp.123**
- *очередь уже существует:*
- **[gun@localhost PosixMsg]$ ./mqcreate1 -e /tmp.123**
- **Result:File exists**
- *свойства очереди:*
- **[gun@CentOS]$ cat /dev/mqueue/tmp.123**
- **QSIZE:0 NOTIFY:0 SIGNO:0 NOTIFY_PID:0**

Функции mq_open, mq_close, mq_unlink

- Свойства очереди:
- **QSIZE** – число байт данных во всех сообщениях очереди;
- **NOTIFY** – метод уведомления (0 - **SIGEV_SIGNAL;** 1 - **SIGEV_NONE;** 2 - **SIGEV_THREAD);**
- **SIGNO** – номер сигнала, используемого для **SIGEV_SIGNAL;**
- **NOTIFY_PID**– если не 0, то это идентификатор (PID) процесса, использовавшего функцию **mq_notify**, чтобы зарегистрироваться на уведомление о появлении сообщения; предыдущие поля описывают настройки уведомления.
- Эта версия программы названа mqcreate1, поскольку

Функции mq_open, mq_close, mq_unlink

- Свойства очереди:
- **QSIZE** – число байт данных во всех сообщениях очереди;
- **NOTIFY** – метод уведомления (0 - **SIGEV_SIGNAL;** 1 - **SIGEV_NONE;** 2 - **SIGEV_THREAD);**
- **SIGNO** – номер сигнала, используемого для **SIGEV_SIGNAL;**
- **NOTIFY_PID**– если не 0, то это идентификатор (PID) процесса, использовавшего функцию **mq_notify**, чтобы зарегистрироваться на уведомление о появлении сообщения; предыдущие 2 поля описывают настройки уведомления.

Функции mq_open, mq_close, mq_unlink

- В той же папке сайта имеется программа mqunlink.c, удаляющая из системы очередь сообщений. С ее помощью можно удалить очередь сообщений, созданную программой mqcreate1:
- **[gun@CentOS]$ ./mqunlink /tmp.123**
- **Result=Success**
- **[gun@CentOS]$ ls -l /dev/mqueue**
- **итого 0**
- При этом будут удален файл из каталога **/dev/mqueue**, который относится к этой очереди.

Функции mq_getattr и mq_setattr

- У каждой очереди сообщений имеются четыре атрибута, которые могут быть получены функцией **mq_getattr** и установлены (по отдельности) функцией **mq_setattr**:
- **#include <mqueue.h>**
- **int mq_getattr(mqd_t mqdes , struct mq_attr *attr);**
- **int mq_setattr(mqd_t mqdes , const struct mq_attr *attr , struct mq_attr *oattr);**
- Обе функции возвращают 0 в случае успешного завершения и –1 в случае возникновения ошибок.

Функции mq_getattr и mq_setattr

- mq_attr {
- long mq_flags; /* флаг очереди: 0, O_NONBLOCK */
- long mq_maxmsg;/ *максимальное количество сообщений в очереди */
- long mq_msgsize; /* максимальный размер сообщения (в байтах) */
- long mq_curmsgs; // текущее количество сообщений в очереди
- }

Функции mq_getattr и mq_setattr

- Указатель на такую структуру может быть передан в качестве четвертого аргумента **mq_open**, что дает возможность установить параметры **mq_maxmsg** и **mq_msgsize** в момент создания очереди. Другие два поля структуры функцией **mq_open** игнорируются.
- Функция **mq_getattr** присваивает полям структуры, на которую указывает **attr**, текущие значения атрибутов очереди.
- Функция **mq_setattr** устанавливает атрибуты очереди, но фактически используется только поле **mqflags** той структуры, на которую указывает **attr.**

Функции mq_getattr и mq_setattr

- Это дает возможность сбрасывать или устанавливать флаг запрета блокировки. Другие три поля структуры игнорируются: максимальное количество сообщений в очереди и максимальный размер сообщения могут быть установлены только в момент создания очереди, а количество сообщений в очереди можно только считать, но не изменить.
- Кроме того, если указатель **oattr** ненулевой, возвращаются предыдущие значения атрибутов очереди (**mq_flags**, **mq_maxmsg**, **mq_msgsize**) и текущий статус очереди (**mq_curmsgs**).

