

Writing a Resource Manager

**Перевод: Владимир Зайцев,
г. Харьков**

**Редактор перевода: Сергей Малышев,
г. Озерск, Челябинская обл.**

Написание менеджера ресурсов

Эта глава содержит следующие разделы:

- Что такое менеджер ресурсов?
- Компоненты менеджера ресурсов.
- Пример простого менеджера ресурсов устройства.
- Структуры, содержащие данные.
- Обработка сообщения IO_READ.
- Обработка сообщения IO_WRITE.
- Методы возврата и ответа.
- Тонкости обработки ввода/вывода.
- Обработка атрибутов.
- Комбинированные сообщения.
- Расширенные структуры управления данными (DCS).
- Обработка сообщений devctl().
- Обработка ionotify() и select().
- Обработка приватных сообщений и импульсов.
- Обработка сообщений open(), dup() и close().
- Обработка разблокирования клиента по сигналу или тайм-ауту.
- Обработка прерываний.
- Многопоточный менеджер ресурсов.
- Менеджер ресурсов файловой системы.
- Типы сообщений.
- Структуры данных менеджера ресурсов.

Что такое менеджер ресурсов?

Эта глава предполагает, что Вы уже знакомы с механизмом передачи сообщений. Если нет - читайте главу "Микроядро Neutrino" в книге "Архитектура системы" и описание семейства

функций `MsgSend()/MsgReceive()/MsgReply()` в книге "Справочник по библиотечным функциям".


Этот раздел содержит следующие подразделы:

- Зачем писать менеджер ресурсов?
- Что творится внутри?
- Типы менеджеров ресурсов.

Менеджер ресурсов – это программа-сервер пользовательского уровня, принимающая сообщения от других программ и, возможно, взаимодействующая с аппаратным обеспечением. Это процесс, который регистрирует префикс имени пути в пространстве имён (например, `/dev/ser1`). После регистрации другие процессы могут открывать это имя как файл, используя стандартную функцию `open()`, и затем, получив дескриптор файла, вызывать функции `read()` и `write()`. В результате менеджер ресурсов получает запрос на открытие, и следом запросы на чтение и запись. Менеджер ресурсов не ограничивается обработкой только таких вызовов, как `open()`, `read()` и `write()` – он может поддерживать любые функции, основанные на файловых дескрипторах, а равно и другие формы сообщений между процессами.

В QNX6 менеджеры ресурсов отвечают за предоставление интерфейса к различным типам устройств. В других операционных системах управление реальными аппаратными устройствами (например, последовательными портами, параллельными портами, сетевыми картами и приводами дисков) или виртуальными устройствами (например, `/dev/null`, сетевой файловой системой и псевдо-tty) связано с драйверами устройств. Но в отличие от драйверов устройств, менеджеры ресурсов QNX6 исполняются как процессы, отдельно от ядра. Фактически, менеджер ресурса выглядит просто как другая программа пользовательского уровня. Весьма ценное свойство! Добавление менеджеров ресурсов в QNX6 не будет влиять на другие части операционной системы – драйверы разрабатываются и отлаживаются подобно любым другим приложениям. И, поскольку менеджеры ресурсов находятся в собственном защищённом адресном пространстве, ошибка в драйвере устройства не будет вызывать крах всей операционной системы.

Программисты, пишущие драйверы устройств для разных UNIX-систем, ограничены в том, что они могут сделать внутри драйвера устройства. Но, поскольку драйвер устройства в QNX6 является обычным процессом, программист не испытывает никаких ограничений в своих действиях (за исключением того, что существуют некоторые ограничения внутри ISR – стандартных программ обслуживания прерываний).

 Для присоединения к системе менеджер ресурсов должен быть запущен от имени `root`.

Несколько примеров

Последовательный порт может управляться менеджером ресурсов `devc-ser8250`, реальный ресурс в пространстве имён может называться `/dev/ser1`. Когда некий процесс запрашивает услуги последовательного порта, он делает это, открывая последовательный порт (в данном случае `/dev/ser1`).

```
fd = open("/dev/ser1", O_RDWR);
for (packet = 0; packet < npackets; packet++) write(fd, packets[packet], PACKET_SIZE);
close(fd);
```

Поскольку менеджеры ресурсов выполняются как процессы, их использование не ограничивается драйверами устройств – любой сервер может быть написан как менеджер

ресурса. Сервер, работающий с DVD-файлами и передающий данные графическому интерфейсу, не может быть классифицирован как драйвер. Тем не менее, он может быть написан как менеджер ресурса. Он может зарегистрировать имя `/dev/dvd` и в результате клиент может выполнить следующее:

```
fd = open("/dev/dvd", O_RDONLY);
while (data = get_dvd_data(handle, &nbytes)) write(fd, data, nbytes);
close(fd);
```

Зачем писать менеджер ресурсов?

Вот несколько причин, по которым Вы захотите написать менеджер ресурса:

1. API (программный интерфейс приложения) является POSIX-совместимым. API взаимодействия (интерфейс) с менеджером ресурса является, в основном, POSIX-совместимым. Все программисты, пишущие на языке C знакомы с функциями `open()`, `read()` и `write()`. Поэтому затраты на обучение минимальны, также как и необходимость документирования интерфейса с Вашим сервером.
2. Вы можете унифицировать интерфейсы. Если Вы пишете много процессов-серверов, то написание каждого сервера как менеджера ресурсов сводит к минимуму количество различных интерфейсов, которые должны использовать клиенты. Если серверы написаны как менеджеры ресурсов, то интерфейсом всех этих серверов будут функции POSIX: `open()`, `read()`, `write()` и любые другие, имеющие смысл. Для сообщений управляющего типа, не вписывающихся в модель `read/write`, существует функция `devctl()` (хотя `devctl()` и не является функцией POSIX).
3. Утилиты командной строки так же могут взаимодействовать с менеджерами ресурсов. Поскольку API для связи с менеджером ресурсов является набором POSIX-функций, и стандартные POSIX-утилиты используют этот же API, для связи с менеджерами ресурсов можно использовать утилиты.

Например, урезанный модуль протоколов TCP/IP (`ttcpip`) содержит код менеджера ресурсов, регистрирующий имя `/proc/ipstats`. Если Вы откроете это имя, и будете читать из него, код менеджера ресурсов в ответ предоставит блок текста, описывающего статистику для IP.

Утилита `cat` использует имя файла, открывает его, читает его содержимое и отображает всё, что прочитала, в стандартный вывод (обычно на экран). В результате, Вы можете набрать команду:

```
cat /proc/ipstats
```

Код менеджера ресурсов в модуле протоколов TCP/IP выдаст ответное сообщение с таким текстом:

```
Ttcpip Sep  5 2000 08:56:16

verbosity level 0
ip checksum errors: 0
udp checksum errors: 0
tcp checksum errors: 0

packets sent: 82
packets received: 82

lo0:   addr 127.0.0.1  netmask 255.0.0.0   up
DST:   127.0.0.0      NETMASK: 255.0.0.0   GATEWAY: lo0
```

```
TCP 127.0.0.1.1227      > 127.0.0.1.6000      ESTABLISHED snd      0 rcv      0
TCP 127.0.0.1.6000     > 127.0.0.1.1227     ESTABLISHED snd      0 rcv      0
TCP 0.0.0.0.6000       > *                   LISTEN
```

Вы можете использовать утилиты командной строки, например, для драйвера манипулятора робота. Драйвер может зарегистрировать имя `/dev/robot/arm/angle` и любая запись в это устройство будет трактоваться, как угол для установки манипулятора робота. Чтобы протестировать драйвер из командной строки, Вы введёте:

```
echo 87 > /dev/robot/arm/angle
```

Утилита `echo` открывает `/dev/robot/arm/angle` и пишет в него строку ("87"). Драйвер обрабатывает запись путём установки манипулятора робота на угол 87°. Заметьте, что это было достигнуто без написания специальной тестовой программы.

Другим примером могут быть имена, такие как `/dev/robot/register/r1,r2,...`. Чтение из этих имён возвращает содержание соответствующих регистров, а запись значений приводит к установке значений регистров.

Что творится внутри?

Несмотря на то, что Вы будете использовать API менеджера ресурсов, который скрывает от Вас множество деталей, всё равно важно понимать, что же творится у него внутри. Допустим, Ваш менеджер ресурсов является сервером, содержащим цикл с вызовом `MsgReceive()` и клиенты посылают ему сообщения, используя `MsgSend*()`. Это означает, что Вы должны регулярно и своевременно отвечать клиентам, или оставлять их заблокированными, но при этом сохранять идентификатор процесса, отославшего сообщение (`rcvid`), чтобы использовать его позднее для ответа.

Для более полного понимания обсуждаемого предмета, рассмотрим события, происходящие внутри процесса-клиента и в менеджере ресурсов:

- Что творится внутри клиента.
- Что творится внутри менеджера ресурсов.

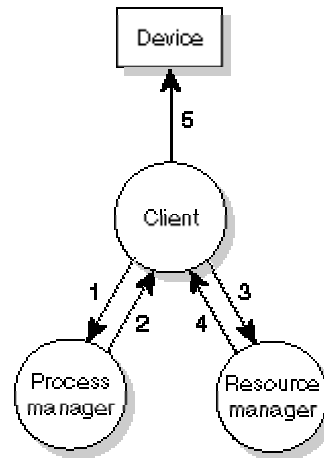
Что творится внутри клиента

Когда клиент вызывает функцию, которая требует разрешения (анализа) пространства имён (например, `open()`, `rename()`, `stat()` или `unlink()`), функция посылает сообщения и процессу, и менеджеру ресурсов, для получения файлового дескриптора. Когда файловый дескриптор получен, клиент может использовать его для передачи сообщений непосредственно в устройство, связанное с этим именем.

В приведенном ниже фрагменте клиент получает файловый дескриптор и затем пишет непосредственно в устройство:

```
/* этот этап включает переговоры клиента */
/* с администратором процессов и менеджером ресурсов */
fd = open("dev/ser1", O_RDWR);
/* этот этап включает переговоры клиента */
/* с менеджером ресурсов */
for (packet=0; packet<npackets; packet++)
write(fd, packets[packet], PACKET_SIZE);
close (fd);
```

Давайте посмотрим, что же происходит «за кулисами» в вышеприведенном примере. Предположим, что последовательный порт управляется менеджером ресурсов `devc-ser8250`, который зарегистрировал имя `/dev/ser1`:



Что происходит «за кулисами» между клиентом, администратором процессов и менеджером ресурсов

1. Библиотека клиента посылает сообщение-«запрос». Функция `open()` библиотеки клиента посылает сообщение администратору процессов с запросом на поиск имени в пространстве имен путей (например, `/dev/ser1`).
2. Администратор процессов находит процесс, связанный с данным префиксом пространства имен и возвращает `nd`, `pid`, `chid` и `handle` этого процесса.

Вот что произошло «за кулисами»...

Когда менеджер ресурсов `devc-ser8250` регистрирует своё имя (`/dev/ser1`) в пространстве имён, он вызывает администратор процессов. Администратор процессов отвечает за поддержку информации о префиксах пространства имен путей. При регистрации имени он добавляет в свою таблицу элемент, который выглядит примерно так:

```
0, 47167, 1, 0, 0, /dev/ser1
```

Элементы таблицы представляют:

- Дескриптор узла (`nd`)
- Идентификатор процесса-менеджера ресурсов (`pid`)
- Идентификатор канала, по которому менеджер ресурсов получает сообщения (`chid`)
- Дескриптор (`handle`)
- Тип открытия (`open type`)
- Префикс имени в пространстве имен путей (`name`)

Менеджер ресурсов уникально идентифицируется тремя параметрами – дескриптором узла, идентификатором процесса и идентификатором канала. Элемент в таблице администратора процессов связывает менеджер ресурса с именем,

обработчиком (для того, чтобы различать несколько имён, если менеджер ресурсов регистрирует более одного имени) и типом открытия.

Когда библиотека клиента делает запрос (шаг 1 на рисунке), администратор процессов просматривает свои таблицы в поисках зарегистрированных префиксов имён путей, совпадающих с заданным именем. Однако если до этого другой менеджер ресурсов зарегистрировал имя пути / (примечание – в реальной жизни это *procnto* и файловая система), то будет найдено более одного совпадения. Таким образом, в этом случае совпадают и /, и /dev/ser1. Администратор процессов ответит запросу *open()*, предоставив список серверов или менеджеров ресурсов, соответствующих заданному имени пути. Далее серверы по очереди будут опрошены на предмет обработки ими данного имени пути, причём первым будет тот сервер, у которого совпадение имени наиболее длинное.

3. Библиотека клиента посылает сообщение "соединения" менеджеру ресурсов. Чтобы сделать это, она должна создать соединение по каналу менеджера ресурсов:

```
fd=ConnectAttach(nd, pid, chid, 0, 0);
```

Файловый дескриптор, возвращаемый функцией *ConnectAttach()*, также является идентификатором соединения и используется для отправки сообщений непосредственно менеджеру ресурсов. В этом случае он используется для отправки сообщения соединения, содержащего дескриптор (`_IO_CONNECT` определён в `<sys/iomsg.h>`) менеджеру ресурсов с запросом на открытие /dev/ser1.

Когда менеджер ресурсов получает сообщение соединения, он выполняет проверку правомерности использования режимов доступа, определённых в вызове функции *open()*, (т.е. не пытаетесь ли Вы писать в устройство, предназначенное только для чтения, и т.д.).

4. Менеджер ресурсов в основном отвечает разрешением на доступ (и функция *open()* возвращает дескриптор файла) или сообщением о неудаче (тогда опрашивается следующий сервер).
5. Когда файловый дескриптор получен, клиент может использовать его, чтобы посылать сообщения непосредственно на устройство, соответствующее данному имени пути.

В коде примера это выглядит так, будто клиент открывает устройство и пишет непосредственно в него. Фактически же функция *write()* вызывает отсылку сообщения `_IO_WRITE` менеджеру ресурсов с запросом на запись переданных данных, и менеджер ресурсов отвечает, что он либо записал часть или все данные, либо запись не удалась.

В завершение клиент выполняет вызов функции *close()*, которая посылает сообщение `_IO_CLOSE_DUP` менеджеру ресурсов. Менеджер ресурсов обрабатывает его, исполняя какие-либо завершающие действия.

Что твориться внутри менеджера ресурса

Менеджер ресурсов является сервером, который использует протокол передачи *send/receive/reply* для получения сообщений и ответа на них. Ниже приводится псевдокод менеджера ресурсов.

```
инициализация менеджера ресурсов
регистрация имени у администратора процессов
DO всегда
получение сообщения
```

```

SWITCH по типу сообщения
CASE _IO_CONNECT:
    вызов обработчика io_open
ENDCASE
CASE _IO_READ:
    вызов обработчика io_read
ENDCASE
CASE _IO_WRITE
    вызов обработчика io_write
ENDCASE
/* прочие обработки всех остальных */
/* сообщений, которые могут появиться */
ENDSWITCH
ENDDO

```

Большинство деталей приведенного выше псевдокода скрыты в библиотеке менеджера ресурсов, которой Вы будете пользоваться. Например, на самом деле Вы не будете вызывать функции *MsgReceive**() – Вы будете вызывать библиотечные функции, такие как *resmgr_block()* или *dispath_block()*, которые сделают это за Вас. Если Вы пишете однопоточный менеджер ресурсов, то можете предусмотреть цикл обработки сообщений, но если Вы пишете многопоточный менеджер ресурсов, то цикл от Вас скрыт.

Вам не надо знать формат всех возможных сообщений, и Вы не будете обрабатывать их. Вместо этого Вы регистрируете "функции-обработчики", и когда поступает сообщение соответствующего типа, библиотека вызывает Ваш обработчик. Предположим, Вы хотите, чтобы клиент получал данные с использованием функции *read()* – Вы пишете обработчик, который вызывается каждый раз, когда приходит сообщение *_IO_READ*. Поскольку Ваш обработчик обрабатывает сообщение *_IO_READ*, мы и будем называть его "*обработчик io_read*".

Итак, библиотека менеджера ресурсов:

- получает сообщение
- проверяет, является ли оно сообщением типа *_IO_READ*
- вызывает Ваш обработчик *io_read*.

Однако Вы все же должны реализовать ответ на сообщение *_IO_READ*. Вы можете выполнить это внутри обработчика *io_read* или позже, когда будут получены данные (возможно как результат прерывания от некоего аппаратного устройства, генерирующего данные).

Библиотека выполняет обработку по умолчанию для всех сообщений, которые Вы не желаете обрабатывать самостоятельно. Плюс ко всему, большинство менеджеров ресурсов обычно не предоставляют клиенту POSIX-совместимую файловую систему. При написании менеджера ресурсов, лучше сконцентрироваться на коде, реализующем обмен данными с управляемым Вами устройством. Зачем тратить кучу времени на код, который предоставлял бы клиенту POSIX-совместимую файловую систему? Это отдельный вопрос.

Типы менеджеров ресурсов

Чтобы предоставить клиенту POSIX-совместимую файловую систему, нужно приложить немалые усилия, поэтому менеджеры ресурсов можно разделить на два *отдельных* типа:

- Менеджеры ресурсов устройств.
- Менеджеры ресурсов файловой системы.

Менеджеры ресурсов устройств

Менеджеры ресурсов устройств создают в файловой системе отдельные имена путей, каждое из которых регистрируется администратором процессов. Каждое имя обычно представляет отдельное устройство. Такие менеджеры ресурсов обычно полагаются на библиотеку, которая и выполняет большую часть работы по представлению POSIX-устройства пользователю.

Например, драйвер последовательного порта регистрирует имена `/dev/ser1` и `/dev/ser2`. Когда пользователь вызывает команду `"ls -l /dev"`, библиотека сама выполняет необходимую обработку, чтобы ответить сообщениями `_IO_STAT` с надлежащей информацией. Тот, кто пишет драйвер последовательного порта, может вместо этого сконцентрироваться на деталях управления аппаратным обеспечением порта.

Менеджеры ресурсов файловой системы

Менеджеры ресурсов файловой системы регистрируют в администраторе процессов точку монтирования. Точка монтирования – это часть пути, зарегистрированного администратором процессов. Оставшаяся часть пути управляется менеджером ресурсов файловой системы. Например, когда менеджер ресурсов файловой системы подключает точку монтирования как `/mount`, то путь `/mount/home/thomasf` анализируется следующим образом:

<code>/mount/</code>	Идентифицирует точку монтирования, управляемую администратором процессов
<code>home/thomasf</code>	Идентифицирует оставшуюся часть как управляемую менеджером ресурсов файловой системы

Примерами использования менеджера ресурсов файловой системы являются:

- Драйверы файловой системы флэш-памяти
- Процесс архивной файловой системы `tar`, который представляет содержание `tar`-файла как файловой системы, в которую пользователь может "входить" по команде `cd` и получать выходные данные по `ls`.
- Процесс управления почтовым ящиком, который регистрирует имя `/mailboxes` и управляет личными почтовыми ящиками, которые выглядят как директории и файлы, содержащие реальные сообщения.

Компоненты менеджера ресурсов

Менеджер ресурсов состоит из нескольких уровней:

- уровень функций ввода/вывода `iofunc` (верхний уровень)
- уровень управления сообщениями `resmgr`
- уровень диспетчеризации `dispatch`
- уровень пула потоков (нижний уровень).

Уровень функций ввода/вывода `iofunc`

Верхний уровень состоит из набора функций, которые ведут обработку большинства деталей POSIX-совместимой файловой системы – они обеспечивают POSIX-персонализацию ресурса. Если Вы пишете менеджер ресурсов устройства, Вы будете использовать этот уровень,

поскольку он позволяет не заботиться о деталях, связанных с представлением POSIX-совместимой файловой системы.

Этот уровень состоит из обработчиков по умолчанию, которые библиотека менеджера ресурсов использует в тех случаях, когда Вы не написали собственный обработчик. Например, если Вы не написали обработчик `io_open`, то вызывается функция `iofunc_open_default()`.

Этот уровень содержит также вспомогательные функции, вызываемые обработчиками по умолчанию. Если Вы перекрываете обработчик по умолчанию своим собственным, Вы можете, тем не менее, вызывать эти вспомогательные функции. Например, если Вы пишете свой собственный обработчик `io_read`, Вы можете при его старте вызвать функцию `iofunc_read_verify()`, чтобы проверить, открыт ли клиент на чтение.

Имена функций и структур для этого уровня имеют форму `iofunc_*`. Заголовочный файл называется `<sys/iofunc.h>`.

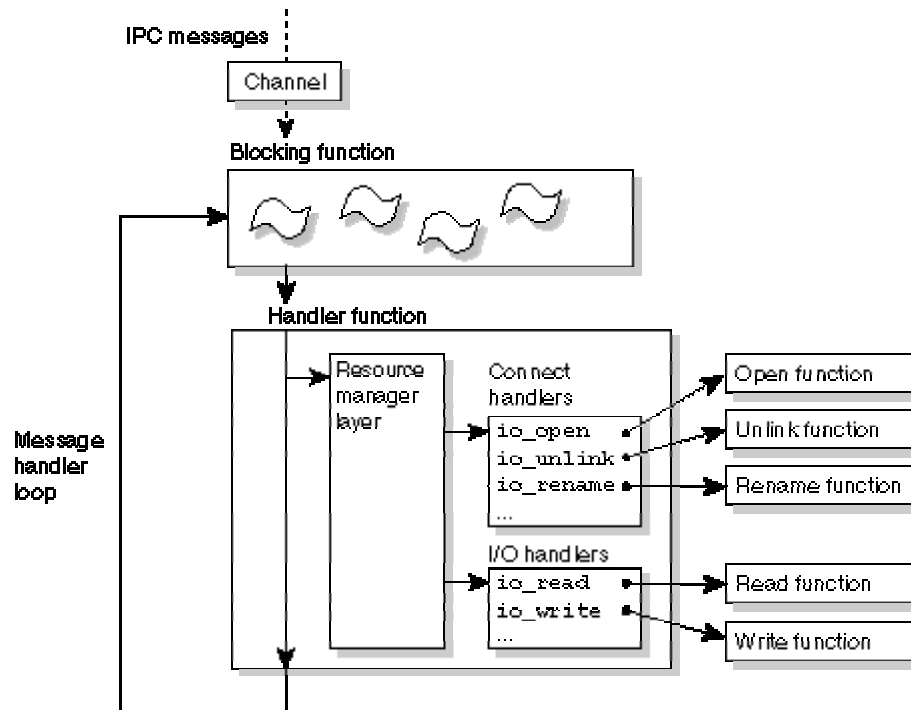
Уровень управления сообщениями `resmgr`

Этот уровень управляет большинством деталей библиотеки менеджера ресурсов:

- проверяет входящие сообщения
- вызывает для обработки сообщения соответствующий обработчик.

Если Вы не используете этот уровень, то Вам придётся производить синтаксический анализ сообщения самостоятельно. Большинство менеджеров ресурсов используют этот уровень.

Имена функций и структур этого уровня имеют форму `resmgr_*`. Заголовочный файл называется `<sys/resmgr.h>`.