Функции mq_getattr и mq_setattr

- Программа mqgetattr.c из указанного ранее раздела сайта открывает указанную очередь сообщений и выводит значения ее атрибутов.
- Мы можем создать очередь сообщений и вывести значения ее атрибутов, устанавливаемые по умолчанию (программой mqcreate1.c):
- **[gun@CentOS]$ ./mqgetattr /tmp.123**
- **Result:Success**
- **max #msgs = 10, max #bytes/msg = 8192, #currently on queue = 0**

Функции mq_getattr и mq_setattr

- Изменим программу mqcreate1.c таким образом (см. mqcreate2.c), чтобы при создании очереди иметь возможность указывать максимальное количество сообщений и максимальный размер сообщения. Нужно обязательно задать оба параметра. Причем параметр командной строки, требующий аргумента, указывается с помощью двоеточия (после параметров m и z) в вызове **getopt**. Если не указан ни один из двух новых параметров, то необходимо передать функции **mq_open** пустой указатель в качестве последнего аргумента. В противном случае передается указатель на структуру **attr**.

Функции mq_getattr и mq_setattr

- Запустим теперь новую версию программы в системе CentOS 6.9, указав параметры, превышающие значения по умолчанию: максимальное количество сообщений 20 и максимальный размер сообщения 16384 байт.
- **[gun@CentOS]$ ./mqcreate2 -m 20 -z 16384 /tmp.123**
- **Result:Invalid argument**
- Видим, что в данной реализации по умолчанию устанавливаются максимально возможные параметры очереди сообщений. Действительно, указав значения меньше предельных, получим:

Функции mq_getattr и mq_setattr

- [gun@CentOS]$ ./mqcreate2 -m 5 -z 4096 /tmp.123
- Result:Success
- [gun@CentOS]$ ./mqgetattr /tmp.123
- max #msgs = 5, max #bytes/msg = 4096, #currently on queue = 0

Функции mqsend и mqreceive

- Эти две функции предназначены для помещения сообщений в очередь и получения их оттуда. Каждое сообщение имеет свой приоритет, который представляет собой беззнаковое целое, не превышающее **MQ_PRIO_MAX**. Стандарт Posix требует, чтобы эта константа была не меньше 32.
- Функция **mq_receive** всегда возвращает старейшее в указанной очереди сообщение с максимальным приоритетом, и приоритет может быть получен вместе с содержимым сообщения и его длиной. Действие **mq_receive** отличается от действия **msgrcv** в System V (параграф 6.3.2).

Функции mqsend и mqreceive

- Сообщения System V имеют поле **type**, аналогичное по смыслу приоритету, но для функции **msgrcv** можно указать три различных алгоритма возвращения сообщений: старейшее сообщение в очереди, старейшее сообщение с указанным типом или старейшее сообщение с типом, не превышающим указанного значения.
- **#include <mqueue.h>**
- **int mq_send(mqd_t mqdes , const char *ptr , size_t len , unsigned int prio);**
- Функция возвращает 0 в случае успешного завершения или –1 в случае возникновения ошибок.

Функции mqsend и mqreceive

- **ssize_t mq_receive(mqd_t mqdes , char *ptr , size_t len , unsigned int *priop);**
- Функция возвращает количество байтов в сообщении в случае успешного завершения или –1 в случае ошибки.
- Первые три аргумента обеих функций аналогичны первым трем аргументам функций **write** и **read** соответственно. Значение аргумента **len** функции **mq_receive** должно быть по крайней мере не меньше максимального размера сообщения, которое может быть помещено в очередь, то есть значения поля **mq_msgsize** структуры **mq_attr** для этой очереди.

Функции mqsend и mqreceive

- Если **len** оказывается меньше этой величины, немедленно возвращается ошибка **EMSGSIZE**. Это означает, что большинству приложений, использующих очереди сообщений Posix, придется вызывать **mq_getattr** после открытия очереди для определения максимального размера сообщения, а затем выделять память под один или несколько буферов чтения этого размера. Требование, чтобы буфер был больше по размеру, чем максимально возможное сообщение, позволяет функции **mq_receive** не возвращать уведомление о том, что размер письма превышает объем буфера.

Функции mqsend и mqreceive

- Сравните это, например, с флагом **MSG_NOERROR** и ошибкой **E2BIG** для очередей сообщений System V (параграф 6.3.2) и флагом **MSG_TRUNC** для функции **recvmsg**, используемой с дейтаграммами UDP (описан в главе 8 учебного пособия).
- Аргумент **prio** устанавливает приоритет сообщения для **mq_send**, его значение должно быть меньше **MQ_PRIO_MAX**. Если при вызове **mq_receive** аргумент **priop** является ненулевым указателем, в нем сохраняется приоритет возвращаемого сообщения.

Функции mqsend и mqreceive

- Если приложению не требуется использование различных приоритетов сообщений, можно указывать его равным нулю для **mq_send** и передавать **mq_receive** нулевой указатель в качестве последнего аргумента.
- Разрешена передача сообщений нулевой длины. В стандарте (Posix.1) нигде не запрещена передача сообщений нулевой длины. Функция **mq_receive** возвращает количество байтов в сообщении (в случае успешного завершения работы) или –1 в случае возникновения ошибок, так что 0 обозначает сообщение нулевой длины.

Функции mqsend и mqreceive

- Файл mqsend.c, доступный на сайте в разделе «PosixMsg», содержит текст программы, помещающей сообщение в очередь.
- И размер сообщения, и его приоритет являются обязательными аргументами командной строки. Буфер под сообщение выделяется функцией **calloc**, которая инициализирует его нулем.
- Файл mqreceive.c, доступный на сайте в том же разделе «PosixMsg», содержит текст программы, считывающей сообщение из очереди.

Функции mqsend и mqreceive

- Параметр командной строки **–n** отключает блокировку. При этом программа возвращает сообщение об ошибке, если в очереди нет сообщений. Открыв очередь, необходимо получить ее атрибуты, вызвав **mq_getattr**. Обязательно нужно определить максимальный размер сообщения, потому что необходимо выделить буфер подходящего размера перед вызовом **mq_receive**. Программа выводит размер считываемого сообщения и его приоритет.

Функции mqsend и mqreceive

- Воспользуемся этими и ранее упомянутыми программами, чтобы проиллюстрировать использование поля приоритета.
- *1. создаем очередь*
- **[gun@CentOS]$ ./mqcreate2 /test1**
- **Result:Success**
- *2. смотрим на ее атрибуты*
- **[gun@CentOS]$ ./mqgetattr /test1**
- **max #msgs = 10, max #bytes/msg = 8192, #currently on queue = 0**

Функции mqsend и mqreceive

- *3. отправка 100 байт с некорректным значением приоритета*
- **[gun@CentOS]$ ./mqsend /test1 100 99999**
- **Sent:-1 bytes**
- *4. отправка 100 байт, приоритет 6, возврат нуля – это успех*
- **[gun@CentOS]$ ./mqsend /test1 100 6**
- **Sent:0 bytes**
- *5. отправка 50 байт, приоритет 18*
- **[gun@CentOS]$./mqsend /test1 50 18**
- **Sent:0 bytes**

Функции mqsend и mqreceive

- *6. отправка 33 байт, приоритет 18*
- **[gun@CentOS]$./mqsend /test1 33 18**
- **Sent:0 bytes**
- *7. проверка содержимого очереди*
- **[gun@CentOS]$./mqgetattr /test1**
- **max #msgs = 10, max #bytes/msg = 8192, #currently on queue = 3**
- *8. возвращается старейшее сообщение с наивысшим приоритетом*
- **[gun@CentOS]$./mqreceive /test1**
- **read 50 bytes, priority = 18**

Функции mqsend и mqreceive

- **[gun@CentOS]$./mqreceive /test1**
- **read 33 bytes, priority = 18**
- **[gun@CentOS]$./mqreceive /test1**
- **read 100 bytes, priority = 6**
- *9. отключаем блокировку, убеждаемся, что сообщений больше нет*
- **[gun@CentOS]$./mqreceive –n /test1**
- **read -1 bytes, priority = 4029664**
- *10. удаляем очередь*
- **[gun@CentOS]$./mqunlink /test1**
- **Result=Success**