Уровень `resmgr` может использоваться для обработки `_IO_*` сообщений

Уровень диспетчеризации *dispatch*

Этот уровень действует как общая точка блокирования различных типов событий. На этом уровне Вы можете обрабатывать:

IO* сообщения

Для этого он использует уровень *resmgr*

select

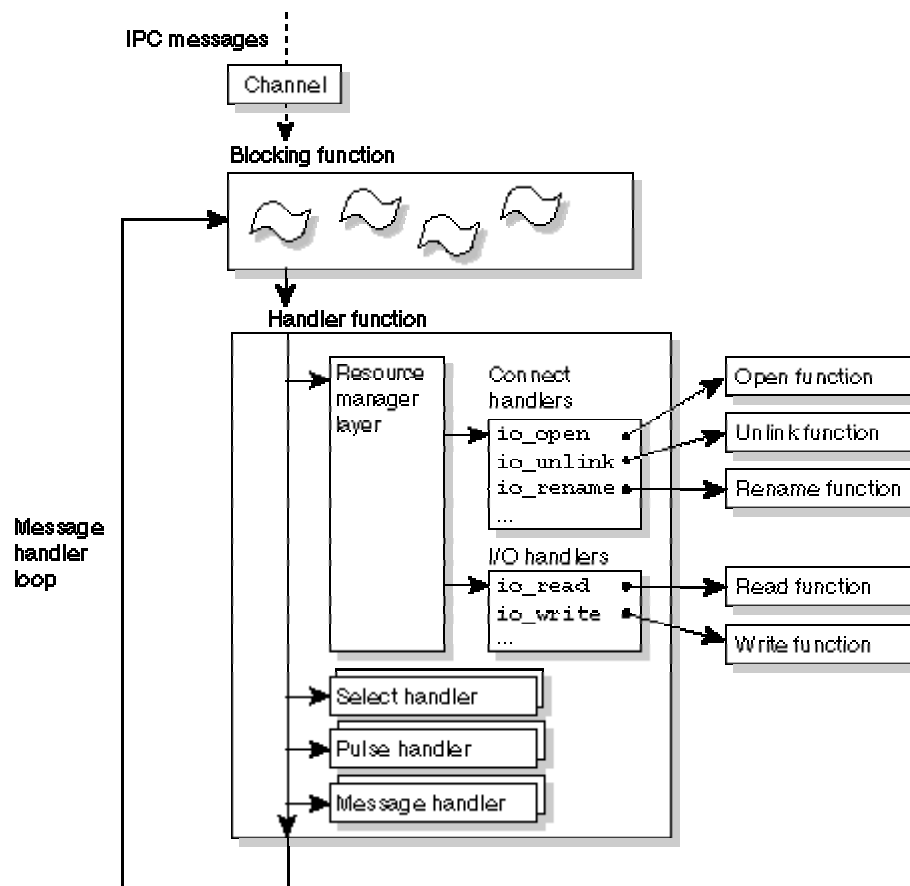
Процессы, работающие по TCP/IP, часто вызывают функцию *select()* для блокирования в ожидании прихода пакетов, или для освобождения места под запись большого объема данных. При наличии уровня *dispatch* Вы регистрируете функцию обработчика, вызываемую при получении пакета. Для этого используются функции *select_**() .

импульсы

Как и на других уровнях, Вы регистрируете функцию обработчика, вызываемую по прибытии определенного импульса. Для этого используются функции *pulse_**() .

другие сообщения

Вы можете задать для уровня *dispatch* диапазон типов сообщений, которые Вы будете поддерживать, и указать обработчик. Если приходит сообщение, и первые несколько байтов в нем содержат тип, принадлежащий заданному диапазону, уровень *dispatch* вызывает Ваш обработчик. Для этого используются функции *message_**() .



Уровень `dispatch` может использоваться для обработки `_IO_`сообщений, `select`, импульсов и других сообщений*

Ниже описан способ, которым сообщения обрабатываются на `dispatch` уровне (или более точно, функцией `dispatch_handler()`). В зависимости от типа блокирования, обработчик может вызывать подсистему функций `message_*`(`.`). Поиск основан на типе сообщения или коде импульса для сопоставления функции, которая была подсоединена с использованием функций `message_attach()` или `pulse_attach()`. Если совпадение найдено, вызывается присоединённая функция.

Если тип сообщения находится в диапазоне, обрабатываемом менеджером ресурсов (сообщение ввода/вывода) и пути имён присоединены с использованием `resmgr_attach()`, вызывается подсистема менеджера ресурсов и сообщение обрабатывается. Когда приходит импульс, он может быть обработан подсистемой менеджера ресурсов, если его код – один из кодов, обрабатываемых менеджером ресурсов (импульсы `UNBLOCK` и `DISCONNECT`). Если была выполнена функция `select_attach()` и импульс совпал с используемым `select`, то вызывается подсистема `select`, которая диспетчеризует это событие.

Если сообщение получено, но для этого типа сообщения не нашлось соответствующего обработчика, возвращается `MsgError(ENOSYS)`, чтобы разблокировать клиента, пославшего сообщение.

Уровень пула потоков

Этот уровень позволяет Вам реализовать однопоточный или многопоточный менеджер ресурсов. Это означает, что один поток может обрабатывать запросы `write()`, тогда как другой – запросы `read()`.

Для использования потоков Вы должны предоставить функцию блокирования и функцию обработки, которая вызывается, когда функция блокирования возвращает управление. Чаще всего Вы передаёте это функциям уровня диспетчеризации. Однако Вы можете передать это функциям уровня `resmgr` или своим собственным.

Этот уровень может быть использован независимо от уровня менеджера ресурсов.

Пример простого менеджера ресурсов устройства

Ниже приводятся два *полных*, но *простых* примера менеджера ресурсов устройства:

- однопоточный менеджер ресурсов устройства
- многопоточный менеджер ресурсов устройства

i Читая эту главу, Вы увидите много фрагментов кода. Большая часть кода написана так, что может быть скомбинирована с любым из этих двух простых менеджеров ресурсов.

Оба менеджера ресурсов устройства функционируют аналогично устройству `/dev/null`:

- функция `open()` работает всегда
- функция `read()` возвращает нулевое количество байтов (указывая EOF)

- функция `write()` работает с любым количеством байтов (со сбросом данных)
- работает несколько других функций POSIX (например, `chown()`, `chmod()`, `lseek()`)

Пример однопоточного менеджера ресурсов устройства

Вот полный код однопоточного менеджера ресурсов устройства:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

main(int argc, char **argv)
{
    /* объявление переменных, которые мы будем использовать */
    resmgr_attr_t                resmgr_attr;
    dispatch_t                  *dpp;
    dispatch_context_t          *ctp;
    int                          id;

    /* инициализация интерфейса диспетчеризации */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Невозможно разместить обработчик диспетчеризации.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    /* инициализация атрибутов менеджера ресурсов */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* инициализация функций обработки сообщений */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
        _RESMGR_IO_NFUNCS, &io_funcs);

    /* инициализация структуры атрибутов, используемой устройством */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

    /* прикрепление нашего имени устройства */
    id = resmgr_attach(dpp,
        /* обработчик диспетчеризации */
        &resmgr_attr,
        /* атрибуты менеджера ресурсов */
        "/dev/sample",
        /* имя устройства */
        _FTYPE_ANY,
        /* тип открытия */
        0,
        /* флаги */
        &connect_funcs,
        /* подпрограммы связи */
        &io_funcs,
        /* подпрограммы ввода/вывода */
        &attr);
    /* идентификатор-описатель */

    if(id == -1) {
        fprintf(stderr, "%s: Невозможно прикрепить имя.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* размещение контекстной структуры */
    ctp = dispatch_context_alloc(dpp);
}
```

```

/* запуск цикла сообщений менеджера ресурсов */
while(1) {
    if((ctp = dispatch_block(ctp)) == NULL) {
        fprintf(stderr, "ошибка блока\n");
        return EXIT_FAILURE;
    }
    dispatch_handler(ctp);
}
}

```

i Включайте заголовок `<sys/dispatch.h>` после `<sys/iofunc.h>`, чтобы исключить предупреждающие сообщения (warnings) о переопределении членов в некоторых функциях.

Давайте проанализируем пример кода шаг за шагом:

- Инициализация интерфейса диспетчеризации.
- Инициализация атрибутов менеджера ресурсов.
- Инициализация функций, используемых для обработки сообщений.
- Инициализация структуры атрибутов, используемых устройством.
- Размещение имени в пространстве имён.
- Запуск цикла обработки сообщений менеджера ресурсов.

Инициализация интерфейса диспетчеризации

```

/* инициализация интерфейса диспетчеризации */
if((dpp = dispatch_create()) == NULL)
{
    fprintf(stderr, "%s: Невозможно разместить обработчик диспетчеризации.\n",
            argv[0]);
    return EXIT_FAILURE;
}

```

Нам необходимо создать некий механизм, с помощью которого клиент мог бы посылать сообщения менеджеру ресурсов. Это делается посредством вызова функции `dispatch_create()`, которая создаёт и возвращает структуру диспетчеризации. Эта структура содержит идентификатор канала. Заметьте, что идентификатор канала в действительности не создаётся до тех пор, пока Вы что-либо не присоедините – в `resmgr_attach()`, `message_attach()` или `pulse_attach()`.

i Эта структура является "непрозрачной" и её содержание не может быть доступно «напрямую». Для создания связи с использованием этого скрытого идентификатора канала используется функция `message_connect()`.

Инициализация атрибутов менеджера ресурсов

```

/* инициализация атрибутов менеджера ресурсов */
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

```

Структура атрибутов менеджера ресурсов используется для конфигурирования:

- количества структур IOV (векторов ввода/вывода) доступных для ответа сервера (`nparts_max`)

- максимального размера буфера получения сообщений (*msg_max_size*)

Инициализация функций, используемых для обработки сообщений

```
/* инициализация функций обработки сообщений */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                _RESMGR_IO_NFUNCS, &io_funcs);
```

Здесь мы предоставляем две таблицы (структуры), в которых указывается, какие функции вызываются, когда приходит конкретное сообщение:

- таблица функций связи
- таблица функций ввода/вывода.

Вместо того, чтобы заполнять эти таблицы вручную, мы используем вызов функции *iofunc_func_init()*, чтобы разместить функции-обработчики *iofunc_*_default()* в соответствующих местах.

Инициализация структуры атрибутов, используемой устройством

```
/* инициализация структуры атрибутов, используемой устройством */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
```

Структура атрибутов содержит информацию о нашем конкретном устройстве, связанном с именем */dev/sample*. Она содержит, по меньшей мере, следующую информацию:

- права доступа и тип устройства
- идентификаторы владельца и группы

Это поимённая структура данных (на каждое имя устройства имеется по одной атрибутной записи). Позже мы увидим, как Вы можете расширить структуру, включая свою собственную информацию отдельно по каждому устройству.

Размещение имени в пространстве имён

```
/* прикрепление нашего имени устройства */
id = resmgr_attach(dpp, /* обработчик диспетчеризации */
                  &resmgr_attr, /* атрибуты менеджера ресурсов */
                  "/dev/sample", /* имя устройства */
                  _FTYPE_ANY, /* тип открытия */
                  0, /* флаги */
                  &connect_funcs, /* подпрограммы связи */
                  &io_funcs, /* подпрограммы ввода/вывода */
                  &attr); /* идентификатор-описатель */

if(id == -1) {
    fprintf(stderr, "%s: Невозможно прикрепить имя.\n", argv[0]);
    return EXIT_FAILURE;
}
```

Прежде чем менеджер ресурсов сможет получать сообщения от других программ, ему необходимо информировать их (с помощью администратора процессов), что он является ответственным за конкретное имя в пространстве путей. Это делается посредством регистрации имени пути. После регистрации другие процессы могут найти этот процесс и связаться с ним, используя зарегистрированное имя. В этом примере менеджером ресурсов, называемым *devc-xxx*, может управляться последовательный порт, но реальный ресурс зарегистрирован в

пространстве имён как `/dev/sample`. Поэтому, когда программа затребует службу последовательного порта, она открывает порт `/dev/sample`.

Рассмотрим параметры по очереди, пропуская те, что мы уже обсудили.

device name (имя устройства)

Имя, ассоциированное с устройством (т.е. `/dev/sample`)

open type (тип открытия)

Задаёт значение, равное константе `_FTYPE_ANY`. Это сообщает администратору процессов, что наш менеджер ресурсов будет принимать любой тип запроса открытия – мы не ограничиваем виды связи, которые будут обрабатываться.

Некоторые менеджеры ресурсов принимают только ограниченные типы запросов на открытие, которые они обрабатывают. Например, менеджер ресурсов очереди сообщений POSIX (*mqqueue*) позволяет открывать сообщения только типа `_FTYPE_QUEUE`.

flags (флаги)

Управляет поведением определения пути имени файла, выполняемым администратором процессов. Установкой значения в ноль мы позволяем осуществлять запросы только по имени `"dev/sample"`.

Размещение контекстной структуры

```
/* размещение контекстной структуры */
ctp = dispatch_context_alloc(dpp);
```

Контекстная структура содержит буфер, где будут размещаться получаемые сообщения. Размер буфера был установлен, когда мы инициализировали структуру атрибутов менеджера ресурсов. Контекстная структура также содержит буфер векторов ввода/вывода, которые библиотека менеджера может использовать для ответов на сообщения. Число векторов ввода/вывода было установлено, когда мы инициализировали структуру атрибутов менеджера ресурсов.

Запуск цикла сообщений менеджера ресурсов

```
/* запуск цикла сообщений менеджера ресурсов */
while(1) {
    if((ctp = dispatch_block(ctp)) == NULL) {
        fprintf(stderr, "ошибка блока\n");
        return EXIT_FAILURE;
    }
    dispatch_handler(ctp);
}
```

После того, как менеджер ресурсов регистрирует своё имя, он переходит в ожидание получения сообщений, приходящих, когда клиент пытается выполнить операцию (например, `open()`, `read()`, `write()`) с этим именем. В нашем примере, как только имя `/dev/sample` будет зарегистрировано и клиентская программа выполнит:

```
fd=open("dev/sample", O_RDONLY);
```

библиотека клиента создаёт сообщение `_IO_CONNECT`, которое посылается нашему менеджеру ресурсов. Он получает это сообщение внутри функции `dispatch_block()`. Затем мы вызываем функцию `dispatch_handler()`, которая декодирует это сообщение и вызывает соответствующую функцию обработки, основываясь на таблицах функций связи и ввода/вывода, которые мы предварительно передали. После возврата из функции `dispatch_handler()` мы попадаем обратно в функцию `dispatch_block()` на ожидание следующего сообщения.

Некоторое время спустя, когда клиентская программа исполняет:

```
read(fd, buf, BUFSIZ);
```

библиотека клиента создаёт сообщение `_IO_READ`, которое затем посылается нашему менеджеру ресурсов, и цикл декодирования повторяется.

Пример многопоточного менеджера ресурсов устройства

Вот полный код простого многопоточного менеджера ресурсов устройства:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>

/*
 * определение параметра THREAD_POOL_PARAM_T так, чтобы мы могли
 * избежать появления предупреждающих сообщений компилятора при
 * использовании далее функции dispatch_*()
 */
#define THREAD_POOL_PARAM_T    dispatch_context_t

#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

main(int argc, char **argv)
{
    /* объявление переменных, которые мы будем использовать */
    thread_pool_attr_t    pool_attr;
    resmgr_attr_t        resmgr_attr;
    dispatch_t            *dpp;
    thread_pool_t        *tpp;
    dispatch_context_t    *ctp;
    int                    id;

    /* инициализация интерфейса диспетчеризации */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Невозможно разместить обработчик диспетчеризации.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    /* инициализация атрибутов менеджера ресурсов */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* инициализация функций обработки сообщений */
```



```

iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                _RESMGR_IO_NFUNCS, &io_funcs);

/* инициализация структуры атрибутов, используемой устройством */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

/* прикрепление нашего имени устройства */
id = resmgr_attach(dpp, /* обработчик диспетчеризации */
                  &resmgr_attr, /* атрибуты менеджера ресурсов */
                  "/dev/sample", /* имя устройства */
                  _FTYPE_ANY, /* тип открытия */
                  0, /* флаги */
                  &connect_funcs, /* подпрограммы связи */
                  &io_funcs, /* подпрограммы ввода/вывода */
                  &attr); /* идентификатор-описатель */

if(id == -1) {
    fprintf(stderr, "%s: Невозможно прикрепить имя.\n", argv[0]);
    return EXIT_FAILURE;
}

/* инициализация атрибутов пула потоков */
memset(&pool_attr, 0, sizeof pool_attr);
pool_attr.handle = dpp;
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

/* размещение дескриптора пула потоков */
if((tpp = thread_pool_create(&pool_attr, POOL_FLAG_EXIT_SELF)) == NULL) {
    fprintf(stderr, "%s: Невозможно инициализировать пул потоков.\n",
            argv[0]);
    return EXIT_FAILURE;
}

/* запуск потоков, без возврата */
thread_pool_start(tpp);
}

```

Большая часть кода та же, что и в примере однопоточного менеджера ресурсов, так что мы обсудим лишь то, что не было описано выше. К тому же мы более детально рассмотрим многопоточные менеджеры ресурсов ниже в этой главе, так что здесь подробности сведены к минимуму.

Мы рассмотрим следующие этапы:

- Определение параметра `THREAD_POOL_PARAM_T`.
- Инициализация атрибутов пула потоков.
- Размещение указателя на структуру пула потоков.
- Запуск потоков.

В коде этого примера потоки для своих циклов блокирования используют функции *dispatch_**(т.е. уровень диспетчеризации).

Определение параметра `THREAD_POOL_PARAM_T`

```

/*
 * определение параметра THREAD_POOL_PARAM_T так чтобы мы могли
 * избежать появления предупреждающих сообщений компилятора при

```

```

* использовании далее функции dispatch_*()
*/
#define THREAD_POOL_PARAM_T dispatch_context_t

#include <sys/iofunc.h>
#include <sys/dispatch.h>

```

Декларация `THREAD_POOL_PARAM_T` указывает компилятору, какой тип параметра передаётся между различными функциями блокировки/обработки, которые будут использовать потоки. Этот параметр будет использоваться контекстной структурой для передачи информации между функциями. По умолчанию он определён как `resmgr_context_t`, но поскольку этот пример использует уровень диспетчеризации, нам необходимо установить его в `dispatch_context_t`. Мы определяем его до того, как включаем заголовочные файлы, поскольку они ссылаются на этот параметр.

Инициализация атрибутов пула потоков

```

/* инициализация атрибутов пула потоков */
memset(&pool_attr, 0, sizeof pool_attr);
pool_attr.handle = dpp;
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

```

Атрибуты пула указывают потокам, какие функции используются для создания циклов блокирования и управления, сколько потоков может существовать в любой момент времени. Более полно мы рассмотрим эти атрибуты, когда будем говорить подробнее о многопоточных менеджерах ресурсов ниже в этой главе.

Размещение указателя на структуру пула потоков

```

/* размещение дескриптора пула потоков */
if((tpp = thread_pool_create(&pool_attr, _FLAG_EXIT_SELF)) == NULL) {
    fprintf(stderr, "%s:Невозможно инициализировать пул потоков.\n",
            argv[0]);
    return EXIT_FAILURE;
}

```

Указатель на управляющую структуру используется для управления пулом потоков. Среди всего прочего, она (структура) содержит заданные атрибуты и флаги. Размещает этот указатель и заполняет структуру функция `thread_pool_create()`.

Запуск потоков

```

/* запуск потоков, без возврата */
thread_pool_start(tpp);

```

Функция *thread_pool_start()* запускает пул потоков. Каждый вновь созданный поток размещает контекстную структуру, используя функцию *context_allock()*, которую мы задали выше в структуре атрибутов. Затем потоки будут заблокированы на функции *block_func()*, и когда функция *block_func()* вернёт управление, они вызовут функцию *handler_func()*. Обе эти функции также определены через структуру атрибутов. Каждый поток в сущности делает то же, что однопоточный менеджер ресурсов, приведенный выше, в своем цикле обработки событий.

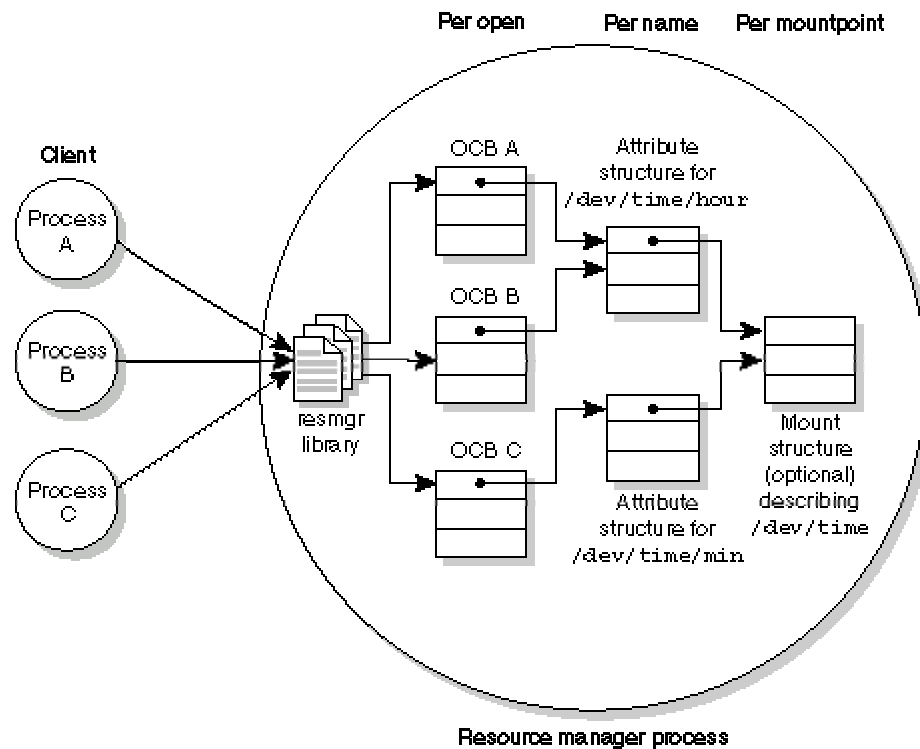
С этого момента Ваш менеджер ресурсов готов обрабатывать сообщения. Поскольку для функции *thread_pool_create()* мы задали флаг `POOL_FLAGS_EXIT_SELF`, то после того, как были запущены потоки, будет вызвана функция *pthread_exit()* и этот поток (выполнивший вызов *thread_pool_create()*) будет завершён.

Структуры, содержащие данные

Библиотека менеджера ресурсов описывает несколько ключевых структур для хранения данных:

- структура блока открытого контекста (*open control block, ocb*), содержащая данные по каждому открытому дескриптору файла
- структура атрибутов, содержащая данные по каждому имени
- структура точек монтирования, содержащая данные по каждой точке монтирования (обычно менеджер ресурсов устройства не будет иметь структуры точек монтирования).

Этот рисунок может помочь разъяснить их взаимосвязи:



Несколько клиентов связаны в одну структуру монтирования с несколькими ocb

Структура блока открытого контекста (ocb)

Блок открытого контекста (*ocb*) содержит информацию о состоянии конкретной сессии, относящейся к клиенту и менеджеру ресурсов. Он создается во время обработки открытия и существует до тех пор, пока не будет выполнено закрытие.

Эта структура используется функциями *iofunc* на уровне вспомогательных функций. (Позже мы покажем, как ее расширить, включив Ваши собственные данные).

Структура *ocb* содержит по меньшей мере следующее:

```
typedef struct _iofunc_ocb {
    IOFUNC_ATTR_T    *attr;
    int32_t          ioflag;
    off_t            offset;
    uint16_t         sflag;
    uint16_t         flags;
} iofunc_ocb_t;
```

где значения представляют:

attr

Указатель на структуру атрибутов (см. ниже)

ioflag

Содержит режим (т.е. чтение, запись, блокировка), в котором ресурс был открыт. Эта информация наследуется из структуры *io_connect_t*, которая доступна в сообщении, передаваемом открытому обработчику.

offset

Доступен пользователю для изменения. Определяет смещение для чтения/записи ресурса (например, *lseek()* текущая позиция внутри файла).

sflag

Определяет режим совместного доступа (разделяемого использования). Эта информация наследуется из структуры *io_connect_t*, которая доступна в сообщении, передаваемом открытому обработчику.

flags

Доступен пользователю для изменения. Когда бит IOFUNC_OCB_PRIVILEGED установлен, это указывает, что *open()* выполняется привилегированным процессом (т.е. *root*). Вы можете использовать флаги в диапазоне IOFUNC_OCB_FLAGS_PRIVATE (см. <sys/iofunc.h>) для своих собственных целей.