Ограничения очередей сообщений

- Ранее были описаны два ограничения, устанавливаемые для любой очереди в момент ее создания:
- **mq_maxmsg** — максимальное количество сообщений в очереди;
- **mq_msgsize** — максимальный размер сообщения.
- Не существует каких-либо ограничений на эти значения, хотя в некоторых реализациях необходимо наличие в файловой системе места для файла требуемого размера. Кроме того, ограничения на эти величины могут накладываться реализацией виртуальной памяти.

Ограничения очередей сообщений

- Другие два ограничения определяются реализацией:
- **MQ_OPEN_MAX** — максимальное количество очередей сообщений, которые могут быть одновременно открыты каким-либо процессом (Posix требует, чтобы эта величина была не меньше 8);
- **MQ_PRIO_MAX** — максимальное значение приоритета плюс один (Posix требует, чтобы эта величина была не меньше 32).
- Эти две константы определяются в заголовочном файле <unistd.h> и могут быть получены во время выполнения программы вызовом функции sysconf, как показано в тексте программы mqsysconf.c, доступной на сайте.

Ограничения очередей сообщений

- Запустив эту программу, получим:
- **[gun@CentOS]$ ./mqsysconf**
- **MQ_OPEN_MAX = 256, MQ_PRIO_MAX = 32768**
- Можно также использовать утилиту **sysctl**:
- **[gun@CentOS]# sysctl -A | grep fs.mqueue**
- **fs.mqueue.queues_max = 256**
- **fs.mqueue.msg_max = 10**
- **fs.mqueue.msgsize_max = 8192**
- **fs.mqueue.msg_default = 10**
- **fs.mqueue.msgsize_default = 8192**

Функция mq_notify

- Один из недостатков очередей сообщений System V, заключается в невозможности уведомить процесс о том, что в очередь было помещено сообщение. Можно заблокировать процесс при вызове **msgrcv**, но тогда невозможно выполнять другие действия во время ожидания сообщения. Если указать флаг отключения блокировки при вызове **msgrcv** (**IPC_NOWAIT**), процесс не будет заблокирован, но придется регулярно вызывать эту функцию, чтобы получить сообщение, когда оно будет отправлено. Такая процедура называется опросом и на нее тратится лишнее время.

Функция mq_notify

- Очереди сообщений Posix допускают асинхронное уведомление о событии, когда сообщение помещается в очередь. Это уведомление может быть реализовано либо отправкой сигнала, либо созданием программного потока (см. главу 7) для выполнения указанной функции.
- Режим уведомления включается с помощью функции **mq_notify**:
- **#include <mqueue.h>**
- **int mq_notify(mqd_t mqdes, const struct sigevent *notification);**

Функция mq_notify

- Функция возвращает 0 в случае успешного выполнения или –1 в случае ошибки. Она включает и выключает асинхронное уведомление о событии для указанной очереди. Структура **sigevent** впервые появилась в стандарте Posix.1. Эта структура и все новые константы, относящиеся к сигналам, определены в заголовочном файле <signal.h>:
- **union sigval {**
- **int sival_int; /* целое значение */**
- **void *sival_ptr; /* указатель */**
- **};**

Функция mq_notify

- struct sigevent {
- int sigev_notify;/*SIGEV_{NONE,SIGNAL,THREAD} */
- int sigev_signo; /* номер сигнала, если SIGEV_SIGNAL */
- union sigval sigev_value; /* передается обработчику сигнала или потоку */
- /* Следующие два поля определены для SIGEV_THREAD */
- void (*sigev_notify_function) (union sigval);
- pthread_attr_t *sigev_notify_attributes;
- }

Функция mq_notify

- Есть несколько правил, действующих для этой функции:
- 1. Если аргумент **notification** ненулевой, процесс ожидает уведомления при поступлении нового сообщения в указанную очередь, пустую на момент его поступления. Мы говорим, что процесс регистрируется на уведомление для данной очереди.
- 2. Если аргумент **notification** представляет собой нулевой указатель и процесс уже зарегистрирован на уведомление для данной очереди, то уведомление для него отключается.