Структура атрибутов

Это структура, определяющая характеристики устройства, для которого предназначен менеджер ресурсов. Она используется в связке со структурой блока *ocb*. Структура атрибутов содержит следующее:

```
typedef struct _iofunc_attr {
    IOFUNC_MOUNT_T    *mount;
    uint32_t          flags;
    int32_t           lock_tid;
    uint16_t          lock_count;
    uint16_t          count;
    uint16_t          rcount;
    uint16_t          wcount;
    uint16_t          rlocks;
    uint16_t          wlocks;
    struct _iofunc_mmap_list    *mmap_list;
    struct _iofunc_lock_list    *lock_list;
    void                    *list;
    uint32_t                list_size;
    off_t                   nbytes;
    ino_t                   inode;
    uid_t                   uid;
    gid_t                   gid;
    time_t                  mtime;
    time_t                  atime;
    time_t                  ctime;
    mode_t                  mode;
    nlink_t                 nlink;
    dev_t                   rdev;
} iofunc_attr_t;
```

Значения элементов структуры:

****mount***

Указатель на структуру точки монтирования

flags

Имеет побитовое представление и содержит следующие флаги, описывающие состояние других полей структуры:

IOFUNC_ATTR_ETIME

Время доступа больше не действительно. Обычно устанавливается при чтении ресурса.

IOFUNC_ATTR_STIME

Время изменения состояния больше не действительно. Обычно устанавливается при изменении информации о файле.

IOFUNC_ATTR_DIRTY_NLINK

Количество связей было изменено.

IOFUNC_ATTR_DIRTY_MODE

Режим был изменён

IOFUNC_ATTR_DIRTY_OWNER

Идентификатор пользователя или группы был изменён.

IOFUNC_ATTR_DIRTY_RDEV

Поле `rdev` было изменено, например, при вызове `mknod()`.

IOFUNC_ATTR_DIRTY_SIZE

Размер был изменён.

IOFUNC_ATTR_DIRTY_TIME

Одно из полей `mtime`, `atime` или `ctime` было изменено.

IOFUNC_ATTR_MTIME

Время модификации больше не действительно. Обычно устанавливается при записи в ресурс.

Поскольку Ваш менеджер ресурсов использует эти флаги, Вы можете сразу узнать, какие области структуры атрибутов были модифицированы вспомогательными функциями уровня *iofunc*. Таким образом, если Вам надо записать входные данные на какой-то носитель, Вы можете записать только то, что было изменено. Область флагов, которые могут быть заданы пользователем, задана в **IOFUNC_ATTR_PRIVATE** (см. `<sys/iofunc.h>`).

Подробно о том, как изменить данные в структуре атрибутов, см. ниже в разделе "Обновление времени для чтения и записи".

lock_tid* и *lock_count

Для обеспечения многопоточности в менеджере ресурсов, Вам необходимо заблокировать структуру атрибутов так, чтобы одновременно изменять её мог только один поток. Уровень менеджера ресурсов автоматически блокирует структуру атрибутов, когда вызываются некоторые функции-обработчики (т.е. `IO_*`), используя функцию `iofunc_attr_lock()`. Поле *lock_tid* структуры содержит идентификатор потока, который блокирует структуру в данный момент; *lock_count* показывает, сколько потоков пытаются использовать структуру атрибутов. (Для получения более полной информации см. функции `iofunc_attr_lock()` и `iofunc_attr_unlock()` в "Справочнике библиотечных функций").

count*, *rcount*, *wcount*, *rlocks* и *wlocks

В структуре атрибутов хранятся несколько счётчиков, значения которых увеличиваются или уменьшаются некоторыми вспомогательными функциями уровня *iofunc*. И функциональность, и конкретное содержание сообщений, полученных от клиента, определяют, на какие поля оказывается влияние.

count

количество *ocb*, использующих каким-либо образом этот атрибут. Когда счётчик равен нулю, это означает, что никто не использует этот атрибут

rcount

количество *ocb*, использующих этот атрибут для чтения

wcount

количество *ocb*, использующих этот атрибут для записи

rlocks

количество блокировок чтения, зарегистрированных на текущий момент для атрибута

wlocks

количество блокировок записи, зарегистрированных на текущий момент для атрибута.

Если в блоке *ocb* указано, что ресурс открывается на чтение и запись, то *count*, *rcount* и *wcount* все будут увеличены на единицу (см. описание функций *iofunc_attr_init()*, *iofunc_lock_default()*, *iofunc_lock()*, *iofunc_ocb_attach()*, и *iofunc_ocb_default()*).

mmap_list* и *lock_list

Поле *mmap_list* используется функциями *iofunc_mmap()* и *iofunc_mmap_default()* для управления функциональностью ресурса; поле *lock_list* используется функцией *iofunc_lock_default()*. Как правило, Вам не потребуется модифицировать или проверять значение этих полей.

list

Зарезервирован для использования на будущее.

list_size

Размер зарезервированной области *list*. Зарезервирован для использования на будущее.

nbytes

Количество байтов в ресурсе. Может изменяться пользователем. Для файла - это его размер. Для специальных устройств (например, */dev/null*), которые не поддерживают *lseek()* или имеют радикально отличающуюся интерпретацию для *lseek()*, эта область не используется (так как в таком случае Вы не будете использовать вспомогательные функции, но будете вместо них писать свои собственные). Поэтому мы рекомендуем установить это значение в ноль, за исключением случаев, когда имеется очевидная интерпретация того, что Вы хотели бы туда поместить.

inode

Это специфичное для монтирования значение номера узла (файла или ресурса), который должен быть уникальным для каждой точки монтирования. Вы можете задать своё собственное значение, или 0 (ноль), чтобы позволить администратору процессов заполнить его вместо Вас. Для приложений типа файловых систем это может согласовываться с какими-то дисковыми структурами. В любом случае интерпретация этого значения зависит от Вас.

uid* и *gid

Идентификатор пользователя и идентификатор группы владельца этого ресурса. Эти значения обновляются автоматически вспомогательными функциями *chown()* (например, функцией *iofunc_chown_default()*) и указываются в совместно с членом *mode* для предоставления прав доступа вспомогательным функциям *open()* (например, *iofunc_open_default()*)

mtime*, *atime* и *ctime

Три поля, относящиеся к представлению времени в POSIX-формате:

mtime

время последней модификации (его обновляет функция *write()*)

atime

время последнего доступа (его обновляет функция *read()*)

ctime

время последнего изменения состояния (его обновляют функции *write()*, *chmod()* и *chown()*).

i Одно или более из этих трёх полей могут оказаться недействительными в результате вызова функции уровня *iofunc*. Заполнение полей времени в структуре атрибутов позволяет избежать обращений к ядру для получения текущего времени при каждом вызове обработчика сообщения ввода/вывода.

POSIX устанавливает, что эти три поля времени должны быть действительными, когда выполняется функция *fstat()*, но они не отражают реальное время, когда произошло соответствующее событие. Кроме того, если соответствующее событие произошло между вызовами *fstat()*, времена должны измениться. Если соответствующее изменение между вызовами *fstat()* так и не произошло, то возвращаемое время должно быть тем же, что и возвращённое последний раз перед этим. Более того, если соответствующее событие произошло между вызовами *fstat()* несколько раз, то требуется только, чтобы возвращаемое время отличалось от того, что было возвращено предыдущий раз.

Имеются вспомогательные функции, которые заполняют эти поля правильными значениями времени. Вы можете вызывать их в соответствующих обработчиках, чтобы сохранять для устройства правильное время – см. функцию *iofunc_time_update()*.

mode

Содержит режим ресурса (например, тип, режимы доступа). Допустимые режимы могут быть выбраны из серий констант *S_** в файле `<sys/stat.h>`.

nlink

Число связей с этим конкретным именем. Для имён, представляющих директории, это значение должно быть больше 2. Может изменяться пользователем.

rdev

Содержит номер устройства для специального символического устройства и номер – для именованных специальных устройств.

Структура точки монтирования

Поля структуры точки монтирования, в частности члены *conf* и *flags*, изменяют поведение некоторых функций уровня *iofunc*. Эта необязательная структура содержит следующее:

```
typedef struct _iofunc_mount {
    uint32_t flags;
    uint32_t conf;
    dev_t    dev;
    int32_t  blocksize;
    iofunc_funcs_t *funcs;
} iofunc_mount_t;
```

Описание полей:

flags

Содержит только один значащий бит (объявление константы *IOFUNC_MOUNT_32BIT*), который указывает, что смещения, используемые этим менеджером ресурсов, являются 32-битными (как противоположность

расширенным 64-битным смещениям). Модифицируемые пользователем флаги монтирования определены как IOFUNC_MOUNT_FLAGS_PRIVATE (см. файл <sys/iofunc.h>).

conf

Содержит несколько битов:

IOFUNC_PC_CHOWN_RESTRICTED

Задаёт обработчик по умолчанию для сообщения `_IO_CHOWN`, чтобы поведение соответствовало определённому в POSIX как "chown-ограниченное". Никто, кроме root не может выполнить `chown()`.

IOFUNC_PC_NO_TRUNC

Указывает, что файловая система не будет проводить усечение имен. Не оказывает влияния на библиотеки уровня `iofunc`, но возвращает обработчик по умолчанию `_IO_PATHCONF` уровня `iofunc`.

IOFUNC_PC_SYNC_IO

Указывает, что файловая система поддерживает синхронные операции ввода-вывода. Если не установлен, приводит к тому, что обработчик по умолчанию `_IO_OPEN` уровня `iofunc` возвращает ошибку, если клиент задаёт любое из `O_DSYNC`, `O_RSYNC` или `O_SYNC`.

IOFUNC_PC_LINK_DIR

Управляет тем, будет позволено или нет создавать, и уничтожать (*link and unlink*) связи для каталогов.

Заметьте, что четыре опции, приведенные выше для поля `conf`, возвращаются обработчиком по умолчанию `_IO_PATHCONF` уровня `iofunc`.

dev

Содержит номер устройства для файловой системы. Этот номер возвращается в поле `st_dev` структуры `stat` для функции `stat()`.

blocksize

Содержит размер блока устройства в байтах. В менеджере ресурсов файловой системы указывает размер блока, присущий диску, например, 512 байтов.

funks

Содержит следующую структуру:

```
struct _iofunc_funks {
    unsigned        nfuncs;
    IOFUNC_OCB_T   *(*ocb_malloc) (resmgr_context_t *ctp, IOFUNC_ATTR_T *attr);
    void            (*ocb_free) (IOFUNC_OCB_T *ocb);
};
```

где:

nfuncs

размер таблицы, указывает количество функций, представленных в структуре; он должен быть заполнен объявленной константой `_IOFUNC_NFUNCS`;

ocb_malloc() и ocb_free()

Применяются для расширения блока `ocb`. Это указатели на функции выделения и освобождения `ocb` и они должны быть связаны со структурой точки монтирования. (См. раздел "Расширяемость ocb и структур атрибутов"). Если эти поля равны NULL, используются версии библиотек по умолчанию. Вы должны задать либо обе, либо ни одной из этих функций – они действуют как согласованная пара.

Обработка сообщений `_IO_READ`

Обработчик `io_read` отвечает за возврат данных клиенту после получения сообщения `_IO_READ`. Примерами функций, посылающих это сообщение, являются функции `read()`, `readdir()`, `fread()` и `fgetc()`. Давайте начнём с рассмотрения формата сообщения:

```
struct _io_read {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       nbytes;
    uint32_t      xtype;
};

typedef union {
    struct _io_read    i;
    /* unsigned char    data[nbytes]; */
    /* nbytes возвращается с MsgReply */
} io_read_t;
```

Как и для всех сообщений менеджеру ресурсов, мы определим объединение, содержащее структуру ввода (приходящего в менеджер ресурсов) и структуру ответа или вывода (возвращаемого обратно клиенту). Прототип функции `io_read()` объявлен с аргументом `io_read_t *msg` – это указатель на объединение, содержащее сообщение.

Поскольку это функция `read()`, то поле `type` имеет значение `_IO_READ`.

Интересующими нас полями в структуре ввода являются:

- `combine_len`
- `nbytes`
- `xtype`.

Поле `combine_len` имеет смысл для комбинированных сообщений – см. раздел "Комбинированные сообщения" в этой главе.

Поле `xtype` используется для изменения порядка обработки сообщений, если Ваш менеджер ресурсов поддерживает такую возможность. Даже если Ваш менеджер ресурсов это не поддерживает, Вы должны проверять это поле. Более подробно о поле `xtype` – ниже (см. раздел "`xtype`"). Более важным полем на этот момент является `nbytes`, которое указывает, сколько байтов ожидается от клиента. Мы создадим функцию-обработчик `io_read`, которая в действительности возвращает некие данные (фиксированную строку "Hello, world\n"). Будем использовать `ocb` для хранения позиции внутри буфера, который мы возвращаем клиенту.

Когда мы получаем сообщение `_IO_READ`, поле `nbytes` сообщает нам, сколько байт желает прочитать клиент. Предположим, что клиент выполнил:

```
read(fd, buf, 4096);
```

В этом случае мы просто вернем всю строку "Hello, world\n" в выходном буфере и сообщим клиенту, что мы возвращаем 13 байт, т.е. размер строки.

Однако представим случай, когда клиент выполнил следующее:

```
while(read(fd, &character, 1) != EOF) {
    printf("Получен символ \"%c\"\n", character);
```

Хотя это и не самый эффективный способ выполнить чтение! В этом случае мы будем устанавливать поле `msg->i.nbytes` в единицу (размер буфера, который клиент желает получить). Мы не можем просто вернуть клиенту всю строку за один раз – мы передаем ее побайтно. Это тот случай, когда поле `offset` из `ocb` вступает в игру.

Пример кода обработки сообщений _IO_READ

Вот реализация функции *io_read*, корректно обрабатывающая этот случай:

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>

int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);
static char *buffer = "Hello world\n";
static resmgr_connect_funcs_t connect_funcs;
static resmgr_io_funcs_t io_funcs;
static iofunc_attr_t attr;

main(int argc, char **argv)
{
    /* объявление переменных, которые мы будем использовать */
    resmgr_attr_t resmgr_attr;
    dispatch_t *dpp;
    dispatch_context_t *ctp;
    int id;

    /* инициализация интерфейса диспетчеризации */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Невозможно разместить обработчик диспетчеризации.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    /* инициализация атрибутов менеджера ресурсов */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* инициализация функций обработки сообщений */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
        _RESMGR_IO_NFUNCS, &io_funcs);
    io_funcs.read = io_read;

    /* инициализация структуры атрибутов, используемой устройством */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
    attr.nbytes = strlen(buffer) + 1;

    /* прикрепление нашего имени устройства */
    if((id = resmgr_attach(dpp, &resmgr_attr, "/dev/sample", _FTYPE_ANY, 0,
        &connect_funcs, &io_funcs, &attr)) == -1) {
        fprintf(stderr, "%s: Невозможно прикрепить имя.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* размещение контекстной структуры */
    ctp = dispatch_context_alloc(dpp);

    /* запуск цикла сообщений менеджера ресурсов */
    while(1) {
        if((ctp = dispatch_block(ctp)) == NULL) {
            fprintf(stderr, " ошибка блока \n");
            return EXIT_FAILURE;
        }
    }
}
```

```

    }
    dispatch_handler(ctp);
}
}
int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
    int         nleft;
    int         nbytes;
    int         nparts;
    int         status;

    if ((status = iofunc_read_verify (ctp, msg, ocb, NULL)) != EOK)         return (status);
    if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE)         return (ENOSYS);

    /* при всех чтениях (первом и последующих) вычисляется,
     * какое количество байтов мы можем вернуть клиенту,
     * основываясь на числе имеющихся байтов (nleft)
     * и размере буфера клиента */

    nleft = ocb->attr->nbytes - ocb->offset;
    nbytes = min (msg->i.nbytes, nleft);

    if (nbytes > 0) {
        /* установка IOV, возвращающего данные */
        SETIOV (ctp->iov, buffer + ocb->offset, nbytes);

        /* установка количества байтов (возвращаемых клиентской read()) */
        _IO_SET_READ_NBYTES (ctp, nbytes);

        /*
         * продвижение смещения на число байтов,
         * возвращённых клиенту.
         */

        ocb->offset += nbytes;

        nparts = 1;
    } else {
        /* запрос на 0 байтов или ранее уже всё прочитано */
        _IO_SET_READ_NBYTES (ctp, 0);
        nparts = 0;
    }

    /* время доступа помечается как недействительное
     (мы просто обращались к нему) */
    if (msg->i.nbytes > 0) ocb->attr->flags |= IOFUNC_ATTR_ETIME;
    return (_RESMGR_NPARTS (nparts));
}

```

Блок *ocb* предоставляет нам контекст, сохраняя область *offset*, которая указывает позицию внутри буфера, и указатель на структуру атрибутов *attr*, которая в поле *nbytes* сообщает нам действительный размер буфера. Мы передаём библиотеке менеджера ресурсов адрес функции обработчика *io_read()*, так что он знает, как её вызвать. Вот код функции *main()*, где мы вызываем функцию *iofunc_func_init()*:

```

/* инициализация функций обработки сообщений */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                _RESMGR_IO_NFUNCS, &io_funcs);
io_funcs.read = io_read;

```

Нам также необходимо добавить следующее в область под *main()*:

```

#include <errno.h>

```

```
#include <unistd.h>
```

```
int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb);  
static char *buffer = "Hello world\n";
```

Где будет заполняться поле *nbytes* структуры атрибутов? В функции *main()*, сразу после того, как мы выполнили *iofunc_attr_init*. Мы слегка модифицировали функцию *main()*:

После этой строки:

```
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);
```

мы добавили вот это:

```
attr.nbytes=strlen(buffer)+1;
```

В этой точке, если мы запустим менеджер ресурсов (наш пример менеджера ресурсов использует имя */dev/sample*), мы можем выполнить:

```
# cat /dev/sample
```

и получить

```
Hello, world
```

Возвращаемая строка (*_RESMGR_NPARTS(nparts)*) предписывает библиотеке менеджера ресурсов выполнить следующее:

- ответить клиенту
- ответить с *nparts* векторами (IOV) ввода/вывода.

Откуда берётся массив векторов ввода/вывода? Здесь используется *ctp->iov*. Поэтому вначале мы используем макрос *SETIOV()*, чтобы создать указатель

ctp->iov на данные, которые мы передаём с ответом. Если у нас нет данных, как в случае чтения нуля байтов, то мы выполняем возврат из функции (*_RESMGR_NPARTS(0)*). Но функция *read()* возвращает число успешно прочитанных данных. Как мы передадим ей эту информацию? Для этого и выполнялся макрос *_IO_SET_READ_NBYTES()*. Он берёт *nbytes*, которое мы ему передали, и хранит его отдельно в контекстной структуре (*ctp*). Затем, когда мы выполняем возврат в библиотеку менеджера, библиотека берёт этот *nbytes* и передаёт его как второй параметр функции *MsgReply()*. Второй параметр указывает ядру, что именно должна вернуть функция *MsgSend()*. И поскольку функция *read()* вызывает *MsgSend()*, то там она находит, сколько байтов было прочитано. Мы также обновляем время доступа к этому устройству в обработчике *read*. Более подробно об обновлении времени доступа см. раздел "Обновление времени чтения и записи" ниже.

Пути добавления функциональности менеджеру ресурсов

Вы можете добавить функциональности менеджеру ресурсов следующими основными способами:

- Использовать функции по умолчанию внутри Ваших собственных функций.
- Использовать вспомогательные функции внутри Ваших собственных функций.
- Написать всю функцию самостоятельно.

Первые два пути почти идентичны, поскольку в действительности функции по умолчанию сами по себе ничего и не делают – они полагаются на вспомогательные функции POSIX. Третий подход имеет свои достоинства и недостатки.

Использование функций по умолчанию

Поскольку функции, используемые по умолчанию (например, *iofunc_open_default()*) могут быть установлены непосредственно в таблице переходов, нет смысла встраивать их внутрь Ваших собственных функций.

Вот пример того, как Вы могли бы сделать это с собственным обработчиком *io_open()*:

```
main (int argc, char **argv)
{
    ...

    /* установка всех принимаемых по умолчанию функций */
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                    _RESMGR_IO_NFUNCS, &io_funcs);

    /* перехват функции open */
    connect_funcs.open = io_open;
    ...
}

int
io_open (resmgr_context_t *ctp, io_open_t *msg, RESMGR_HANDLE_T *handle, void *extra)
{
    return (iofunc_open_default (ctp, msg, handle, extra));
}
```

Очевидно, что это просто некий шаг вперёд, позволяющий получить управление в Вашей функции *io_open()*, когда от клиента прибывает сообщение. Вы можете сделать что-нибудь перед тем или после того, как функция по умолчанию выполнит свои действия:

```
/* пример выполнения чего-то перед */
extern int accepting_opens_now;

int
io_open (resmgr_context_t *ctp, io_open_t *msg, RESMGR_HANDLE_T *handle, void *extra)
{
    if (!accepting_opens_now) {
        return (EBUSY);
    }

    /*
     * в этой точке мы обнаруживаем успешность произошедшего открытия,
     * так что позволяем функции умолчанию, делать свою «работу»
     */

    return (iofunc_open_default (ctp, msg, handle, extra));
}
```

ИЛИ

```
/* пример выполнения чего-то после */

int
io_open (resmgr_context_t *ctp, io_open_t *msg, RESMGR_HANDLE_T *handle, void *extra)
{
```

```

int     sts;

/*
 * даём функции по умолчанию, выполнять проверку и работу для нас
 */

sts = iofunc_open_default (ctp, msg, handle, extra);

/*
 * если функция по умолчанию говорит, что всё в порядке
 * и можно позволить открытие, мы хотим зарегистрировать запрос
 */

if (sts == EOK) {
    log_open_request (ctp, msg);
}
return (sts);
}

```

Само собой разумеется, что Вы можете сделать нечто до и после стандартного обработчика POSIX по умолчанию.

Принципиальное достоинство такого подхода – совсем небольшие затраты на добавление функциональности стандартному обработчику POSIX по умолчанию.

Использование вспомогательных функций

Функции по умолчанию используют вспомогательные функции – они не могут размещаться непосредственно в таблице переходов связей или таблице переходов ввода/вывода, но они осуществляют большую часть работы.

Вот исходный код двух функций: *iofunc_chmod_default()* и *iofunc_stat_default()*:

```

int iofunc_chmod_default (resmgr_context_t *ctp, io_chmod_t *msg, iofunc_ocb_t *ocb)
{
    return (iofunc_chmod (ctp, msg, ocb, ocb -&gt; attr));
}

int iofunc_stat_default (resmgr_context_t *ctp, io_stat_t *msg, iofunc_ocb_t *ocb)
{
    iofunc_time_update (ocb -&gt; attr);
    iofunc_stat (ocb -&gt; attr, &msg -&gt; o);
    return (_RESMGR_PTR (ctp, &msg -&gt; o, sizeof (msg -&gt; o)));
}

```

Заметьте, как обработчик *iofunc_chmod()* выполняет всю работу для обработчика по умолчанию *iofunc_chmod_default()*. Это типично для простых функций.

Наиболее интересным случаем является обработчик по умолчанию *iofunc_stat_default()*, который вызывает две вспомогательные подпрограммы. Вначале он вызывает функцию *iofunc_time_update()*, чтобы гарантировать обновление всех полей времени (*atime*, *ctime* и *mtime*). Затем он вызывает функцию *iofunc_stat()*, которая создает ответное сообщение. Наконец, функция по умолчанию создаёт указатель на структуру *ctp* и возвращает его.

Наиболее запутанной является обработка, выполняемая обработчиком *iofunc_open_default()*:

```

int iofunc_open_default (resmgr_context_t *ctp, io_open_t *msg,
                        iofunc_attr_t *attr, void *extra)
{
    int     status;
    iofunc_attr_lock (attr);

    if ((status = iofunc_open (ctp, msg, attr, 0, 0)) != EOK) {

```

```

        iofunc_attr_unlock (attr);
        return (status);
    }

    if ((status = iofunc_ocb_attach (ctp, msg, 0, attr, 0))
        != EOK) {
        iofunc_attr_unlock (attr);
        return (status);
    }

    iofunc_attr_unlock (attr);
    return (EOK);
}

```

Этот обработчик вызывает четыре вспомогательные функции:

1. Он вызывает функцию *iofunc_attr_lock()*, чтобы заблокировать структуру атрибутов, так что обработчик получает единоличный доступ к этой структуре.
2. Затем он вызывает вспомогательную функцию *iofunc_open()*, которая осуществляет проверку существующих прав доступа.
3. Затем он вызывает *iofunc_ocb_attach()*, чтобы связать *ocb* с этим запросом, так что *ocb* будет автоматически передан позже всем функциям ввода/вывода.
4. И, наконец, он вызывает функцию *iofunc_attr_unlock()*, чтобы снять блокировку со структуры атрибутов.

Написание всей функции самостоятельно

Иногда функция по умолчанию не способна помочь Вашему менеджеру ресурсов. Например, функции *iofunc_read_default()* и *iofunc_write_default()* реализуют `/dev/null` – они выполняют работу по возврату 0 байтов (EOF) или "заглатыванию" всех байтов сообщения (соответственно).

Может возникнуть необходимость сделать что-либо в этих обработчиках (если Ваш менеджер ресурсов не поддерживает сообщений `_IO_READ` или `_IO_WRITE`). В этих случаях есть ещё функции, которыми Вы можете пользоваться: *iofunc_read_verify()* и *iofunc_write_verify()*.

Обработка сообщений `_IO_WRITE`

Обработчик *io_write* отвечает за запись байтов данных в устройство после получения от клиента сообщения `_IO_WRITE`. Примерами функций, посылающих это сообщение, являются функции *write()* и *fflush()*. Вот сообщение:

```

struct _io_write {
    uint16_t type;
    uint16_t combine_len;
    int32_t nbytes;
    uint32_t xtype;
    /* unsigned char    data[nbytes]; */
};

typedef union {
    struct _io_write    i;
    /* nbytes возвращаемые с MsgReply */
} io_write_t;

```


Как и в случае *io_read_t*, мы имеем объединение для сообщения ввода и сообщения вывода, с пустым сообщением вывода (число байтов, записанных в действительности, возвращается библиотекой менеджера ресурсов непосредственно в клиентском *MsgSend()*).

Данные, записанные клиентом, почти всегда следуют за заголовком, хранящимся в структуре *_io_write*. Исключением являются случаи, когда запись была выполнена с использованием функций *pwrite()* или *pwrite64()*. Подробнее об этом – когда мы будем обсуждать поле *xtype*.

При получении доступа к данным, мы рекомендуем, чтобы Вы читали их из своего собственного буфера. Вы можете создать буфер (например, под именем *inbuf*), достаточного размера, чтобы хранить все данные, которые Вы ожидаете получить от клиента (если буфер будет недостаточно велик, Вам придётся читать данные по частям).

Пример кода обработчика сообщений **_IO_WRITE**

Ниже приведен фрагмент кода, который может быть добавлен в любой из примеров простых менеджеров ресурсов. Он выводит на печать всё, что ему передаётся (предполагая, что ему передаётся только символьный текст):

```
int io_write (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb)
{
    int    status;
    char   *buf;

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK) return (status);
    if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE) return(ENOSYS);

    /* установка числа байтов (возвращаемых клиентской write()) */
    _IO_SET_WRITE_NBYTES (ctp, msg->i.nbytes);

    buf = (char *) malloc(msg->i.nbytes + 1);
    if (buf == NULL) return(ENOMEM);

    /*
     * Перепрочтение данных из буфера сообщений отправителя.
     * Мы не предполагаем, что все данные в буфере получения
     * библиотеки менеджера ресурсов готовы.
     */

    resmgr_msgread(ctp, buf, msg->i.nbytes, sizeof(msg->i));
    buf[msg->i.nbytes] = '\0'; /* просто на случай, если текст не завершён NULL'ом */
    printf ("Received %d bytes = '%s'\n", msg->i.nbytes, buf);
    free(buf);

    if (msg->i.nbytes > 0)
        ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;

    return (_RESMGR_NPARTS (0));
}
```

Конечно, мы должны передать библиотеке менеджера ресурсов адрес нашего обработчика *io_write*, чтобы она знала, как его вызвать. В текст кода для функции *main()*, там, где мы вызываем функцию *iofunc_func_init()*, мы добавим строку, чтобы зарегистрировать наш обработчик *io_write*:

```
/* инициализация функций для обработки сообщений */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
```

```
        _RESMGR_IO_NFUNCS, &io_funcs);  
io_funcs.write = io_write;
```

Вам может понадобиться добавить следующий прототип:

```
int io_write(resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb);
```

На этом этапе, если Вы запустите менеджер ресурсов (наш простой менеджер ресурсов использует `/dev/sample`), Вы можете передать ему данные, выполнив команду:

```
#echo Hello > /dev/sample  
Received 6 bytes = 'Hello'
```

Заметьте, что мы передали последний аргумент в функции `resmgr_msgread()` (аргумент смещения `offset`) как размер входного буфера сообщений. Это приводит к "перепрыгиванию" через заголовок и получению компоненты данных.

Если буфер, который Вы создали, недостаточен для того, чтобы вместить всё сообщение клиента (например, у Вас буфер в 4К, а клиенту приспичило записать 1 мегабайт), Вы должны читать буфер пошагово, используя цикл и изменяя смещение, переданное функцией `resmgr_msgread()` на число байт, прочитанных каждый раз.

В отличие от примера обработчика `io_read`, на это раз мы ничего не делаем с `ocb->offset`. В этом случае и нет необходимости что-то делать. Поле `ocb->offset` будет иметь больше смысла, если понадобится управлять изменением смещения, например, позицией в файле.

Ответ клиенту более прост, чем в обработчике `io_read`, поскольку вызов `call()` не ожидает в ответ блока данных. Ему только необходимо знать, была ли запись успешной, и если да, сколько байт было записано. Чтобы сообщить, сколько байт записано, мы используем макрос `_IOSET_WRITE_NBYTES()`. Он берёт `nbytes`, которые мы ему передаём, и хранит их отдельно в контекстной структуре (`ctp`). После возврата в библиотеку менеджера ресурсов, библиотека берёт эти `nbytes` и передаёт их как второй параметр в `MsgReply()`. Второй параметр говорит ядру, что именно `MsgSend()` должна будет вернуть. И поскольку функция `write()` вызывает `MsgSend()`, то там она и найдёт, сколько байтов было записано.

Поскольку мы пишем в устройство, мы также должны обновить время модификации и, возможно, время создания. Подробнее об обновлении времени модификации и изменении статуса файла см. в разделе "Обновление времени для чтения и записи".

Методы возвращения и ответа

Вы можете вернуться в библиотеку менеджера ресурсов из Ваших функций-обработчиков различными способами. И при этом есть только одна сложность. Если Вы хотите, чтобы библиотека менеджера ресурсов сделала ответ вместо Вас, то Вы должны прямо указать ей, что это надо сделать, и поместить информацию, которую она будет использовать во всех необходимых местах.

В этом разделе мы обсудим следующие способы возвращения в библиотеку менеджера ресурсов:

- Возврат с ошибкой.
- Возврат с использованием массива векторов ввода/вывода, указывающих на Ваши данные.

- Возврат с одиночным буфером, содержащим данные.
- Успешный возврат, но без данных.
- Вызов библиотеки менеджера ресурсов для исполнения ответа.
- Выполнение ответа в сервере.
- Возврат и указание библиотеке выполнить действия по умолчанию.

Возврат с ошибкой

Чтобы функция, которую вызвал клиент (например, *read()*), вернула ошибку, Вы просто выполняете возврат с соответствующим кодом номера ошибки *errno* (см. `<errno.h>`), например:

```
return(ENOMEM);
```

Иногда Вы можете встретить другую форму использования (устаревшую и не рекомендуемую), которая работает точно так же:

```
return(_RESMGR_ERRNO(ENOMEM));
```

В обоих приведенных случаях функция *read()* вернёт `-1` и *errno*, установленную в `ENOMEM`.

Возврат с использованием массива векторов ввода/вывода, указывающих на Ваши данные

Иногда может понадобиться создать «составной» ответ, состоящий из заголовка, за которым следует один из `N` буферов, и при каждом ответе используются различные буферы. Для этого Вы можете установить массив векторов ввода/вывода, элементы которого указывают на заголовок и на буфер.

Контекстная структура уже имеет массив векторов ввода/вывода. Если Вы хотите, чтобы библиотека менеджера ресурсов выполнила ответ за Вас, то Вы должны использовать этот массив. Но массив должен содержать достаточное количество элементов. Чтобы застраховать себя на этот случай, необходимо задать значение поля *nparts_max* структуры *resmgr_attr_t*, которую Вы передаёте функции *resmgr_attach* при регистрации имени в пространстве имён.

В следующем примере предполагается, что переменная *i* это значение индекса буфера, предназначенного для ответа в массиве буферов. Значение `2` в вызове `RESMGR_NPARTS(2)` указывает библиотеке, с каким количеством элементов в *ctp->iov* выполнить ответ.

```
my_header_t    header;
a_buffer_t    buffers[N];

...

SETIOV(&ctp->iov[0], &header,    sizeof(header));
```

```
SETIOV(&ctp->iov[1], &buffers[i],      sizeof(buffers[i]));  
return (_RESMGR_NPARTS(2));
```

Возврат с одиночным буфером, содержащим данные

Вот пример ответа на *read()*, в котором все данные возвращаются в одном буфере. Это можно реализовать двумя способами:

```
return(_RESMGR_PTR(ctp, buffer, nbytes));
```

и

```
SETIOV(ctp->iov, buffer, nbytes);  
return(_RESMGR_NPARTS(1));
```

Первый метод, использующий макрос `_RESMGR_PTR()`, является более удобной разновидностью второго метода, в котором возвращается один вектор ввода/вывода.

Успешный возврат, но без данных

Это можно реализовать несколькими способами. Наиболее просто это сделать так:

```
return(EOK);
```

Но мы чаще видим такой способ:

```
return(_RESMGR_NPARTS(0));
```

Заметьте, что ни в одном из рассматриваемых случаев функция *MsgSend()* не возвращает ноль. Значение, возвращаемое *MsgSend()* – это значение, переданное `_IO_SET_READ_NBYTES()`, `_IO_SET_WRITE_NBYTES()`, или другими подобными макросами. Эти макросы использованы в приведенных выше примерах чтения и записи.

Вызов библиотеки менеджера ресурсов для исполнения ответа

В этом случае Вы готовите данные для клиента и вызываете библиотеку менеджера ресурсов, чтобы она ответила за Вас. Однако данные на момент ответа уже не обязаны быть верными. Например, если данные находились в буфере, который Вы захотите освободить перед возвратом, Вы можете использовать следующее:

```
resmgr_msgwrite (ctp, buffer, nbytes, 0);  
free(buffer);  
return(EOK);
```

Функция *resmgr_msgwrite()* немедленно копирует содержание буфера данных в буфер ответа клиенту. Заметим, что ответ, разблокирующий клиента ещё не выполнен, так что есть возможность проверить эти данные. Далее мы освобождаем буфер. Наконец, мы возвращаемся

в библиотеку менеджера ресурсов, и она выполняет ответ с данными нулевой длины. Поскольку ответ имеет нулевую длину, он не переписывает данные, уже записанные в буфер ответа клиенту. Когда клиент возвращается из вызова *send*, данные уже ожидают его.

Выполнение ответа в сервере

Во всех предыдущих примерах используется функция библиотеки менеджера ресурсов, которая вызывает *MsgReply**() или *MsgError*(), чтобы разблокировать клиента. В некоторых случаях Вы не захотите, чтобы библиотека отвечала за Вас. Например, Вы можете выполнить ответ сами, или намереваетесь ответить позднее. В любом случае, Вы должны вернуть следующее:

```
return(_RESMGR_NOREPLY);
```

Оставить клиента заблокированным, ответив позднее

Примером менеджера ресурсов, который отвечает клиенту позднее, является менеджер ресурсов каналов (*pipe*). Если клиент выполняет чтение из канала, но у Вас ещё нет данных для клиента, то у Вас есть выбор. Вы можете ответить, вернув код ошибки (EAGAIN). Либо оставить клиента заблокированным и позднее, когда будет выполнен вызов функции обработки записи, Вы можете ответить клиенту с новыми данными.

Другим примером может быть случай, когда клиент пытается произвести запись на некое устройство, но не хочет получать ответ до тех пор, пока данные не будут полностью записаны. Вот последовательность событий, которые могут произойти:

1. Менеджер ресурсов выполняет какой-то ввод/вывод в устройство, сообщая ему, что данные доступны.
2. Устройство генерирует прерывание, когда оно готово обработать данные.
3. Вы обрабатываете прерывания, записывая данные в устройство.
4. До того, как все данные будут записаны, может произойти несколько прерываний, и только затем Вы ответите клиенту.

Возникает вопрос – а хочет ли клиент оставаться заблокированным все это время? Если клиент не желает оставаться заблокированным, то он открывает устройство с флагом O_NONBLOCK:

```
fd=open("dev/sample", O_RDWR | O_NONBLOCK);
```

По умолчанию Вам позволено блокировать клиента. Одним из первых действий, выполненных в примерах, приведенных выше, был вызов функций верификации POSIX *iofunc_read_verify*() и *iofunc_write_verify*(). Если мы передали адрес последнего параметра *int*, то при возврате функция вернёт этот параметр равным нулю, если клиент не желает быть заблокированным (флаг O_NONBLOCK был установлен) или ненулевым, если клиент желает остаться заблокированным:

```
int    nonblock;

if ((status = iofunc_read_verify (ctp, msg, ocb, &nonblock)) != EOK)
    return (status);

...

int    nonblock;
```

```
if ((status = iofunc_write_verify (ctp, msg, ocb, &nonblock)) != EOK)
    return (status);
```

Когда пришло время решить, отвечать ли Вам с кодом ошибки или ответить позже, Вы делаете следующее:

```
if (nonblock) {
    /* клиент не хочет быть заблокированным */
    return (EAGAIN);
} else {
    /*
     * клиент желает оставаться заблокированным
     * сохранение отдельно по крайней мере ctp->rcvid так чтобы Вы могли
     * ответить ему позже
     */
    ...
    return (_RESMGR_NOREPLY);
}
```

Остаётся вопрос: как ответить самостоятельно? Мы знаем, что *rcvid* для ответа – это *ctp->rcvid*. Если Вы отвечаете позже, то сохраните *ctp->rcvid* отдельно и используйте сохранённое значение в ответе.

```
MsgReply(saved_rcvid, 0, buffer, nbytes);
```

или

```
iov_t    iov[2];

SETIOV(&iov[0], &header, sizeof(header));
SETIOV(&iov[1], &buffers[i], sizeof(buffers[i]));
MsgReplyv(saved_rcvid, 0, iov, 2);
```

Заметим, что Вы можете заполнить буфер ответа клиенту, когда данные становятся доступными, используя *resmgr_msgwrite()* и *resmgr_msgwritev()*. Просто не забудьте потом выполнить *MsgReply*()*, чтобы разблокировать клиента.

Возврат и указание библиотеке выполнить действие по умолчанию

В большинстве случаев для библиотеки менеджера ресурсов действием по умолчанию, является отказ от выполнения функции клиента с кодом возврата ENOSYS:

```
return(_RESMGR_DEFAULT);
```

Другие тонкости обработки ввода/вывода

Предметом обсуждения этого раздела является:

- Обработка поля *xtype*.
- Обработка вызовов *pread*()* и *pwrite()*.
- Обработка вызова *readcond()*.

Обработка поля *xtype*

В составе структур сообщений *io_read* и *io_write* имеется поле *xtype*. В структуре *_io_read*:

```
struct _io_read {
    .....
    uint32_t  xtype;
    .....
}
```

Обычно *xtype* содержит расширенную информацию для определения поведения стандартной функции ввода/вывода. Большинство менеджеров ресурсов обрабатывают лишь несколько значений:

_IO_XTYPE_NONE

Информация расширенного типа не обеспечивается.

_IO_XTYPE_OFFSET

Если клиенты вызывают *pread()*, *pread64()*, *pwrite()* или *pwrite64()*, то они не используют смещение в *ocb*. Вместо этого они предоставляют одноразовое смещение. Это смещение следует за заголовками структур *_io_read* или *_io_write*, которые находятся в начале буферов сообщений.

Например:

```
struct myread_offset
{
    struct _io_read      read;
    struct _xtype_offset offset;
}
```

Некоторые менеджеры ресурсов могут быть уверены, что их клиенты никогда не вызовут *pread*()* или *pwrite*()* (например, менеджер ресурсов, управляющий манипулятором робота, вероятно, не заботился бы об этом). В этом случае Вы можете его игнорировать.

_IO_XTYPE_READCOND

Если некий клиент вызывает функцию *readcond()*, он хочет установить синхронизацию по времени (таймаут) и ограничить размер буфера, выделенного на чтение. Эти ограничения следуют за заголовками структур *_io_read* или *_io_write* в начале буферов сообщений. Например:

```
struct myreadcond
{
    struct _io_read      read;
    struct _xtype_readcond cond;
}
```

как и для _IO_XTYPE_OFFSET, если Ваш менеджер ресурсов не готов обрабатывать *readcond()*, Вы можете его игнорировать.

Если Вы не ожидаете расширенного типа (*xtype*)

Нижеследующий пример показывает, как обработать случай, когда Вы не ожидаете каких-либо расширенных типов. В этом случае, если Вы получаете сообщение, содержащее *xtype*,

отправьте ответ с кодом ENOSYS. Этот пример может быть использован в обработчике либо *io_read*, либо *io_write*.

```
int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb) {
    int    status;

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) {
        return (_RESMGR_ERRNO(status));
    }

    /* Тип xtype не обслуживается */
    if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE)
        return (_RESMGR_ERRNO(ENOSYS));

    ...
}
```

Обработка *pread*()* и *pwrite*()*

Ниже приведены примеры кода, демонстрирующего, как обрабатывать сообщения *_IO_READ* или *_IO_WRITE*, когда клиент вызывает:

- функцию *pread*()*
- функцию *pwrite*()*.

Пример кода, обрабатывающего сообщения *_IO_READ* в функциях *pread*()*

Следующий пример демонстрирует, как обрабатывать сообщения *_IO_READ* в случае, когда клиент вызывает одну из функций *pread*()*.

```
/* здесь мы определяем io_pread_t чтобы сделать проще последующий код */
typedef struct {
    struct _io_read    read;
    struct _xtype_offset offset;
} io_pread_t;

int io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
    off64_t offset; /* откуда читать */
    int    status;

    if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) {
        return(_RESMGR_ERRNO(status));
    }

    switch(msg->i.xtype & _IO_XTYPE_MASK) {
    case _IO_XTYPE_NONE:
        offset = ocb->offset;
        break;
    case _IO_XTYPE_OFFSET:
        /*
         * io_pread_t определена выше
         * клиент выполняет одноразовое чтение с этим смещением offset
         * при вызове одной из функций pread*()
         */
        offset = ((io_pread_t *) msg)->offset.offset;
        break;
    default:
        return(_RESMGR_ERRNO(ENOSYS));
    }
}
```



```

    }
    ...
}

```

Пример кода, обрабатывающего сообщения IO_WRITE() в функциях pwrite*()

Следующий пример демонстрирует, как обрабатывать сообщения IO_WRITE если клиент вызывает одну из функций *pwrite*()*. Имейте в виду, что в буфере посылаемого сообщения информация по структуре *_xtype_offset* следует за структурой *_io_write*. Это подразумевает, что данные будут записываться после информации по структуре *_xtype_offset* (в отличие от обычного случая, когда она следует за данными структуры *_io_write*). Это нужно учесть, когда делается вызов *resmgr_msgreadv()*, чтобы получить данные из буфера посланного сообщения.

```

/* здесь мы определяем io_pwrite_t чтобы сделать проще последующий код */
typedef struct {
    struct _io_write      write;
    struct _xtype_offset  offset;
} io_pwrite_t;

int io_write (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb) {
    off64_t  offset;      /* где писать */
    int      status;
    size_t   skip;        /* смещение в сообщении msg где находятся данные */

    if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK) {
        return(_RESMGR_ERRNO(status));
    }

    switch(msg->i.xtype & _IO_XTYPE_MASK) {
    case _IO_XTYPE_NONE:
        offset = ocb->offset;
        skip = sizeof(io_write_t);
        break;
    case _IO_XTYPE_OFFSET:
        /*
         * io_pwrite_t определена выше
         * клиент выполняет однократную запись с этим смещением offset
         * при вызове одной из функций pwrite*()
         */
        offset = ((io_pwrite_t *) msg)->offset.offset;
        skip = sizeof(io_pwrite_t);
        break;
    default:
        return(_RESMGR_ERRNO(ENOSYS));
    }

    ...