Функция mq_notify

- 3. Только один процесс может быть зарегистрирован на уведомление для любой данной очереди в любой момент.
- 4. При помещении сообщения в пустую очередь, для которой имеется зарегистрированный на уведомление процесс, оно будет отправлено только в том случае, если нет заблокированных в вызове **mq_receive** для этой очереди процессов. Таким образом, блокировка в вызове **mq_receive** имеет приоритет перед любой регистрацией на уведомление.

Функция mq_notify

- 5. При отправке уведомления зарегистрированному процессу регистрация снимается. Процесс должен зарегистрироваться снова (если в этом есть необходимость), вызвав **mq_notify** еще раз.
- Пример программы, включающую отправку сигнала **SIGUSR1** при помещении сообщения в пустую очередь, представлен в файле mqnotify.c на сайте. В ней объявлено несколько глобальных переменных, используемых совместно функцией **main** и обработчиком сигнала (**sig_usr1**). Программа открывает очередь сообщений, получает ее атрибуты и выделяет буфер считывания соответствующего размера.

Функция mq_notify

- Затем устанавливается обработчик для сигнала **SIGUSR1**. Для этого полю **sigev_notify** структуры sigevent присваивается значение **SIGEV_SIGNAL**, что говорит системе о необходимости отправки сигнала, когда очередь из пустой становится непустой. Полю **sigev_signo** присваивается значение, соответствующее тому сигналу, который необходимо получить. Затем вызывается функция **mq_notify**.
- Функция **main** после этого зацикливается, и процесс приостанавливается при вызове функции **pause**, возвращающей –1 при получении сигнала.

Функция mq_notify

- Обработчик сигнала вызывает **mq_notify** для перерегистрации, считывает сообщение и выводит его длину. В этой программе игнорируется приоритет полученного сообщения.
- Запустив эту программу в одном из окон:
- **[gun@CentOS]$ ./mqcreate /test1**
- **[gun@CentOS]$ ./mqnotifysig /test1**
- выполним затем следующую команду в другом окне:
- **[gun@CentOS]$ ./mqsend /test1 50 16**

Функция mq_notify

- Как и ожидалось, программа mqnotifysig.c выведет сообщение:
- **SIGUSR1 received**, **read 50 bytes**.
- Заметим, что пустые сообщения тоже вызывают отправку сигнала:
- **[gun@CentOS]$ ./mqnotifysig /test1**
- **[gun@CentOS]$ ./mqsend /test1 0 16**
- **SIGUSR1 received, read 0 bytes**
- Можно проверить, что только один процесс может быть зарегистрирован на получение уведомления, запустив копию программы в другом окне:

Функция mq_notify

- **[gun@CentOS]$ ./mqnotifysig /test1**
- **Result:Device or resource busy**
- Это сообщение соответствует коду ошибки **EBUSY**.
- Недостаток программы mqnotifysig.c в том, что она вызывает **mq_notify**, **mq_receive** и **printf** из обработчика сигнала. Ни одну из этих функций вызывать оттуда не следует. Функции, которые могут быть вызваны из обработчика сигнала, относятся к группе, называемой, согласно Posix, async-signal-safe functions (функции, обеспечивающие безопасную обработку асинхронных сигналов).

Функция mq_notify

- Функции, которых нет в этом списке, не должны вызываться из обработчика сигнала. В списке отсутствуют стандартные функции библиотеки ввода-вывода и функции **pthread_XXX** для работы с потоками (см. главу 7). Из всех функций IPC, рассматриваемых в этом пособии, в список попали только **sem_post**, **read** и **write** (подразумевается, что последние две используются с программными каналами и FIFO).

Функция mq_notify

- Одним из способов исключения вызова каких-либо функций из обработчика сигнала является установка этим обработчиком глобального флага, который проверяется программой для получения информации о приходе сообщения. Проблема тут в том, что уведомление отсылается только в том случае, когда сообщение помещается в пустую очередь. Если в очередь поступают два сообщения, прежде чем первое будет считано, то отсылается только одно уведомление.