    /*
     * получение данных из буфера посланного сообщения,
     * с пропуском всеё возможной заголовочной информации
     */
    resmgr_msgreadv(ctp, iovs, niovs, skip);

    ...
}

```

Обработка readcond()

Тот же способ действий, который был применён при обработке в случае *pread()/_IO_XTYPE_OFFSET*, может быть использован и при обработке вызовов клиентом функции *readcond()*:

```
typedef struct {
    struct _io_read    read;
    struct _xtype_readcond cond;
} io_readcond_t
```

затем:

```
struct _xtype_readcond *cond
...
CASE _IO_XTYPE_READCOND:
    cond = ((io_readcond_t *)msg)->cond;
    break;
}
```

И затем Ваш менеджер надлежащим образом интерпретирует и обрабатывает аргументы функции *readcond()* (как указано в описании функции *readcond()* в книге "Справочник библиотечных функций").

Обработка атрибутов

Этот раздел содержит:

- Обновление времени при чтении и записи.

Обновление времени при чтении и записи

В примере чтения, приведенном выше мы делаем:

```
if (msg->i.nbytes > 0)
    ocb->attr->flags |= IOFUNC_ATTR_ETIME;
```

В соответствии с требованиями POSIX, если чтение выполнено успешно и клиент запросил более нуля байтов, то время доступа должно быть обновлено. Но POSIX не требует, чтобы оно было обновлено немедленно. Если Вы выполняете несколько операций чтения, может быть не выгодным получать время, обращаясь к ядру при каждом чтении. В приведенном выше примере мы только помечаем, что время нуждается в обновлении. Когда будет обрабатываться следующее сообщение *_IO_STAT* или *_IO_CLOSE_OCB*, библиотека менеджера ресурсов увидит, что время нуждается в обновлении, и тогда получит его из ядра. Конечно, это время – на самом деле не совсем то время, когда было проведено чтение.

Подобно примеру, приведенному выше, мы делаем:

```
if (msg->i.nbytes > 0)
    ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;
```

так что произойдёт то же самое.

Если Вы захотите, чтобы время действительно представляло время чтения или записи, то после установки флагов Вам необходимо вызвать вспомогательную функцию `iofunc_time_update()`. Тогда чтение будет выглядеть так:

```
if (msg->i.nbytes > 0) {
    ocb->attr->flags |= IOFUNC_ATTR_ETIME;
    iofunc_time_update(ocb->attr);
}
```

а запись так:

```
if (msg->i.nbytes > 0) {
    ocb->attr->flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;
    iofunc_time_update(ocb->attr);
}
```

Функция `iofunc_time_update()` должна быть вызвана до того, как Вы сбросите какие-либо кэшированные атрибуты. В результате поля со значениями времени будут обновлены. В структуре атрибутов в поле флагов будет установлен бит `IOFUNC_ATTR_DIRTY_TIME`, указывая, что это поле должно быть обновлено, когда атрибуты будут сбрасываться из кэша.

Комбинированные сообщения

В этом разделе:

- Когда используются комбинированные сообщения.
- Обработка комбинированных сообщений библиотечными функциями.

Когда используются комбинированные сообщения

Для того чтобы сохранить пропускную способность сети и обеспечить поддержку атомарных операций, в QNX6 поддерживаются комбинированные сообщения. Комбинированные сообщения конструируются библиотекой клиента и состоят из нескольких сообщений ввода/вывода и/или сообщений связи, упакованных в одно сообщение. Давайте посмотрим, как это используется.

Атомарные операции

Рассмотрим случай, когда два потока исполняют следующий код, пытаясь одновременно читать из одного файлового дескриптора:

```
a_thread () {
    char buf [BUFSIZ];

    lseek (fd, position, SEEK_SET);
    read (fd, buf, BUFSIZ);
    ...
}
```

Первый поток выполняет *lseek()* и затем вытесняется вторым потоком. Когда первый поток возобновляет исполнение, смещение в файле для него будет в конце той области, из которой закончил чтение второй поток, а совсем не в той позиции, где был его указатель по *lseek()*.

Это может быть решено одним из трёх способов:

- два потока могут использовать мутекс (взаимоисключающий семафор), чтобы гарантировать одновременное использование файлового дескриптора только одним потоком
- каждый поток может открывать файл лично для себя, что генерирует уникальный файловый дескриптор, не влияющий на другие потоки
- потоки могут использовать функцию *readblock()*, которая выполняет атомарные *lseek()* и *read()*.

Давайте рассмотрим эти три метода.

Использование мутекса

При первом решении, если два потока используют мутекс, возникают следующие проблемы: каждая операция *read()*, *lseek()* или *write()* **должна** использовать мутекс.

Если не следовать этой практике, то проблема останется нерешенной. Например, предположим, что один поток, выполняя соглашение о блокировании мутекса и выполняя *lseek()*, думает, что он защищен. Однако другой поток (не выполняющий соглашения), может вытеснить первый поток и передвинуть смещение куда-то в другое место. Когда первый поток возобновит свою работу, мы опять столкнемся с проблемой, когда смещение находится в другом (непредсказуемом) месте. В целом, использование мутекса будет успешным только в очень тщательно управляемых проектах, когда только проверка кода гарантирует, что все и всякие файловые функции потоков соблюдают соглашение о блокировании мутексов.

Каждому потоку – по файлу!

Второе решение – использование различных файловых дескрипторов – является хорошим универсальным решением, но только до тех пор, пока Вам явно не потребуется, чтобы файловый дескриптор обеспечивал совместный доступ.

Функция readblock()

Для того, чтобы функция *readblock()* была в состоянии действовать на атомарные операции *seek/read*, необходимо гарантировать, что все запросы, которые она посылает менеджеру ресурсов, будут исполнены одновременно. Это достигается комбинированием сообщений *_IO_LSEEK* и *_IO_READ* в одном сообщении. Таким образом, когда базовый уровень выполняет *MsgReceive()*, эта функция получит запрос *readblock()* в одном атомарном сообщении.

Рассуждение о пропускной способности

Другим случаем, когда полезны комбинированные сообщения, является функция *stat()*, которая может быть реализована через функции *open()*, *fstat()* и *close()*.

Вместо того, чтобы генерировать три отдельных сообщения (по одному на каждую функцию), библиотека комбинирует их в одно непрерывное сообщение. Это даёт выигрыш в производительности, особенно при обмене по сети, а также упрощает код менеджера ресурсов, поскольку не вынуждает иметь функцию связи для обработки *stat()*.

Обработка комбинированных сообщений библиотекой

Библиотека менеджера ресурсов обрабатывает комбинированные сообщения, предоставляя каждой компоненте сообщения соответствующие подпрограммы-обработчики. Например, если мы получим комбинированное сообщение, содержащее в себе `_IO_LSEEK` и `_IO_READ` (например, `readblock()`), библиотека вызовет для нас по очереди наши функции `io_lseek()` и `io_read()`.

Но давайте посмотрим, что происходит в менеджере ресурсов, когда он обрабатывает эти сообщения. В случае многопоточности одновременно оба потока клиента могут послать свои "атомарные" комбинированные сообщения. Два потока в менеджере ресурсов будут теперь пытаться обслужить эти два сообщения. Мы снова пришли к той же проблеме синхронизации, которую мы первоначально имели на клиентской стороне – один поток может пройти часть пути по обработке сообщения и затем вытесниться другим потоком.

Решение? Библиотека менеджера ресурсов предоставляет вызовы для блокирования `ocb`, пока обрабатывается какое-либо сообщение (исключая `_IO_CLOSE` и `_IO_UNBLOCK` – мы еще к этому вернёмся). Например, когда обрабатывается комбинированное сообщение `readblock()`, библиотека менеджера ресурсов выполняет вызовы в таком порядке:

1. Обработчик `_IO_LOCK_OCB`.
2. Обработчик сообщения `_IO_LSEEK`.
3. Обработчик сообщения `_IO_READ`.
4. Обработчик сообщения `_IO_UNLOCK_OCB`.

Следовательно, в нашем сценарии два потока внутри менеджера ресурсов будут взаимно исключать друг друга путём блокирования – первый поток, захвативший блокировку, полностью обработает комбинированное сообщение, снимет блокировку, и затем второй поток будет выполнять свою работу.

Давайте рассмотрим несколько вопросов, связанных с обработкой комбинированных сообщений:

Составные ответы

- Доступ к компонентам данных.
- Блокирование и разблокирование структуры атрибутов.
- Различные стили сообщений связи.
- `_IO_CONNECT_COMBINE_CLOSE`.
- `_IO_CONNECT_COMBINE`.

Составные ответы

Как мы видим, комбинированное сообщение в действительности состоит из нескольких "нормальных" сообщений менеджера ресурсов, собранных в одно большое непрерывное сообщение. Библиотека менеджера ресурсов обрабатывает каждый компонент комбинированного сообщения отдельно, путём выделения отдельных компонент и затем внешнего вызова обработчиков, которые мы определили в таблице функций связи и функций ввода/вывода для каждого компонента соответственно.

Это обычно не вызывает каких-либо новых затруднений для самих обработчиков сообщений, за исключением одного случая. Рассмотрим комбинированное сообщение `readblock()`:

Клиент вызывает:

```
readblock()
```

Сообщение(я):

`_IO_LSEEK, _IO_READ`

Внешние вызовы:

```
io_lock_ocb()  
io_lseek()  
io_read()  
io_unlock_ocb()
```

Обычно после обработки сообщения `_IO_LSEEK`, Ваш обработчик возвращает текущую позицию внутри файла. Однако следующее сообщение (`_IO_READ`) также возвращает данные. По соглашению принимается, что только последнее сообщение (из содержащихся внутри комбинированного сообщения), в действительности вернёт данные. Промежуточным сообщениям позволено возвращать только признак «действие прошло/не прошло».

В результате обработчик сообщения `_IO_LSEEK` должен знать, был ли он вызван (или нет), как часть обработки комбинированного сообщения. Если был, то он вернёт либо только ЕОК (указывая, что функция `lseek()` отработала успешно), либо определенный код ошибки.

Но если обработчик `_IO_LSEEK` был вызван не как часть обработки комбинированного сообщения, он вернёт ЕОК **и** новое значение смещения (или, в случае ошибки, только код ошибки). Вот пример кода по умолчанию для обработчика `lseek()` уровня `iofunk`:

```
int iofunc_lseek_default (resmgr_context_t *ctp, io_lseek_t *msg, iofunc_ocb_t *ocb)  
{  
    /*  
     * исполнение здесь обработки может дать "раннее обнаружение" состояния ошибки  
     */  
    ...  
  
    /* касательно решения: комбинированные сообщения выполняются здесь */  
    if (msg -> i.combine_len & _IO_COMBINE_FLAG) {  
        return (ЕОК);  
    }  
  
    msg -> o = offset;  
    return (_RESMGR_PTR (ctp, &msg -> o, sizeof (msg -> o)));  
}
```

В этом операторе было принято важное решение:

```
if (msg-> i.combine_len & _IO_COMBINE_FLAG)
```

Если в для поля `combine_len` установлен бит `_IO_COMBINE_FLAG`, это указывает на то, что сообщение обрабатывается как часть комбинированного сообщения.

Когда библиотека менеджера ресурсов обрабатывает отдельные компоненты комбинированного сообщения, она проверяет код ошибки, возвращаемый обработчиками отдельных сообщений. Если обработчик возвращает что-то вместо ЕОК, обработка последующих компонентов комбинированного сообщения не выполняется. Клиенту возвращается код ошибки, который был возвращён обработчиком соответствующей компоненты.

Доступ к компонентам данных

Второй проблемой, связанной с обработкой комбинированных сообщений, является получение доступа к области данных для последовательных компонент сообщений.

Например, в комбинированном сообщении *writeblock()* первым идет сообщение *lseek()*, за ним - сообщение *write()*. Это означает, что данные, связанные с запросом *write()*, находятся в буфере полученного сообщения дальше, чем это было бы в случае простого сообщения *_IO_WRITE*:
Клиент вызывает:

```
writeblock()
```

Сообщение(я):

```
_IO_LSEEK, _IO_WRITE, данные
```

Внешние вызовы:

```
io_lock_ocb()  
io_lseek()  
io_write()  
io_unlock_ocb()
```

Эта проблема решается легко. Имеется функция библиотеки менеджера ресурсов *resmgr_msgread()*, которая знает, как получить данные, соответствующие корректным компонентам сообщения. Таким образом, если в обработчике *io_write* Вы использовали функцию *resmgr_msgread()* вместо *MsgRead()*, для Вас это будет прозрачным.

i Менеджеры ресурсов всегда должны использовать "обволакивающие" функции *resmgr_msg*()*.

Для справки, вот исходный код функции *resmgr_msg*()*:

```
int resmgr_msgreadv (resmgr_context_t *ctp,  
                    struct iovec *rmsg,  
                    int rparts,  
                    int offset)  
{  
    return (MsgReadv (ctp -> rvid, rmsg, rparts, ctp -> offset + offset));  
}
```

Как вы можете видеть, функция *resmgr_msgread()* просто вызывает функцию *MsgRead()* со смещением на компоненту сообщения от начала буфера комбинированного сообщения. Для полноты картины отметим, что имеется также функция *resmgr_msgwrite()*, которая работает точно так же, как функция *MsgWrite()*, за исключением того, что она разыменовывает ссылку переданного *ctp*, чтобы получить *rvid*.

Блокирование и разблокирование структуры атрибутов

Как обсуждалось выше, другой гранью действий функции *readblock()* с точки зрения клиента является то, что она атомарная. Для того, чтобы атомарно обработать запросы для конкретного *ocb*, мы должны заблокировать и разблокировать структуру атрибутов, на которую указывает *ocb*, гарантируя, что только один поток менеджера ресурсов будет иметь доступ к *ocb* одновременно.

Библиотека менеджера ресурсов предоставляет для этого два внешних вызова:

- *lock_ocb*
- *unlock_ocb*

Это поля структуры функций ввода/вывода. Обработчики, которые Вы снабдили этими вызовами, будут блокировать и разблокировать структуру атрибутов, на которую указывает *ocb*, путём вызова *iofunk_attr_lock()* и *iofunk_attr_unlock()*. Таким образом, если Вы блокируете структуру атрибутов, есть надежда, что внешний вызов *lock_ocb* будет на определённое время заблокирован. Это является нормальным и ожидаемым поведением. Заметим также, что структура атрибутов автоматически блокируется, когда вызываются Ваши функции ввода/вывода.

Типы сообщений связи

Давайте рассмотрим основной случай для обработчика *io_open* – он не всегда соответствует вызову клиентом функции *open()*! Например, пусть клиент вызвал функции *stat()* и *access()*.

_IO_CONNECT_COMBINE_CLOSE

Для вызова клиентом функции *stat()*, мы фактически выполняем последовательность *open()/fstat()/close()*. Заметьте, если бы мы в действительности делали это, потребовалось бы три сообщения. Из соображений повышения производительности мы выполняем функцию *stat()* как одно комбинированное сообщение:

Клиент вызывает:

```
stat()
```

Сообщение(я):

```
_IO_CONNECT_COMBINE_CLOSE, _IO_STAT
```

Внешние вызовы:

```
io_open()  
io_lock_ocb()  
io_stat()  
io_unlock_ocb()  
io_close()
```

Сообщение *_IO_CONNECT_COMBINE_CLOSE* вызывает обработчик *io_open*. Затем оно неявно (в конце обработки комбинированного сообщения) вызывает обработчик *io_close_ocb*.

_IO_CONNECT_COMBINE

Для вызова функции *access()* библиотека клиента создаст соединение с менеджером ресурсов и выполнит вызов функции *stat()*. Затем, на основе результатов вызова *stat()*, функция *access()* библиотеки клиента, может быть, вызовет функцию *devctl()*, чтобы получить больше информации. В любом случае, поскольку функция *access()* открыла устройство, она должна также вызвать функцию *close()*, чтобы его закрыть:

Клиент вызывает:

```
access()
```

Сообщение(я):

```
_IO_CONNECT_COMBINE, _IO_STAT  
_IO_DEVCTL (необязательно)  
_IO_CLOSE
```


Внешние вызовы:

```
io_open()
io_lock_ocb()
io_stat()
io_unlock_ocb() }
io_lock_ocb()   }  необязательно
io_devctl()    }
io_unlock_ocb()
io_close()
```

Заметьте, как функция *access()* получила доступ к устройству (имени) – она передала ему вместе с сообщением `_IO_STAT` сообщение `_IO_CONNECT_COMBINE`. Это действие создало *ocb* (когда был вызван обработчик *io_open*), заблокировало соответствующую структуру атрибутов (через функцию *io_lock_ocb()*), исполнило *stat()* (*io_stat()*) и затем разблокировало структуру атрибутов (*io_unlock_ocb()*). Заметьте, что мы не закрывали неявно *ocb* – он остался для более позднего, явного сообщения. Сравните эту обработку с "чистой" функцией *stat()*, описанной выше.

Расширенные структуры управления данными (DCS)

Этот раздел содержит:

- Расширение структур атрибутов и *ocb*.
- Расширение структур точек монтирования.

Расширение структур атрибутов и *ocb*

В нашем примере `/dev/sample` мы создали статический буфер, связанный с ресурсом в целом. Иногда может понадобиться хранить указатель на буфер, связанный с ресурсом, а не глобальную область памяти. Чтобы поддерживать указатель для ресурса, мы должны сохранить его в структуре атрибутов. Но структура атрибутов не имеет резервных полей, и мы должны расширить её, чтобы было, где хранить этот указатель.

Иногда Вам может понадобиться добавить дополнительные входы стандартных функций *iofunk_**(*)* в *ocb* (*iofunk_ocb_t*).

Давайте посмотрим, как мы можем расширить обе эти структуры. Базовая стратегия - включить существующие атрибуты и структуры стандартного *ocb* внутрь новой структуры, которая содержит наши расширения. Вот пример:

```
/* Определение наших перекрытий перед включением <sys/iofunc.h> */
struct device;
#define IOFUNC_ATTR_T      struct device      /* см. прим. 1 */
struct ocb;
#define IOFUNC_OCB_T      struct ocb         /* см. прим. 1 */

#include <sys/iofunc.h>
#include <sys/dispatch.h>

struct ocb {                                /* см. прим. 2 */
```

```

    iofunc_ocb_t    hdr;                /* см. прим. 4; всегда должно быть первым */
    struct ocb     *next;
    struct ocb     **prev;             /* см. прим. 3 */
};

struct device {
    iofunc_attr_t  attr;               /* см. прим. 2 */
    struct ocb     *list;             /* всегда должно быть первым */
    /* ожидая записи */
};

/* прототипы, необходимые, поскольку мы ссылаемся на них несколькими строками ниже */
struct ocb *ocb_alloc (resmgr_context_t *ctp, struct device *device);
void ocb_free (struct ocb *ocb);

iofunc_funcs_t ocb_funcs = { /* наше размещение ocb и функции очистки */
    _IOFUNC_NFUNCS,
    ocb_alloc,
    ocb_free
};

/* структура подмонтирования, она у нас одна и поэтому мы статически её декларируем */
iofunc_mount_t  mountpoint = { 0, 0, 0, 0, &ocb_funcs };

/* одна структура device для каждого присоединённого имени,
 * в этом примере только одно имя */
struct device   deviceattr;

main()
{
    ...

    /*
     * deviceattr будет косвенно содержать адреса функций размещения ocb
     * и функций очистки
     */

    deviceattr.attr.mount = &mountpoint;
    resmgr_attach (... , &deviceattr);

    ...
}

/*
 * ocb_alloc
 *
 * Цель этого - дать нам место для размещения нашего собственного ocb.
 * Это вызывается как результат произошедшего открытия
 * (напр. это вызывается функцией iofunc_open_default).
 * Мы регистрируем это посредством структуры монтирования
 */
IOFUNC_OCB_T
ocb_alloc (resmgr_context_t *ctp, IOFUNC_ATTR_T *device) {
    struct ocb *ocb;

    if (!(ocb = calloc (1, sizeof (*ocb)))) {
        return 0;
    }

    /* см. прим. 3 */
    ocb -> prev = &device -> list;
    if (ocb -> next = device -> list) {
        device -> list -> prev = &ocb -> next;
    }
    device -> list = ocb;

    return (ocb);
}

```

```

/*
 * ocb_free
 *
 * Цель этого - дать нам место для освобождения нашего ocb.
 * Это вызывается как результат выполненного закрытия
 * (напр. это вызывается функцией iofunc_close_ocr_default.
 * Мы регистрируем это посредством структуры монтирования
 */
void
ocr_free (IOFUNC_OCB_T *ocr)
{
    /* см. прим. 3 */
    if (*ocr -> prev = ocr -> next) {
        ocr -> next -> prev = ocr -> prev;
    }
    free (ocr);
}

```

Примечания:

1. Мы разместили определения для наших расширенных структур *перед* включением заголовочного файла стандартных функций ввода/вывода. Поскольку заголовочный файл стандартных функций ввода/вывода проверяет, определены уже или нет две декларирующие константы, это открывает удобный путь семантически перекрыть структуры.
2. Определяет новые расширенные структуры данных, обеспечивая, что включенные в нее поля будут размещены первыми.
3. Примеры функций *ocr_malloc()* и *ocr_free()* показывают, каким образом вновь размещенные *ocr* будут храниться в связанном списке. Обратите внимание на использование поля ***prev* структуры *ocr* – это указатель на указатель на *ocr*.
4. Вы *всегда должны* размещать структуру *iofunc*, которую Вы перекрыли, как первый член новой расширенной структуры. Это позволяет общей библиотеке правильно работать в случаях обработки по умолчанию.

Расширение структуры точки монтирования

Структура *iofunc_mount_t* может быть расширена таким же образом, как и структуры атрибутов и *ocr*. В этом случае мы определяем:

```
#define IOFUNC_MOUNT_T          struct newmount
```

затем объявляется новая структура:

```

struct newmount {
    iofunc_mount_t      mount;
    int                 ourflag;
};

```

Обработка сообщений devctl()

Функция *devctl()* представляет механизм общего назначения для связей с менеджером ресурсов. Клиенты могут посылать данные, получать, или и посылать и получать данные от менеджера ресурсов. Формат клиентской функции *devctl()*:

```
devctl(      int    fd,
            int    dcmd,
            void   * data,
            size_t nbytes,
            int    * return_info);
```

Следующие значения (детально описаны в документации по функции *devctl()* в "Справочнике библиотечных функций") отображаются непосредственно на сообщение `_IO_DEVCTL`:

```
struct _io_devctl {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       dcmd;
    int32_t       nbytes;
    int32_t       zero;
    /* char       data[nbytes]; */
};

struct _io_devctl_reply {
    uint32_t      zero;
    int32_t       ret_val;
    int32_t       nbytes;
    int32_t       zero2;
    /* char       data[nbytes]; */
} ;

typedef union {
    struct _io_devctl      i;
    struct _io_devctl_reply o;
} io_devctl_t;
```

Как и в большинстве сообщений менеджеру ресурсов, мы определяем объединение, которое содержит структуру ввода (приходящую в менеджер ресурсов) и структуру ответа или вывода (идущую обратно клиенту). Обработчик сообщений *io_devctl* менеджера ресурсов имеет прототип с аргументом:

```
io_devctl_t * msg
```

который является указателем на объединение, содержащее сообщение.

Поле *type* имеет значение `_IO_DEVCTL`.

Область *combine_len* имеет смысл для комбинированных сообщений, см. раздел "Комбинированные сообщения" в этой главе.

Значение *nbytes* – это число байт, переданных функции *devctl()*. Значение содержит размер данных, посылаемых драйверу устройства, или максимальный размер данных, получаемых от драйвера устройства.

Наиболее интересным полем структуры ввода является *dcmd*. Оно передаётся функции *devctl()*. Эта команда сформирована с использованием макроса, определённого в `<devctl.h>`:

```
#define _POSIX_DEVDIR_NONE      0
#define _POSIX_DEVDIR_TO       0x80000000
#define _POSIX_DEVDIR_FROM     0x40000000
#define __DIOF(class, cmd, data) ((sizeof(data)<<16) + ((class)<<8) + (cmd) +
    _POSIX_DEVDIR_FROM)
```

```

#define __DIOT(class, cmd, data) ((sizeof(data)<<16) + ((class)<<8) + (cmd) +
_POSIX_DEVDIR_TO)
#define __DIOF(class, cmd, data) ((sizeof(data)<<16) + ((class)<<8) + (cmd) +
_POSIX_DEVDIR_TOFROM)
#define __DION(class, cmd) (((class)<<8) + (cmd) + _POSIX_DEVDIR_NONE)

```

Важно понимать, как эти макросы упаковывают данные, создавая команду. Восьмибитовый класс (определённый в `<devctl.h>`) комбинируется с характерным для менеджера восьмибитовым подтипом, и помещается в младшие 16 бит целого числа.

Старшие 16 бит содержат направление (TO, FROM), а также рекомендацию о размере передаваемой структуры данных. Этот размер является только рекомендацией для однозначного определения сообщений, которые могут использовать тот же класс и код, но передавать различные структуры данных.

В следующем примере генерируется `cmd` для индикации того, что клиент посылает данные серверу (TO), но не получает ничего в ответ. Единственными битами, которые просматривает библиотека уровня менеджера ресурсов, являются биты TO и FROM, определяющие, какие аргументы переданы в `MsgSend()`.

```

struct _my_devctl_msg {
    .....
}
#define MYDCMD _ _DIOT(_DCMD_MISC, 0x54, struct _my_devctl_msg)

```

i *Размер структуры, которая передаётся как последний параметр макросов `__DIO*`, должна быть меньше чем $2^{14}=16К$. Всё, что больше этого, пересечётся с верхними двумя битами направления.*

Как указано в комментарии `/* char data[nbytes] */` к структуре `_io_devctl`, данные следуют непосредственно за этой структурой сообщения.

Пример кода для обработки сообщений `_IO_DEVCTL`

Следующий фрагмент кода может быть добавлен к любому из примеров, приведенных в разделе "Примеры простого менеджера ресурсов устройства". Оба примера этих кодов поддерживают имя `/dev/sample`. С изменениями, указанными ниже, клиент может использовать функцию `devctl()`, чтобы установить и получить значение глобальной переменной (целой в этом случае), которая поддерживается в менеджере ресурсов.

Первое добавление определяет, что будут делать команды `devctl()`. Обычно это помещается в заголовочный файл – общий или совместно используемый:

```

typedef union _my_devctl_msg {
    int tx;           // Заполняется клиентом при отсылке (send)
    int rx;           // Заполняется клиентом при отклике (reply)
} data_t

#define MY_CMD_CODE 1
#define MY_DEVCTL_GETVAL __DIOF(_DCMD_MISC, MY_CMD_CODE + 0, int)
#define MY_DEVCTL_SETVAL __DIOT(_DCMD_MISC, MY_CMD_CODE + 1, int)
#define MY_DEVCTL_SETGET __DIOF(_DCMD_MISC, MY_CMD_CODE + 2, union _my_devctl_msg)

```

В приведенном примере мы определяем три команды, которые может использовать клиент:

`MY_DEVCTL_SETVAL`

Присваивает глобальной переменной сервера значение (целое), заданное клиентом.
MY_DEVCTL_GETVAL

Получает глобальную переменную сервера и помещает это значение в буфер клиента.

MY_DEVCTL_SETGET

Присваивает глобальной переменной сервера значение (целое), заданное клиентом, и возвращает в буфер клиента предыдущее значение этой переменной.

Вот что добавляется в функцию *main()*:

```
io_funcs.devctl=io_devctl; /* Для обработки _IO_DEVCTL посланной функцией devctl() ./
```

И следующий код добавляется перед функцией *main()*:

```
int handle_devctl(resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T *ocb);  
int global_integer = 0;
```

Наконец, Вам необходимо включить новую функцию обработки сообщения _IO_DEVCTL:

```
int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg, RESMGR_OCB_T *ocb) {  
    int nbytes, status, previous;  
    union {  
        data_t data;  
        int data32;  
        // ... другие типы devctl, которые Вы можете получать  
    } *rx_data;  
  
    /*  
    Вот общий код обработки в случаях DCMD_ALL_*  
    Вы можете выполнить его перед или после того, как Вы перехватываете devctl. Здесь мы не  
    используем никаких предопределённых значений, позволяя вначале выполнить обработку системе.  
    */  
  
    if ((status = iofunc_devctl_default(ctp, msg, ocb)) != _RESMGR_DEFAULT) {  
        return(status);  
    }  
    status = nbytes = 0;  
  
    /*  
    Заметьте: здесь предполагается, что Вы можете подсоединить весь объём данных для devctl в  
    одно сообщение. В действительности Вы, возможно, захотите исполнить MsgReadv(), когда Вы  
    знаете тип сообщения, которое получаете, чтобы перекачать все данные, вместо того чтобы  
    предполагать, что все данные присоединены в сообщении. Мы устанавливаем в нашей main-  
    программе, что допускаем общий размер сообщения до 2К, так что мы не беспокоимся об этом в  
    данном примере, где имеем дело с целыми (int).  
    */  
  
    rx_data = _DEVCTL_DATA(msg->i);  
  
    /*  
    Три примера операций devctl.  
    SET: Установка значения (int) на сервере  
    GET: Получение значения (int) с сервера  
    SETGET: Установка нового значения и возврат предыдущего значения  
    */  
  
    switch (msg->i.dcmd) {  
    case MY_DEVCTL_SETVAL:  
        global_integer = rx_data->data32;  
        nbytes = 0;  
        break;
```

```

case MY_DEVCTL_GETVAL:
    rx_data->data32 = global_integer;
    nbytes = sizeof(rx_data->data32);
    break;

case MY_DEVCTL_SETGET:
    previous = global_integer;
    global_integer = rx_data->data.tx;
    rx_data->data.rx = previous;           //Данные tx переписываются
    nbytes = sizeof(rx_data->data.rx);
    break;

default:
    return(ENOSYS);
}

/* Очистка сообщения return ... заметьте, что мы сохранили наши данные до этого */
memset(&msg->o, 0, sizeof(msg->o));
/*
Если Вы хотите передать в поле return для devctl() что-то другое, Вы можете сделать это с
помощью этого поля.
*/
msg->o.ret_val = status;

/* Указывает число байтов и возвращается сообщение */
msg->o.nbytes = nbytes;
return(_RESMGR_PTR(ctp, &msg->o, sizeof(msg->o) + nbytes));
}

```

Работая с кодом обработчика *devctl()*, Вы должны быть знакомы со следующим:

- Обработчик *devctl()* по умолчанию вызывается перед тем, как мы начинаем обрабатывать сообщения. Это позволяет обрабатывать обычные системные сообщения. Если сообщение не обслужено обработчиком по умолчанию, то он возвращает `_RESMGR_DEFAULT`, указывая, что сообщение может быть особым сообщением. Это предполагает, что мы будем проверять поступающие сообщения на предмет команд, которые понимает наш менеджер ресурсов.
- Передаваемые данные располагаются непосредственно после структуры *io_devctl_t*. Указатель на их местоположение может быть получен с использованием макроса `_DEVCTL_DATA(msg->i)`, определённого в `<devctl.h>`. Аргумент этого макроса **должен** быть структурой **входного** сообщения – если это будет структура сообщения или указатель на структуру входного сообщения, то полученный указатель не будет ссылаться на правильное местоположение.

Для удобства определено объединение всех сообщений, которые может получать сервер. Однако, оно не будет работать с сообщениями, несущими большой объём данных. В этом случае, чтобы прочитать сообщение от клиента Вам надо использовать функцию *resmgr_msgread()*. Наши сообщения (в примере) никогда не будут больше, чем `sizeof(int)` и они прекрасно помещаются в минимальном по размеру буфере получаемого сообщения.
- Последним аргументом функции *devctl()* является указатель на целое. Если этот указатель передается в функцию, то целое заполняется значением, хранящимся в ответном reply-сообщении – в `msg->o.ret_val`. Это позволяет менеджеру ресурсов вернуть элементарную информацию о состоянии, не используя операции *devctl()* ядра. В данном примере это не используется.
- Данные, возвращаемые клиенту, размещаются в конце ответного reply-сообщения. Это тот же механизм, который используется для входных данных, так что мы можем использовать функцию `_DEVCTL_DATA()`, чтобы получить указатель на это

местоположение. Для больших по размеру ответов, которые могут не поместиться в буфере входных сообщений сервера, Вы можете использовать один из механизмов ответа, описанных в разделе "Методы возвращения и ответа". Кроме того, в этом примере мы возвращаем только целое, которое без проблем помещается в буфер входных сообщений.

Если Вы добавите следующий код обработчика, клиент сможет открывать `/dev/sample` и впоследствии устанавливать и получать значение глобальной целой переменной:

```
int main(int argc, char **argv) {
    int      fd, ret, val;
    data_t  data;

    if ((fd = open("/dev/sample", O_RDONLY)) == -1) { return(1); }

    /* Выясняем, какое значение установлено при инициализации */
    val = -1;
    ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
    printf("GET вернул %d w/ значение сервера %d \n", ret, val);

    /* Устанавливаем какое-нибудь другое значение */
    val = 25;
    ret = devctl(fd, MY_DEVCTL_SETVAL, &val, sizeof(val), NULL);
    printf("SET вернул %d \n", ret);

    /* Проверяем, действительно ли мы установили значение */
    val = -1;
    ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
    printf("GET вернул %d w/ значение сервера %d == 25? \n", ret, val);

    /* Теперь выполняем комбинацию set/get */
    memset(&data, 0, sizeof(data));
    data.tx = 50;
    ret = devctl(fd, MY_DEVCTL_SETGET, &data, sizeof(data), NULL);
    printf("SETGET вернул %d w/ значение сервера %d == 25?\n", ret, data.rx);

    /* Проверяем, как сработала set/get */
    val = -1;
    ret = devctl(fd, MY_DEVCTL_GETVAL, &val, sizeof(val), NULL);
    printf("GET вернул %d w/ значение сервера %d == 50? \n", ret, val);

    return(0);
}
```

Обработка *ionotify()* и *select()*

Функции `ionotify()` и `select()` используются клиентом для запроса об определённых состояниях менеджера ресурсов (выполнении неких условий, напр., доступно ли чтение данных). Эти условия могут быть выполнены, а могут и нет. Менеджер ресурсов может ответить:

- Немедленно, проверив условия и вернуть ответ при любом их состоянии.
- Доставить событие позже, когда условия будут соблюдены (это называется "ввести менеджер ресурсов").

Функция `select()` отличается от `ionotify()` тем, что большая часть работы выполняется в библиотеке. Например, код клиента не будет и подозревать, что происходят какие-то события,

не будет знать о блокирующих функциях, ожидающих события. Всё это скрыто в коде библиотечной функции `select()`. Однако с точки зрения менеджера ресурсов нет различий между функциями `ionotify()` и `select()`. Они обрабатываются одним и тем же кодом. Для получения более подробной информации по функциям `ionotify()` и `select()` см. "Справочник библиотечных функций".

i В настоящее время API для обработки уведомлений от Вашего менеджера ресурсов не поддерживает многопоточные клиентские процессы в полной мере. Проблемы могут возникнуть, когда поток в процессе клиента запросит уведомление, и другие потоки в том же процессе будут иметь дело с менеджером ресурсов. Проблема снимается, когда потоки принадлежат различным процессам.

Поскольку функции `ionotify()` и `select()` требуют от менеджера ресурсов одинаковой работы, они обе посылают менеджеру ресурсов сообщение `_IO_NOTIFY`. За это сообщение отвечает обработчик `io_notify`. Давайте начнём с рассмотрения формата самого сообщения:

```
struct _io_notify {
    uint16_t      type;
    uint16_t      combine_len;
    int32_t       action;
    int32_t       flags;
    struct sigevent event;
};

struct _io_notify_reply {
    uint32_t      flags;
};

typedef union {
    struct _io_notify      i;
    struct _io_notify_reply o;
} io_notify_t;
```

Как и для всех сообщений менеджеру ресурсов, мы определяем объединение, содержащее структуру входного сообщения (приходящего в менеджер ресурсов), и ответную, или структуру вывода (идущую обратно клиенту). Прототип обработчика `io_notify` имеет аргумент:

```
io_notify_t .msg
```

с указателем на объединение, содержащее сообщение. Полями структуры ввода являются:

```
type
combine_len
action
flags
event.
```

Поле `type` имеет значение `_IO_NOTIFY`.

Поле `combine_len` имеет смысл для комбинированных сообщений, см. раздел "Комбинированные сообщения" в этой главе.

Поле `action` используется вспомогательной функцией `iofunc_notify()`, чтобы сообщить ей, будет ли она:

- просто проверить состояние в настоящий момент;

- проверять состояние в настоящий момент, и, если это состояние не выполняется, взводить его;
- просто взводить.

Поскольку за это отвечает функция *iofunc_notify()*, Вам не надо об этом заботиться.

Поле *flags* содержит состояния, которые могут интересовать клиента, и оно может быть сформировано любым сочетанием из следующих значений:

_NOTIFY_COND_INPUT

Это состояние возникает, когда становятся доступными один или более блоков входных данных (т.е. клиенты могут теперь выполнять чтение). Количество блоков по умолчанию равно 1, но может быть изменено. Определение понятия блока - это Ваше дело: для символьных устройств, таких как последовательный порт, это должен быть символ; для очереди сообщений POSIX это должно быть сообщение. Каждый менеджер ресурсов выбирает соответствующий объект.

_NOTIFY_COND_OUTPUT

Это состояние возникает, когда появляется свободное пространство в выходном буфере для одного или более блоков данных (т.е. клиент может теперь выполнять запись). По умолчанию количество блоков равно 1, но может быть изменено. Определение блока возлагается на Вас – одни менеджеры ресурсов могут предполагать по умолчанию полностью пустой буфер вывода, тогда как другие могут выбирать только часть буфера.

_NOTIFY_COND_OBAND

Состояние возникает, когда доступен один или более блоков данных вспомогательного канала. Количество блоков по умолчанию равно 1, но может быть изменено. Определение блока данных вспомогательного канала является специфичным для каждого менеджера ресурсов.

Поле *event* определяет, что именно отправляет менеджер, когда наступает некое состояние.

Менеджер ресурсов должен хранить список клиентов, запросивших оповещения при наступлении определённых состояний, вместе с событиями, которые должны использоваться для оповещения. Когда возникает некое состояние, менеджер ресурсов должен пройти по списку, отыскивая клиентов, которые интересуются этим состоянием, и затем инициировать соответствующее событие. И так же, если клиент закрывает свой файловый дескриптор, из списка должны быть удалены любые запросы уведомлений, относящиеся к этому клиенту. Для упрощения этой задачи предоставлены структура и вспомогательные функции:

- структура *iofunc_notify_t*

Содержит три списка уведомлений, по одному на каждое возможное состояние. Каждый список – это список клиентов, которые уведомляются о наступлении этого состояния.

- *iofunc_notify()*

Добавляет или удаляет входы запросов уведомлений, а также опрашивает состояния. Вызывайте эту функцию изнутри Вашей функции обработки *io_notify*.

- *iofunc_notify_trigger()*

Посылает уведомления находящимся в очереди клиентам. Вызывайте эту функцию, когда возникает одно или более состояний.

- *iofunc_notify_remove()*

Удаляет входы запросов уведомлений из списка. Вызывайте эту функцию, когда клиент закрывает свой файловый дескриптор.

Пример кода для обработки сообщений `_IO_NOTIFY`

Следующие фрагменты кода могут быть добавлены в любой из примеров, приведенных в разделе "Примеры простого менеджера ресурсов устройства". Оба этих примера поддерживают имя `/dev/sample`. С этими изменениями, клиенты могут использовать запись (*writes*), чтобы посылать данные, которые будут храниться как отдельные сообщения. Другие клиенты могут использовать *ionotify()* или *select()*, чтобы запрашивать уведомления, когда эти данные будут доставлены. Когда клиенты получают уведомление, они могут выполнить чтение (*reads*), чтобы получить данные. Вам понадобится заменить этот код, размещённый перед функцией *main()*:

```
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t      connect_funcs;
static resmgr_io_funcs_t          io_funcs;
static iofunc_attr_t              attr;

    следующим:

struct device_attr_s;
#define IOFUNC_ATTR_T    struct device_attr_s

#include <sys/iofunc.h>
#include <sys/dispatch.h>

/*
 * Определение структуры и переменных для хранения полученных данных.
 * Когда клиент пишет для нас данные, мы помещаем их сюда.
 * Когда клиент выполняет чтение, мы берём данные отсюда.
 * В результате... простая очередь сообщений.
 */
typedef struct item_s {
    struct item_s    * next;
    char             * data;
} item_t;

/* расширенная структура атрибутов */
typedef struct device_attr_s {
    iofunc_attr_t    attr;
    iofunc_notify_t notify[3];          /* список уведомлений, используемый
iofunc_notify*() */
    item_t           *firstitem;      /* очередь элементов */
    int              nitems;         /* число элементов в очереди */
} device_attr_t;

/* мы имеем только одно устройство, device_attr - его структура атрибутов */
static device_attr_t    device_attr;

int io_read(resmgr_context_t *ctp,      io_read_t  *msg,      RESMGR_OCB_T *ocb);
int io_write(resmgr_context_t *ctp,     io_write_t *msg,      RESMGR_OCB_T *ocb);
int io_notify(resmgr_context_t *ctp,    io_notify_t *msg,     RESMGR_OCB_T *ocb);
int io_close_ocb(resmgr_context_t *ctp, void *reserved,    RESMGR_OCB_T *ocb);

static resmgr_connect_funcs_t connect_funcs;
static resmgr_io_funcs_t      io_funcs;
```

Нам нужно место для хранения данных, специфичных для нашего устройства. Хорошим местом для этого является структура атрибутов, которую мы можем присоединить к имени, которое мы зарегистрировали: `/dev/sample`. Так, в коде, приведенном выше, мы определили для этой цели *device_attr_t* и *IOFUNC_ATTR_T*. Более подробно мы обсудили этот тип

специфичной для устройства структуры атрибутов в разделе "Расширенные структуры управления данными (DCS)". Нам необходимы два типа специфичных для устройства данных:

- массив из трёх списков уведомлений – по одному для каждого возможного состояния, о котором клиент может запросить уведомления. В *device_attr_t* мы назвали это *notify*;
- очередь на сохранение данных, которые нами получены на запись, и используемых для ответа клиенту. Для этого мы определяем *item_t* - это тип, который содержит данные для одиночного элемента очереди, а также указатель *item_t* на следующий элемент. В *device_attr_t* мы используем *firstitem* (указатель на первый элемент очереди) и *nitems* (число элементов). Заметьте, что мы удалили определение *attr*, поскольку используем вместо него *device_attr*.

Конечно, мы передаём библиотеке менеджера ресурсов адрес наших обработчиков, так что он будет знать, как их вызывать. В коде функции *main()*, где мы вызываем функцию *iofunc_func_init()*, для регистрации наших обработчиков добавим следующий код:

```
/* инициализация функций обработки сообщений */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                 _RESMGR_IO_NFUNCS,      &io_funcs);
io_funcs.notify = io_notify; /* для обработки _IO_NOTIFY, отсылается как */
                             /* результат вызовов клиентом ionotify() и select()*/
io_funcs.write = io_write;
io_funcs.read = io_read;
io_funcs.close_ocb = io_close_ocb;
```

И, поскольку мы используем *device_attr* вместо *attr*, нам надо изменить в *main()* код всюду, где мы это используем. Так, нам надо заменить этот код:

```
/* присоединение нашего имени устройства */
id = resmgr_attach(dpp, /* обработчик диспетчеризации */
                  &resmgr_attr, /* атрибуты менеджера ресурсов */
                  "/dev/sample", /* имя устройства */
                  _FTYPE_ANY, /* тип открытия */
                  0, /* флаги */
                  &connect_funcs, /* подпрограммы связи */
                  &io_funcs, /* подпрограммы ввода/вывода */
                  &attr); /* идентификатор-описатель */
```

следующим:

```
/* инициализация используемой устройством структуры атрибутов */
iofunc_attr_init(&device_attr.attr, S_IFNAM | 0666, 0, 0);
IOFUNC_NOTIFY_INIT(device_attr.notify);
device_attr.firstitem = NULL;
device_attr.nitems = 0;

/* присоединение нашего имени устройства */
id = resmgr_attach(dpp, /* обработчик диспетчеризации */
                  &resmgr_attr, /* атрибуты менеджера ресурсов */
                  "/dev/sample", /* имя устройства */
                  _FTYPE_ANY, /* тип открытия */
                  0, /* флаги */
                  &connect_funcs, /* подпрограммы связи */
                  &io_funcs, /* подпрограммы ввода/вывода */
                  &device_attr); /* идентификатор-описатель */
```

Заметьте, что мы установили данные, специфичные для нашего устройства, в *device_attr*. И в вызове функции *resmgr_attach()* мы передаём *&device_attr* (вместо *&attr*) как параметр .

Теперь Вам надо включить новую функцию-обработчик для обработки сообщения *_IO_NOTIFY*:

```
int io_notify(resmgr_context_t *ctp, io_notify_t *msg, RESMGR_OCB_T *ocb) {
```

```

device_attr_t      *datatr = (device_attr_t *) ocb->attr;
int      trig;

/*
 *      'trig' будет говорить функции iofunc_notify(), какие состояния удовлетворены
 *      в текущий момент. 'datatr->nitems' - это число сообщений в нашем списке
 *      хранимых сообщений.
 */
trig = _NOTIFY_COND_OUTPUT;      /* клиент может всегда дать нам данные */
if (datatr->nitems > 0)
    trig |= _NOTIFY_COND_INPUT; /* у нас имеются какие-то доступные данные */
/*
 * iofunc_notify() выполнит всю необходимую обработку, включая добавление
 * при необходимости клиента в список уведомления.
 */
return (iofunc_notify(ctp, msg, datatr->notify, trig, NULL, NULL));
}

```

Как сказано выше, наш обработчик *io_notify* будет вызываться, когда клиент вызовет *ionotify()* или *select()*. В обработчике мы запоминаем обратившихся к нам клиентов, и то, о каких состояниях они хотят получить уведомления. Мы также должны немедленно передать ответ с состояниями, которые уже возникли. Вспомогательная функция *iofunc_notify()* делает это весьма простым.

Первое, что мы делаем – это разбираемся, какие из обрабатываемых нами состояний выполняются в настоящий момент. В этом примере мы всегда можем разрешить запись, так что в коде, приведенном выше, мы установили в *trig* бит *_NOTIFY_COND_OUTPUT*. Мы также проверяем *nitems*, чтобы посмотреть, имеются ли у нас данные, и если да, устанавливаем *_NOTIFY_COND_INPUT*. Затем мы вызываем *iofunc_notify()*, передавая ей полученное сообщение (*msg*), список уведомлений (*notify*), и выполненные состояния (*trig*). Если одно из состояний, о которых спрашивает клиент, выполняется, и клиент хочет от нас последовательного опроса состояния перед взведением, то *iofunc_notify()* будет возвращать значение, указывающее, какое состояние выполняется, и состояние не будет взведено. В противном случае состояние будет взведено. В любом случае мы вернёмся из обработчика с кодом, возвращённым функцией *iofunc_notify()*.

Раньше, когда мы говорили о трёх возможных состояниях, мы упоминали, что если Вы определили *_NOTIFY_COND_INPUT*, клиент получает уведомление, когда один или более блоков входных данных становятся доступными, и что количество этих блоков определяется Вами. Подобное мы говорили и о *_NOTIFY_COND_OUTPUT* и *_NOTIFY_COND_OBAND*. В приведенном выше коде мы приняли для всех узлов по умолчанию количество блоков равным 1. Если Вы желаете использовать что-то другое, Вы должны объявить массив таким образом:

```
int notifycounts[3]={10,2,1};
```

Это устанавливает количество блоков для *_NOTIFY_COND_INPUT* равным 10, для *_NOTIFY_COND_OUTPUT* – 2; и *_NOTIFY_COND_OBAND* равным 1. Мы должны передать *notifycounts* функции *iofunc_notify()* как второй с конца параметр. Затем, когда поступают данные, мы уведомляем всех тех клиентов, которые запрашивали уведомление. В этом примере данные поступают от клиента, посылающего нам сообщения *_IO_WRITE*, и мы обрабатываем их, используя обработчик *io_write*.

```

int io_write(resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb) {
    device_attr_t      *datatr = (device_attr_t *) ocb->attr;
    int      i;
    char      *p;
    int      status;
    char      *buf;
    item_t      *newitem;

```

```

if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK) return (status);
if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE) return (ENOSYS);
if (msg->i.nbytes > 0) {
    /* получение данных и сохранение их отдельно */
    if ((newitem = malloc(sizeof(item_t))) == NULL) return (errno);
    if ((newitem->data = malloc(msg->i.nbytes+1)) == NULL) {
        free(newitem);
        return (errno);
    }
    /* перепрочтение данных из буфера сообщений посылающего */
    resmgr_msgread(ctp, newitem->data, msg->i.nbytes, sizeof(msg->i));
    newitem->data[msg->i.nbytes] = NULL;

    if (dattr->firstitem)
        newitem->next = dattr->firstitem;
    else
        newitem->next = NULL;
    dattr->firstitem = newitem;
    dattr->nitems++;

    /*
     * уведомление клиентов, заказавших уведомления, когда имеются данные
     */

    if (IOFUNC_NOTIFY_INPUT_CHECK(dattr->notify, dattr->nitems, 0))
        iofunc_notify_trigger(dattr->notify, dattr->nitems, IOFUNC_NOTIFY_INPUT);
}

/* установка числа байтов (возвращаемых клиентской write()) */
_IO_SET_WRITE_NBYTES(ctp, msg->i.nbytes);

if (msg->i.nbytes > 0)
    ocb->attr->attr.flags |= IOFUNC_ATTR_MTIME | IOFUNC_ATTR_CTIME;

return (_RESMGR_NPARTS(0));
}

```

Важной частью обработчика *io_write*, приведенного выше, является код внутри следующей секции:

```

if (msg-> i.nbytes>0) {
    .....
}

```

Здесь мы впервые выделяем место под входные данные, и затем используем функцию *resmgr_msgread()*, чтобы копировать данные из буфера послыки клиента в выделенное пространство памяти. Затем мы добавляем данные к нашей очереди.

Далее, мы передаём количество доступных блоков входных данных в *IOFUNC_NOTIFY_INPUT_CHECK()*, чтобы просмотреть, имеется ли достаточно блоков, чтобы уведомить об этом клиента. Здесь проверяется на соответствие *notifycounts*, о чем мы говорили выше, когда рассматривали обработчик *io_notify*. Если имеется достаточно количество доступных блоков, то мы вызываем функцию *iofunc_notify_trigger()*, сообщая ей, что доступно *nitems* данных (*IOFUNC_NOTIFY_INPUT* подразумевает, что ввод доступен). Функция *iofunc_notify_trigger()* просматривает список клиентов, находит запросы на уведомление (*notify*) и уведомляет всех, кто запросил о доступности данных.

Любой клиент, получивший уведомление, выполняет чтение, чтобы получить данные. В нашем примере мы обрабатываем это следующим обработчиком *io_read*:

```

int io_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb) {
    device_attr_t *dattr = (device_attr_t *) ocb->attr;

```

```

int    status;

if ((status = iofunc_read_verify(ctp, msg, ocb, NULL)) != EOK) return (status);
if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE)    return (ENOSYS);
if (dattr->firstitem) {
    int    nbytes;
    item_t *item, *prev;

    /* получение последнего пункта */
    item = dattr->firstitem;
    prev = NULL;
    while (item->next != NULL) {
        prev = item;
        item = item->next;
    }
    /*
     * вычисление числа передаваемых данных, запись данных на
     * буфер отклика клиенту, выравнивание, если мы имеем больше байтов,
     * чем тот запросил, и удаление пункта из нашего списка
     */
    nbytes = min (strlen (item->data), msg->i.nbytes);

    /* установка числа байтов (возвращаемых клиентской read()) */
    _IO_SET_READ_NBYTES (ctp, nbytes);

    /*
     * теперь записываем байты в буфер отклика клиенту, поскольку теперь
     * нам не надо беспокоиться о данных
     */
    resmgr_msgwrite (ctp, item->data, nbytes, 0);

    /* удаление данных из очереди */
    if (prev)
        prev->next = item->next;
    else
        dattr->firstitem = NULL;
    free(item->data);
    free(item);
    dattr->nitems--;
}
else {
    /* the read() will return with 0 bytes */
    _IO_SET_READ_NBYTES (ctp, 0);
}

/* отмечаем время доступа как неверное (мы как раз имели к нему доступ) */
if (msg->i.nbytes > 0)
    ocb->attr->attr.flags |= IOFUNC_ATTR_ETIME;
return (EOK);
}

```

Важной частью этого обработчика *io_read* является код внутри секции:

```

if (firstitem) {
    .....
}

```

Прежде всего мы проходим по очереди в поисках самого «старого» элемента. Затем мы используем функцию *resmgr_msgwrite()*, чтобы записать данные в буфер ответа клиенту. Мы делаем это сейчас, поскольку следующим шагом является освобождение памяти, которую мы используем для хранения данных. Мы также удаляем этот элемент из очереди.

В заключение, если клиент закрывает свой файловый дескриптор, мы должны удалить его из списка клиентов. Это делается с использованием обработчика *io_close_ocb*:

```

int io_close_ocr(resmgr_context_t *ctp, void *reserved, RESMGR_OCB_T *ocr)
{
    device_attr_t    *datr = (device_attr_t *) ocr->attr;
    /*
     * Клиент закрыл свой файловый дескриптор или был «убит».
     * Удаляем его из списка уведомлений.
     */

    iofunc_notify_remove(ctp, datr->notify);
    return (iofunc_close_ocr_default(ctp, reserved, ocr));
}

```

В обработчике *io_close_ocr* мы вызвали функцию *iofunc_notify_remove()* и передали ей *ctp* (содержащий информацию, идентифицирующую клиента) и *notify* (содержащий список клиентов), чтобы удалить клиента из списка.

Обработка приватных сообщений и импульсов

Может потребоваться, чтобы менеджер ресурсов получал и обрабатывал импульсы. Например, потому, что обработчик прерываний завершает работу, посылая импульс, или же импульс посылается другими потоками или процессами. Главная проблема с импульсами состоит в том, что они принимаются как сообщения – это означает, что поток явно выполняет *MsgReceive()*, чтобы получить импульс. Но если этот импульс посылается в тот же канал, который менеджер ресурсов использует в своём основном интерфейсе сообщений, он будет приниматься библиотекой менеджера ресурсов. Поэтому нам надо рассмотреть, как менеджер ресурсов может подключить код импульса к подпрограмме обработки и передавать эту информацию библиотеке.

Функция *pulse_attach()* может использоваться для того, чтобы подключить код импульса (поле *code* в импульсе) к функции обработки. Поэтому, когда диспетчерский уровень получает импульс, он будет проверять поле кода импульса и смотреть, какой присоединённый обработчик вызывается для обработки сообщения-импульса.

Вы также можете определить собственный диапазон приватных сообщений для связи с менеджером ресурсов. Заметьте, что диапазон от 0x0 до 0x1fff зарезервирован за операционной системой. Чтобы подключить приватный диапазон, Вам надо использовать функцию *message_attach()*.

В этом примере мы создаём некий менеджер ресурсов, но на этот раз мы подключаем диапазон приватных сообщений и подсоединяем импульс, который затем используется как событие таймера.

```

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>

#define THREAD_POOL_PARAM_T    dispatch_context_t
#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_func;
static resmgr_io_funcs_t        io_func;
static iofunc_attr_t            attr;

int timer_tick(message_context_t *ctp, int code, unsigned flags, void *handle) {

```



```

union sigval value = ctp->msg->pulse.value;
/*
 * Делается какая-то полезная работа на каждое срабатывание таймера
 */
printf("получено событие таймера, значение %d\n", value.sival_int);
return 0;
}

int message_handler(message_context_t *ctp, int code, unsigned flags, void *handle) {
printf("получено приватное сообщение, тип %d\n", code);
return 0;
}

int main(int argc, char **argv) {
thread_pool_attr_t pool_attr;
resmgr_attr_t resmgr_attr;
struct sigevent event;
struct itimer itime;
dispatch_t *dpp;
thread_pool_t *tpp;
resmgr_context_t *ctp;
int timer_id;
int id;

if((dpp = dispatch_create()) == NULL) {
fprintf(stderr, "%s: Невозможно создать дескриптор диспетчеризации.\n", argv[0]);
return EXIT_FAILURE;
}

memset(&pool_attr, 0, sizeof pool_attr);
pool_attr.handle = dpp;
/*
 * Если Вы используете приватные сообщения или импульсы
 * с функциями message_attach() или pulse_attach(),
 * то Вы ДОЛЖНЫ использовать функции диспетчеризации
 * (т.е. dispatch_block(), dispatch_handler(), ...),
 * А НЕ функции resmgr (resmgr_block(), resmgr_handler()).
 */
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func = dispatch_block;
pool_attr.handler_func = dispatch_handler;
pool_attr.context_free = dispatch_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

if((tpp = thread_pool_create(&pool_attr, POOL_FLAG_EXIT_SELF)) == NULL) {
fprintf(stderr, "%s: Невозможно инициализировать пул потоков.\n", argv[0]);
return EXIT_FAILURE;
}

iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_func, _RESMGR_IO_NFUNCS,
&io_func);
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

if((id = resmgr_attach(dpp, &resmgr_attr, "/dev/sample", _FTYPE_ANY, 0,
&connect_func, &io_func, &attr)) == -1) {
fprintf(stderr, "%s: Невозможно подсоединить имя.\n", argv[0]);
return EXIT_FAILURE;
}

/* Мы хотим обрабатывать наши собственные приватные сообщения типа от 0x5000 до 0x5fff */

```

```

if(message_attach(dpp, NULL, 0x5000, 0x5fff, &message_handler, NULL) == -1) {
    fprintf(stderr, "Невозможно подсоединить диапазон частных сообщений.\n");
    return EXIT_FAILURE;
}

/* Инициализируем структуру событий, и присоединяем к ней импульс */
if((event.sigev_code = pulse_attach(dpp, MSG_FLAG_ALLOC_PULSE, 0, &timer_tick,
    NULL)) == -1) {
    fprintf(stderr, "Невозможно присоединить импульс таймера.\n");
    return EXIT_FAILURE;
}

/* Подсоединяемся к нашему каналу */
if((event.sigev_coid = message_connect(dpp, MSG_FLAG_SIDE_CHANNEL)) == -1) {
    fprintf(stderr, "Невозможно подсоединить канал.\n");
    return EXIT_FAILURE;
}

event.sigev_notify = SIGEV_PULSE;
event.sigev_priority = -1;
/* Мы можем создать несколько таймеров и использовать для каждого */
/* различные значения sigev */
event.sigev_value.sival_int = 0;

if((timer_id = TimerCreate(CLOCK_REALTIME, &event)) == -1) {;
    fprintf(stderr, "Невозможно подсоединить канал и связь.\n");
    return EXIT_FAILURE;
}

/* И теперь устанавливаем наш таймер на ежесекундное срабатывание */
itime.nsec = 1000000000;
itime.interval_nsec = 1000000000;
TimerSettime(timer_id, 0, &itime, NULL);

/* Никогда не вернётся */
thread_pool_start(tpp);
}

```

Мы можем самостоятельно определить код импульса (например, `#define OurPulseCode 57`) или же можем попросить функцию `pulse_attach()` динамически сгенерировать его для нас (и вернуть значение кода импульса как код возврата из функции `pulse_attach()`), задав код импульса как `_RESMGR_PULSE_ALLOC`.

Подробнее о получении и генерации импульсов см. описание функций `pulse_attach()`, `MsgSendPulse()`, `MsgDeliverEvent()` и `MsgReceive()` в "Справочнике библиотечных функций".

Обработка сообщений `open()`, `dup()` и `close()`

Библиотека менеджера ресурсов предоставляет ещё одну удобную службу: она известна как обработка `dup()` сообщений. Предположим, что клиент выполнил код, который в конечном счёте завершился исполнением:

```

fd = open("/dev/sample", O_RDONLY);
.....
fd2 = dup(fd);
.....
fd3 = dup(fd);
.....

```

```
close(fd3);
.....
close(fd2);
.....
close(fd);
```

Наш менеджер ресурсов получит сообщение `_IO_CONNECT` от первоначального `open()`, следом за ним два сообщения `_IO_DUP` от двух вызовов `dup()`. Затем, когда клиент выполнил вызовы `call()`, мы получим три сообщения `_IO_CLOSE`. Поскольку функции `dup()` генерируют дубликаты файловых дескрипторов, мы не хотели бы выделять новые `ocb` под каждый из них. И так как мы не выделяем новые `ocb` для каждого вызова `dup()`, нам не надо и освобождать память при каждом сообщении `_IO_CLOSE`, когда такие сообщения поступают. Если мы так и делаем, первый же вызов `close()` уничтожает `ocb`.

Библиотека менеджера ресурсов знает, как управлять этим. Она ведет подсчёт количества сообщений `_IO_DUP` и `_IO_CLOSE`, посылаемых клиентом. Только последнее сообщение `_IO_CLOSE` приведёт к вызову библиотекой обработчика `_IO_CLOSE_OCB`.

i Большинство пользователей библиотеки захотят использовать функции по умолчанию, управляющие сообщениями `_IO_DUP` и `_IO_CLOSE`. Вероятнее всего, Вы никогда не будете перекрывать принимаемые по умолчанию действия.

Обработка разблокирования клиента

по сигналам или тайм-аутам

Другой удобной службой, предлагаемой библиотекой менеджера ресурсов, является разблокирование.

Когда клиент выполняет запрос (например, `read()`), он переводится (через функции библиотеки клиента) в сообщение `MsgSend()` нашему менеджеру ресурсов. Вызов функции `MsgSend()` является блокирующим вызовом. Если в то время, когда `MsgSend()` ожидает выполнения, клиент получает сигнал, нашему менеджеру ресурсов потребуется какое-либо указание об этом, чтобы он мог прервать выполнение запроса.

Поскольку библиотека устанавливает флаг `_NTO_CHF_UNBLOCK`, когда вызывает `ChannelCreate()`, мы будем получать импульс всякий раз, когда клиент пытается разблокироваться из `MsgSend()`, на который мы имеем свой `MsgReceive()`.

Отвлекаясь от темы, вспомним, что в модели сообщений QNX6, клиент, в результате вызова `MsgSend()`, может находиться в одном из двух состояний. Если сервер ещё не получил сообщение (из функции сервера `MsgReceive()`), клиент находится в SEND-блокированном состоянии – он ожидает, когда сервер получит сообщение. Когда сервер действительно получает сообщение, клиент переходит в REPLY-блокированное состояние – клиент теперь ожидает, когда сервер ответит ему на сообщение (через `MsgReply()`).

Если в этот момент и был сгенерирован импульс, библиотека менеджера ресурсов обрабатывает сообщение-импульс и создает сообщение `_IO_UNBLOCK`.

Изучая структуры `resmgr_io_funcs_t` и `resmgr_connect_funcs_t`, Вы заметите, что в действительности существует два обработчика разблокирующих сообщений: одно в структуре функций ввода/вывода и одно в структуре функций связи.

Почему две? Потому что мы можем получить разблокирование в одном из двух мест. Мы можем получить импульс сразу после того, как клиент послал сообщение `_IO_OPEN` (но перед

тем как мы ответили на него), или получить импульс разблокирования во время обработки сообщения ввода/вывода.

После того, как мы выполним обработку сообщения `_IO_CONNECT`, разблокирующий член функции ввода/вывода будет использоваться для обслуживания разблокирующих импульсов. Поэтому, если Вы пишете ваш собственный обработчик `io_open`, убедитесь, что установили все нужные области в *ocb* **перед тем**, как Вы вызовете функцию `resmgr_open_bind()`. В противном случае Ваша версия обработчика разблокирования функций ввода/вывода может получить вызов с неверными данными в *ocb*. (Заметим, что проблема разблокирующих импульсов "во время" обработки сообщения возникает, если только имеется несколько потоков, запущенных в Вашем менеджере ресурсов. Если запущен только один поток, то сообщения будут рассматриваться функциями `MsgReceive()` последовательно).

Итак, если клиент является SEND-блокированным, серверу не надо знать, что клиент прекратил запрос, поскольку сервер ещё и не получил его. Только в случае, когда сервер получил запрос и выполняет его обработку, ему надо знать, что клиент теперь уже желает прекращения и не ждет ответа.

Более полную информацию по этим состояниям и взаимодействиям см. в описании функций `MsgSend()`, `MsgReceive()`, `MsgReply()` и `ChannelCreate()` в "Справочнике библиотечных функций"; см. также главу "Связь между процессами" в книге "Архитектура системы".

Если Вы перекрываете разблокирующий обработчик по умолчанию, Вы всегда должны сначала вызвать обработчик по умолчанию для общего случая разблокирования. Например:

```
if((status = iofunc_unblock_default(...)) != _RESMGR_DEFAULT) {
    returns status;
}
/* Делайте Ваши собственные дела по разблокированию клиента */
```

Это гарантирует, что любой клиент, ожидающий в списке менеджера ресурсов (таком как справочный список блокирования) будет, если это возможно, разблокирован.

Обработка прерываний

Менеджер ресурсов, управляющий неким реальным аппаратным ресурсом, вероятно, будет вынужден обрабатывать прерывания, генерируемые аппаратными средствами. Детальное обсуждение стратегии для обработчиков прерываний см. в главе "Написание обработчика прерываний" в этой книге.

Как связаны обработчики прерываний с менеджерами ресурсов? Когда внутри обработчика прерываний происходит важное событие, обработчику нужно информировать об этом поток в менеджере ресурсов. Обычно это делается посредством импульса (обсуждено в разделе "Обработка частных сообщений и импульсов"), но это можно осуществить и через уведомительное событие `SIGEV_INTR`. Давайте рассмотрим это более детально.

Когда менеджер ресурсов стартует, он передаёт управление функции `thread_pool_start()`. Эта функция может вернуть управление, а может и нет, в зависимости от переданных ей флагов (если Вы не передавали никаких флагов, функция возвращает управление после создания пула потоков). Это означает, что если Вы переходите к установке некоего обработчика прерываний, Вы должны это сделать **до того, как** запущен пул потоков, или использовать одну из стратегий, обсуждавшихся выше (таких, как запуск потока для всего менеджера ресурсов в целом).

Однако если Вы пришли к использованию уведомительного события `SIGEV_INTR`, поток, который присоединял прерывание (через функции `InterruptAttach()` или `InterruptAttachEvent()`), должен быть тем же потоком, который вызвал `InterruptWait()`.

Пример кода обработчика прерываний

Вот пример, который включает важные блоки подпрограммы обслуживания прерываний и обрабатывающего потока:

```
#define INTNUM 0
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

void * interrupt_thread (void * data) {
    struct sigevent event;
    int             id;

    /* заполнение структуры event */
    memset(&event, 0, sizeof(event));
    event.sigev_notify = SIGEV_INTR;

    /* intNum - это желаемый уровень прерывания */
    id = InterruptAttachEvent (INTNUM, &event, 0);

    /*... вставьте сюда Ваш код ... */

    while (1) {
        InterruptWait (NULL, NULL);
        /* выполнение чего-то, касающегося прерывания,
         * возможно обновления каких-то структур общего доступа
         * в менеджере ресурсов
         *
         * размаскирование прерывания, когда готово
         */
        InterruptUnmask(INTNUM, id);
    }
}

int main(int argc, char **argv) {
    thread_pool_attr_t    pool_attr;
    resmgr_attr_t        resmgr_attr;
    dispatch_t           *dpp;
    thread_pool_t        *tpp;
    int                   id;

    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Невозможно разместить идентификатор диспетчеризации.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;
    /* Мы выполняем только присоединение resmgr-типе */
    pool_attr.context_alloc = resmgr_context_alloc;
    pool_attr.block_func = resmgr_block;
```

```

pool_attr.handler_func = resmgr_handler;
pool_attr.context_free = resmgr_context_free;
pool_attr.lo_water = 2;
pool_attr.hi_water = 4;
pool_attr.increment = 1;
pool_attr.maximum = 50;

if((tpp = thread_pool_create(&pool_attr, POOL_FLAG_EXIT_SELF)) == NULL) {
    fprintf(stderr, "%s: Невозможно инициализировать пул потоков.\n",
        argv[0]);
    return EXIT_FAILURE;
}

iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
                _RESMGR_IO_NFUNCS,          &io_funcs);
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

if((id = resmgr_attach(dpp, &resmgr_attr, "/dev/sample", FTYPE_ANY, 0,
                    &connect_funcs, &io_funcs, &attr)) == -1)
{
    fprintf(stderr, "%s: Невозможно присоединить имя.\n", argv[0]);
    return EXIT_FAILURE;
}

/* Запуск потока, который будет обрабатывать события прерывания. */
pthread_create (NULL, NULL, interrupt_thread, NULL);

/* Никогда не вернётся */
thread_pool_start(tpp);
}

```

Здесь функция *interrupt_thread()* использует *InterruptAttachEvent()*, чтобы связать источник прерывания (*intNum*) с событием (переданным в *event*), и затем ждёт наступления события.

Этот подход имеет преимущество в сравнении с использованием импульса. Импульс доставляется менеджеру ресурсов как сообщение. Это означает, что если потоки менеджера ресурсов, обрабатывающие сообщения, заняты обработкой запросов, импульс будет ожидать в очереди до тех пор, пока поток не выполнит *MsgReceive()*.

При использовании *InterruptWait()*, если поток, выполняющий *InterruptWait()*, имеет достаточно высокий приоритет, он разблокируется и запускается немедленно после того, как было сгенерировано SIGEV_INTR.

Многопоточный менеджер ресурсов

В этом разделе:

- Пример многопоточного менеджера ресурсов.
- Атрибуты пула потоков.

- Функции пула потоков.

Пример многопоточного менеджера ресурсов

Давайте рассмотрим более детально пример многопоточного менеджера ресурсов.

```
#include <errno.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>

/*
 * определяем THREAD_POOL_PARAM_T чтобы избежать предупреждения (варнинга)
 * компилятора, когда используем ниже функцию dispatch_*()
 */
#define THREAD_POOL_PARAM_T dispatch_context_t

#include <sys/iofunc.h>
#include <sys/dispatch.h>

static resmgr_connect_funcs_t    connect_funcs;
static resmgr_io_funcs_t        io_funcs;
static iofunc_attr_t            attr;

main(int argc, char **argv)
{
    /* объявляем переменные, которые будем использовать */
    thread_pool_attr_t    pool_attr;
    resmgr_attr_t        resmgr_attr;
    dispatch_t            *dpp;
    thread_pool_t        *tpp;
    dispatch_context_t    *ctp;
    int                    id;

    /* инициализируем интерфейс диспетчеризации */
    if((dpp = dispatch_create()) == NULL) {
        fprintf(stderr, "%s: Невозможно разместить идентификатор диспетчеризации.\n",
            argv[0]);
        return EXIT_FAILURE;
    }

    /* инициализируем атрибуты менеджера ресурсов */
    memset(&resmgr_attr, 0, sizeof resmgr_attr);
    resmgr_attr.nparts_max = 1;
    resmgr_attr.msg_max_size = 2048;

    /* инициализируем функции для обработки сообщений */
    iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
        _RESMGR_IO_NFUNCS, &io_funcs);

    /* инициализируем структуру атрибутов, используемую устройством */
    iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);

    /* присоединение нашего имени устройства */
    id = resmgr_attach(dpp,
        &resmgr_attr, /* обработчик диспетчеризации */
        "/dev/sample", /* атрибуты менеджера ресурсов */
        _FTYPE_ANY, /* имя устройства */
        0, /* тип открытия */
        0, /* флаги */
        &connect_funcs, /* подпрограммы связи */
        &io_funcs, /* подпрограммы ввода/вывода */
        &attr); /* идентификатор-описатель */

    if(id == -1) {
```

```

        fprintf(stderr, "%s: Невозможно присоединить имя.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Инициализируем атрибуты пула потоков */
    memset(&pool_attr, 0, sizeof pool_attr);
    pool_attr.handle = dpp;
    pool_attr.context_alloc = dispatch_context_alloc;
    pool_attr.block_func = dispatch_block;
    pool_attr.handler_func = dispatch_handler;
    pool_attr.context_free = dispatch_context_free;
    pool_attr.lo_water = 2;
    pool_attr.hi_water = 4;
    pool_attr.increment = 1;
    pool_attr.maximum = 50;

    /* Размещаем пул потоков */
    if((tpp = thread_pool_create(&pool_attr, POOL_FLAG_EXIT_SELF)) == NULL) {
        fprintf(stderr, "%s: Невозможно инициализировать пул потоков.\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Запуск потоков, без возврата */
    thread_pool_start(tpp);
}

```

Атрибут пула потоков (*pool_attr*) управляет различными аспектами пула потоков, такими как: какие функции вызываются, когда заканчивается, или стартует новый поток, общим числом рабочих потоков, минимальным числом, и т.д.

Атрибуты пула потоков

Вот структура *_thread_pool_attr*:

```

typedef struct _thread_pool_attr {
    THREAD_POOL_HANDLE_T *handle;
    THREAD_POOL_PARAM_T  (*block_func)(THREAD_POOL_PARAM_T *ctp);
    void                  (*unblock_func)(THREAD_POOL_PARAM_T *ctp);
    int                   (*handler_func)(THREAD_POOL_PARAM_T *ctp);
    THREAD_POOL_PARAM_T  (*context_alloc)(THREAD_POOL_HANDLE_T *handle);
    void                  (*context_free)(THREAD_POOL_PARAM_T *ctp);
    pthread_attr_t        *attr;
    unsigned short        lo_water;
    unsigned short        increment;
    unsigned short        hi_water;
    unsigned short        maximum;
    unsigned               reserved[8];
} thread_pool_attr_t;

```

Функции, которые Вы заполняете в приведенной структуре, могут быть взяты из уровня диспетчеризации (*dispatch_block()*, ...), уровня *resmgr* (*resmgr_block()*, ...) или Ваши собственные функции. Если Вы не используете функции уровня *resmgr*, то Вы должны определять *THREAD_POOL_PARAM*, как некий вид контекстной структуры для передачи между различными функциями библиотеки. По умолчанию она (структура) определена как *resmgr_context_t*, но поскольку этот пример использует уровень диспетчеризации, нам нужен тип *dispatch_context_t*. Мы и укажем его в самом начале, до заголовочных файлов, которые ссылаются на структуру *THREAD_POOL_PARAM_T*.

Часть этой структуры содержит информацию, указывающую библиотеке менеджера ресурсов, как Вы хотите обрабатывать множественные потоки (если таковые имеются). Во время разработки Вы должны проектировать Ваш менеджер ресурсов, держа в уме его

многопоточность. Но во время тестирования Вы, вероятнее всего, будете иметь только один исполняющийся поток (для упрощения отладки). Позже, после того как Вы убедитесь в стабильности функционирования Вашего менеджера ресурсов в основном, Вы можете "включить" несколько потоков и вновь пройти круг отладки.

Следующие поля структуры управляют количеством исполняющихся потоков:

lo_water

Минимальное количество одновременно блокированных потоков.

increment

Сколько потоков должно быть создано сразу, если количество блокированных потоков становится меньше *lo_water*.

hi_water

Максимальное количество одновременно блокированных потоков.

maximum

Максимальное общее число потоков, работающих одновременно.

Важными являются параметры, задающие максимальное количество потоков и величину приращения (*increment*). Значение *maximum* должно гарантировать, что всегда будет существовать поток в RECEIVE-блокированном состоянии. Если у Вас максимальное число потоков, то Ваши клиенты будут блокированы до тех пор, пока освободившийся поток не будет готов принимать данные. Значение, которое Вы задаёте в *increment*, будет сокращать число "раз", необходимых Вашему драйверу, чтобы создавать потоки. Видимо, лучше ошибиться в большую сторону – в сторону создания большего количества потоков и оставить их на подхвате, нежели чем всё время заниматься их созданием/уничтожением.

Вы определяете количество потоков, которые будут RECEIVE-блокированными на вызове *MsgReceive()* в любой момент времени, заполняя параметр *lo_water*. Если в какой-то момент RECEIVE-блокированных потоков стало меньше, чем *lo_water*, параметр *increment* задаёт, сколько потоков будет создано за один раз, так что, по крайней мере, *lo_water* число потоков будут снова RECEIVE-блокированными.

Выполнив свою работу, потоки снова дойдут до блокирующей функции. Значение *hi_water* задаёт верхний предел количества RECEIVE-блокированных потоков. Как только этот предел будет достигнут, потоки будут самоуничтожаться, гарантируя, что RECEIVE-блокированными будет не более *hi_water* членов пула потоков.

Чтобы предотвратить неограниченный рост числа потоков, параметр *maximum* задает абсолютный максимум количества потоков, которые могут исполняться одновременно.

Когда потоки создаются библиотекой менеджера ресурсов, они будут иметь токой размер стека, каким он задан в параметре *thread_stack_size*. Если Вы хотите задать размер стека или приоритет, заполните в *pool_attr.attr* соответствующий *pthread_attr_t* указатель.

Структура *thread_pool_attr_t* содержит указатели на несколько функций:

block_func()

Вызывается работающим потоком, когда ему необходимо заблокироваться в ожидании некоего сообщения.

handler_func()

Вызывается потоком, когда он разблокируется по причине получения сообщения. Эта функция обрабатывает сообщение.

context_alloc()

Вызывается, когда создаётся новый поток. Возвращает контекст, который этот поток использует при своей работе

context_free()

Освобождает контекст, когда работающий поток завершается.

unlock_func()

Вызывается библиотекой, чтобы закрыть пул потоков или изменить количество исполняющихся потоков.

Функции пула потоков

Библиотека предоставляет следующие функции пула потоков:

thread_pool_create()

Инициализирует контекст пула. Возвращает указатель на управляющую структуру пула потоков (*tpp*), который используется для запуска пула потоков.

thread_pool_start()

Запускает пул потоков. Эта функция может вернуть управление, а может и не вернуть, в зависимости от флагов, передаваемых *thread_pool_create()*.

thread_pool_destroy()

Уничтожает пул потоков.

thread_pool_control()

Управляет числом потоков.

ⓘ В примере, приведенном в разделе "Многопоточные менеджеры ресурсов", *thread_pool_start(tpp)* никогда не вернёт управление, поскольку мы установили бит *POOL_FLAG_EXIT_SELF*. Флаг *POOL_FLAG_USE_SELF* сам по себе никогда не вернёт управления, но зато текущий поток станет частью пула потоков.

Если никакие флаги не переданы (т.е. 0 вместо каких-либо флагов), функция вернёт управление после того, как пул потоков будет создан.

Менеджер ресурсов файловой системы

В этом разделе:

- Обсуждение менеджеров ресурсов файловой системы.
- Захват более чем одного устройства.
- Обработка директорий.

Обсуждение менеджеров ресурсов файловой системы

Поскольку менеджер ресурсов файловой системы в принципе может получать длинные имена путей, он должен верно производить анализ имени пути и обрабатывать каждый компонент. Предположим, что менеджер ресурсов регистрирует точку монтирования */mount/*, и пользователь ввел команду:

```
ls -l /mount/home
```

где */mount/home* является директорией в устройстве.

Утилита *ls* выполнит следующее:

```
d = opendir("/mount/home");
while(...) {
```

```

        dirent = readdir(d);
        ....
    }

```

Захват более чем одного устройства

Если мы хотим, чтобы наш менеджер ресурсов обслуживал несколько устройств, необходимо всего несколько простых изменений. Мы должны вызвать *resmgr_attach()* для каждого имени устройства, которое мы хотим зарегистрировать. Мы также должны передать структуру атрибутов, которая является уникальной для каждого регистрируемого устройства, чтобы функции типа *chmod()* могли бы модифицировать атрибуты, связанные с правильно определённым ресурсом.

Вот модификации, необходимые для обработки и */dev/sample1* и */dev/sample2*:

```

/*
 * Модификация [1]: Размещаем множество структур атрибутов,
 * и заполняем массив имён (для удобства)
 */

#define NumDevices 2
iofunc_attr_t    sample_attrs [NumDevices];
char             *names [NumDevices] =
{
    "/dev/sample1",
    "/dev/sample2"
};

main ()
{
    ...
    /*
     * Модификация [2]: заполняем структуру атрибутов для каждого устройства
     * и вызываем resmgr_attach для каждого устройства
     */
    for (i = 0; i < NumDevices; i++) {
        iofunc_attr_init (&sample_attrs [i], S_IFCHR | 0666, NULL, NULL);
        pathID = resmgr_attach (dpp, &resmgr_attr, name[i],
                                _FTYPE_ANY, 0,
                                &my_connect_funcs,
                                &my_io_funcs,
                                &sample_attrs [i]);
    }
    ...
}

```

Первая модификация просто объявляет массив структур, так что каждое устройство имеет свою собственную структуру атрибутов. Для удобства, мы также объявляем массив имён, что упрощает передачу имени устройства в цикл *for()*. Некоторые менеджеры ресурсов (такие как *devc-ser8250*) конструируют имена устройств «на лету», или получают их из командной строки.

Вторая модификация инициализирует массив структур атрибутов и затем вызывает *resmgr_attach()* для каждого устройства, передавая уникальное имя и уникальную структуру атрибутов.

Вот и все, что требуется изменить. В наших функциях *io_read()* или *io_write()* ничего менять не надо - функции уровня *iofunc* по умолчанию элегантно обработают несколько устройств.

Обработка директорий

До этого наше обсуждение было сосредоточено на менеджере ресурсов, который подсоединял каждое имя устройства через отдельные вызовы функции *resmgr_attach()*. Мы показали, как "захватить" одиночное имя пути. (Наши примеры использовали имена пути под префиксом */dev*, но ничто не мешает Вам взять любые другие имена путей, например */MyDevice*).

Типичный менеджер ресурсов может зарегистрировать любое количество имен путей. Однако существует практический предел, где-то порядка нескольких сотен – реальный предел зависит от размера памяти и скорости цикла поиска в администраторе процессов.

Что, если Вы хотите объять тысячи или даже миллионы имён путей?

Наиболее простым методом, как это сделать, является захват префикса имени пути (*pathname prefix*) и управление структурой директорий ниже этого префикса (или точки монтирования).

Вот некоторые примеры менеджеров ресурсов, которые могут делать это:

- Файловая система CD-ROM может захватить префикс имени пути */cdrom*, и затем обрабатывать любые запросы на файлы ниже этого имени пути, чтобы перейти на CD-ROM-устройство.
- Файловая система для управления архивными (сжатыми) файлами может захватить префикс */uncompressed*, и затем распаковывать на лету файлы, когда приходят запросы на чтение.
- Сетевая файловая система может представлять структуру директорий удалённой машины, называемой "flipper", под префиксом имени пути */mount/flipper*, и позволять пользователю получить доступ к файлам flipper'a, как будто они располагаются на локальной машине.

И это только наиболее очевидные примеры. Резоны (и возможности) почти безграничны.

Общей характеристикой всех этих менеджеров ресурсов является то, что все они обеспечивают работу файловых систем. Менеджер ресурсов файловой системы отличается от менеджера ресурсов "устройств" (которые мы рассматривали до сих пор) в следующих ключевых областях:

1. Флаг *_RESMGR_FLAG_DIR* в *resmgr_attach()* информирует библиотеку, что менеджер ресурсов будет позволять совпадения имени пути с заданной точкой монтирования или ниже её.
2. Логика *_IO_CONNECT* проверяет отдельные компоненты имени пути на разрешения и авторизацию доступа. Она также должна гарантировать, что когда получаем доступ к конкретному имени файла, присоединяются правильные атрибуты.
3. Логика *_IO_READ* возвращает данные либо для "файла", либо для "директории" в имени пути.

Давайте по очереди рассмотрим эти пункты.

Совпадение с точкой монтирования или ниже её

Когда мы задавали аргумент *flags* для функции *resmgr_attach()* в нашем примере менеджера ресурсов, мы задавали 0, подразумевая, что будут использоваться библиотечные функции по умолчанию. Если мы задали значение *_RESMGR_FLAGS_DIR* отличным от 0, библиотечные функции позволят разрешение имени, начиная с заданной точки монтирования или ниже.

Сообщение *_IO_OPEN* для файловых систем

Как только мы задали точку монтирования, она должна стать известна менеджеру ресурсов, чтобы определить соответствующий ответ на запрос открытия. Давайте допустим, что мы определили точку монтирования `/sample_fsys` для нашего менеджера ресурсов:

```
pathID = resmgr_attach("/sample_fsys", /* точка подмонтирования */
    dpp,
    &resmgr_attr,
    _FTYPE_ANY,
    _RESMGR_FLAG_DIR, /* это директория */
    &connect_funcs,
    &io_funcs,
    &attr);
```

Теперь, когда клиент выполняет вызов, подобный такому:

```
fopen("/sample_fsys/spud", "r");
```

мы получаем сообщение `_IO_CONNECT`, и будет вызван наш обработчик `io_open`. Поскольку мы ещё не рассматривали подробно сообщение `_IO_CONNECT`, давайте рассмотрим его сейчас:

```
struct _io_connect {
    unsigned short type;
    unsigned short subtype; /* _IO_CONNECT_* */
    unsigned long file_type; /* _FTYPE_* в файле sys/ftype.h */
    unsigned short reply_max;
    unsigned short entry_max;
    unsigned long key;
    unsigned long handle;
    unsigned long ioflag; /* O_* в файле fcntl.h, _IO_FLAG_* */
    unsigned long mode; /* S_IF* в файле sys/stat.h */
    unsigned short sflag; /* SH_* в файле share.h */
    unsigned short access; /* S_I в файле sys/stat.h */
    unsigned short zero;
    unsigned short path_len;
    unsigned char eflag; /* _IO_CONNECT_EFLAG_* */
    unsigned char extra_type; /* _IO_EXTRA_* */
    unsigned short extra_len;
    unsigned char path[1]; /* path_len, null, extra_len */
};
```

Рассматривая значимые поля, мы видим `ioflag`, `mode`, `sflag` и `access`, говорящие нам, как был открыт ресурс.

Параметр `path_len` указывает, сколько байт содержится в имени пути. Действительное имя пути появляется в параметре `path`. Заметьте, что появившееся имя пути не `/sample_fsys/spud`, как Вы могли подумать, а просто `spud` – сообщение содержит только имя пути относительно точки монтирования менеджера ресурсов. Это упрощает написание кода, поскольку Вам не надо каждый раз перескакивать имя точки монтирования, коду вообще не известно, что есть точка монтирования, и сообщение будет немножко короче.

Заметим также, что имя пути никогда не будет иметь относительных (`.` и `..`) компонент пути, лишних слэшей в нём (например, `spud//stuff`) – всё это анализируется и удаляется, когда сообщение посылается менеджеру ресурсов.

При написании менеджеров ресурсов файловых систем, мы сталкиваемся с дополнительными трудностями, когда имеем дело с именами путей. Для проверки прав доступа нам необходимо разбить на части переданное имя пути и проверить каждый компонент. Чтобы разбить строку на части, Вы можете использовать функцию `strtok()` и родственные ей, и затем функцию `iofunc_check_access()`, удобные вызовы уровня `iofunc`, которые осуществляют проверку прав доступа для подготовленных компонент имени пути. (См. "Справочник библиотечных

функций", функцию `iofunc_open()`, чтобы получить детальную информацию о шагах, необходимых для этого уровня проверки).

i Ограничение, которое возникает после проверки имени, требует, чтобы каждый обрабатываемый путь имел собственную структуру атрибутов, передаваемую функции `iofunc_open_default()`. Если к имени пути будут привязаны несоответствующие атрибуты (которые это имя обеспечивают), результатом станет непредсказуемое поведение.

Возврат входов директорий из `_IO_READ`

Когда вызывается обработчик `_IO_READ`, ему может понадобиться вернуть данные о файле (если `S_ISDIR (ocb->attr->mode)` есть ложь) или о директории (если `S_ISDIR(ocb->attr->mode)` есть истина). Мы уже рассматривали алгоритм возврата данных, особенно метод согласования размера возвращаемого блока данных с меньшим доступным блоком или размером буфера клиента.

Сходное ограничение имеет место при возвращении клиенту данных о директории. Кроме того, мы имеем дополнительную проблему возвращения цельно-блочных данных (*block-integral data*). Это означает, что вместо того, чтобы возвращать поток (в смысле *stream*) байтов, где мы можем произвольно размещать данные, мы в действительности будем возвращать некоторое количество структур типа `struct dirent`. (Другими словами, мы не можем вернуть полторы таких структуры, мы всегда возвращаем целое число).

Структура `dirent` выглядит подобным образом:

```
struct dirent {
    ino_t          d_ino;
    off_t          d_offset;
    unsigned short d_reclen;
    unsigned short d_namelen;
    char           d_name[NAME_MAX + 1];
};
```

Поле `d_ino` содержит уникальный для точки монтирования порядковый номер файла (*mountpoint-unique file serial number*). Этот порядковый номер часто используется различными утилитами проверки диска для таких операций, как определение бесконечно закольцованных линков директорий. (Заметим, что значение *inode* не может быть равным нулю, это указывает, что представляемый *inode* является неиспользуемым входом).

Поле `d_offset` обычно используется для идентификации собственно элемента каталога (точки входа в директорию). Для дисковых файловых систем это значение может быть реальным смещением в дисковой структуре директорий.

Другие реализации могут назначать индексный номер элемента каталога (0 для первого входа в этой директории, 1 для следующего, и так далее). Единственным ограничением является то, что используемые схемы нумерации должны быть согласованы между обработчиком сообщений `_IO_LSEEK` и обработчиком сообщений `_IO_READ`.

Допустим, `d_offset` представляет индексный номер элемента каталога. Значит, если сообщение `_IO_LSEEK` приводит к тому, что текущее смещение стало равным 7, и затем пришёл запрос `_IO_READ`, Вы должны вернуть информацию о директории, начиная с элемента каталога номер 7.

Поле *d_reclen* содержит размер этого элемента каталога и любую другую присоединённую информацию (такую как необязательная структура *struct stat*, добавляемая к содержимому *struct dirent*; см. ниже).

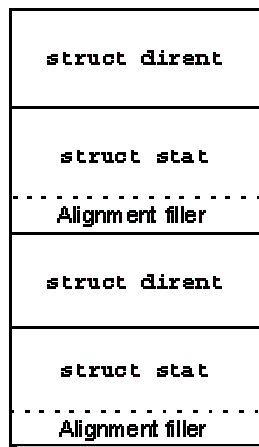
Параметр *d_namelen* указывает размер параметра *d_name*, хранящего действительное имя этого элемента каталога. (Поскольку размер вычисляется с использованием функции *strlen()*, терминатор строки (`\0`), который обязан присутствовать, не учитывается.)

Таким образом, в нашем обработчике *io_read*, нам необходимо генерировать некоторое количество элементов *struct dirent* и возвращать их клиенту. Если у нас есть кэш элементов каталогов, который мы поддерживаем в нашем менеджере ресурсов, самым простым будет построить набор векторов ввода/вывода (IOV) для указаний на эти элементы. Если у нас нет кэша, то мы должны вручную собрать элементы каталогов в буфер, и затем вернуть вектор ввода/вывода, указывающий на этот буфер.

Возврат информации, связанной со структурой директории

Вместо того, чтобы возвращать в сообщении `_IO_READ` просто *struct dirent*, Вы можете также возвращать структуру *struct stat*. Хотя это и улучшит эффективность, возвращение *struct stat* не является обязательным. Если Вы её не возвращаете, пользователи Вашего устройства могут просто вызвать функцию *stat()*, чтобы получить эту информацию. (Это вопрос использования. Если Ваше устройство обычно используется так, что вызывается функция *readdir()*, а затем вызывается *stat()*, будет более эффективно возвращать обе структуры. Для получения более полной информации см. описание функции *readdir()* в "Справочнике библиотечных функций").

Информация дополнительной структуры *struct stat* возвращается после каждого элемента каталога:



Возвращение необязательной "struct stat" вместе со входом "struct dirent" может улучшить эффективность.

i Структура *struct stat* должна быть выровнена по 8-битовой границе. Поле *d_reclen* структуры *struct dirent* должно содержать размер **обеих** структур, включая какой-либо заполнитель, требующийся для выравнивания.

Типы сообщений

Менеджер ресурсов получает два основных типа сообщений:

- сообщения связи
- сообщения ввода/вывода.

Сообщения связи

Сообщение связи посылается клиентом, при выполнении операций, связанных с именами путей. Это может быть сообщение, устанавливающее относительно долгое взаимодействие между клиентом и менеджером ресурсов (например, *open()*), или сообщение, являющееся "одноразовым" событием (например, *rename()*). Библиотека просматривает параметр *connect_funcs* (типа *resmgr_connect_funcs_t*) и вызывает соответствующую функцию.

Если сообщение является сообщением `_IO_CONNECT` (и его вариантами), соответствующее вызову *open()*, то должен быть установлен контекст для последующих сообщений ввода/вывода, которые будут обработаны позднее. На этот контекст ссылаются как на *ocb* – он хранит любую информацию, требуемую между сообщением связи и последующим сообщением ввода/вывода.

Вообще, *ocb* является хорошим местом для хранения информации, которую надо сохранять для каждого открытия (*per-open*). Примером этого может быть текущая позиция внутри файла. Каждый открытый дескриптор файла должен иметь свою собственную позицию в файле. В этом случае на каждое открытие файла размещается новый *ocb*. Во время обработки открытия Вы выполняете инициализацию позиции в файле. При обработке чтения или записи Вы изменяете позицию в файле. Для получения более полной информации см. раздел "Структура блока открытого контекста (*ocb*)".

Сообщения ввода/вывода

Сообщения ввода/вывода – сообщения, зависящие от существующей привязки (например, *ocb*) между клиентом и менеджером ресурсов.

Например, сообщение `_IO_READ` (из функции *read()* клиента) зависит от предварительно установленного клиентом соединения (или контекста) с менеджером ресурсов, осуществлённого вызовом функции *open()* и получения обратно файлового дескриптора. Этот контекст (созданный вызовом функции *open()*) затем используется для выполнения последующих сообщений ввода/вывода, таких как `_IO_READ`.

Для этого есть хорошее основание. К примеру, было бы неэффективно передавать полное имя пути для каждого запроса *read()*. Обработчик *open()* может также выполнять задачи, которые мы хотим исполнить только один раз (например, проверка прав доступа), а не в каждом сообщении ввода/вывода. К тому же, когда функция *read()* прочитает 4096 байтов из файла, ещё могут оставаться 20 мегабайт, ожидающих прочтения. Поэтому функция *read()* потребует некоторой контекстной информации, указывающей ей позицию внутри файла, из которого она читает, сколько было прочитано, и всё такое прочее.

Структура *resmgr_io_funcs_t* заполняется способом, схожим со структурой функций связи *resmgr_connect_funcs_t*. Отметим, что все функции ввода/вывода имеют общий список параметров. Первым параметром является структура контекста менеджера ресурсов, вторым –

сообщение (тип которого соответствует обработанному сообщению и содержит параметры, посланные клиентом), и последним является *oscb* (содержащий то, что мы закрепили, когда обрабатывали функцию клиента *open()*).

Структуры данных менеджера ресурсов

При написании менеджера ресурсов Вам понадобится понимать структуры данных, используемых для управления работой библиотеки:

- структура управления `_resmgr_attr_t`
- таблица связей `resmgr_connect_funcs_t`
- таблица векторов ввода/вывода `resmgr_io_funcs_t`.

Структура управления `_resmgr_attr_t`

Структура управления `_resmgr_attr_t` состоит по меньшей мере из следующих элементов:

```
typedef struct _resmgr_attr {
    unsigned    flags;
    unsigned    nparts_max;
    unsigned    msg_max_size;
    int         (*other_func)(resmgr_context_t *, void *msg);
    unsigned    reserved[4];
} resmgr_attr_t;
```

nparts_max

Число компонентов, которые должны быть размещены в массиве векторов (IOV) ввода/вывода

msg_max_size

Размер буфера сообщения

Эти поля будут важны, когда Вы начнёте писать свои функции-обработчики. Если Вы задали значение 0 для *nparts_max*, библиотека менеджера ресурсов будет сводить значения к минимальному, используемому самой библиотекой. Зачем Вам может понадобиться устанавливать размер массива векторов ввода/вывода? Как мы видели в разделе "Получение выполнения ответа библиотекой менеджера ресурсов", Вы можете указать библиотеке менеджера ресурсов выполнять отклик за нас, самостоятельно. Может понадобиться передать ей некий массив векторов ввода/вывода, указывающий на N буферов, содержащих данные ответа. Но, поскольку мы попросили библиотеку выполнить ответ за нас, нам надо использовать ее массив векторов ввода/вывода, который должен быть достаточно велик, чтобы вместить указатели на наши N буферов.

flags

Позволяет Вам изменить поведение интерфейса менеджера ресурсов.

other_func

Позволяет Вам задать подпрограмму, вызываемую в случаях, когда менеджер ресурсов получает сообщение ввода/вывода, которое он не понимает. (Как правило, мы не рекомендуем Вам использовать это поле. Более подробно см. следующий раздел). Чтобы подсоединить *other_func*, Вы должны установить флаг `RESMGR_FLAG_ATTACH_OTHERFUNC`

Если библиотека менеджера ресурсов получает сообщение ввода/вывода, которое неизвестно как обрабатывать, она вызовет подпрограмму, заданную полем *other_func*, если он не NULL. (Если он NULL, то библиотека менеджера ресурсов вернет клиенту ENOSYS, чётко устанавливая, что она не знает, что сие сообщение означает). Вы можете задавать не-NULL значение *other_func* в случае, когда Вы задаёте некоторую форму особых сообщений между клиентами и Вашим менеджером ресурсов. Хотя рекомендуемым решением является вызов клиентом функции *devctl()* и со стороны сервера обработчик сообщений *_IO_DEVCTL*. Или вызов клиентом функций семейства *MsgSend*()* и со стороны сервера обработчик сообщений *_IO_MSG*.

Для сообщений, которые не являются сообщениями ввода/вывода, Вы должны использовать функцию *message_attach()*, которая подсоединяет диапазон сообщений для диспетчирующего обработчика. Когда будет получено сообщение типа, входящего в этот диапазон, функция *dispatch_block()* вызывает предоставляемую пользователем функцию, которая выполняет какие-то специальные действия, такие как отклик клиенту.

Таблица связей *resmgr_connect_funcs_t*

Вот определение структуры таблицы связей *resmgr_connect_funcs_t*:

```
typedef struct _resmgr_connect_funcs {
    unsigned nfuncs;
    int (*open)      (resmgr_context_t *ctp, io_open_t *msg,
                    RESMGR_HANDLE_T *handle, void *extra);
    int (*unlink)   (resmgr_context_t *ctp, io_unlink_t *msg,
                    RESMGR_HANDLE_T *handle, void *reserved);
    int (*rename)   (resmgr_context_t *ctp, io_rename_t *msg,
                    RESMGR_HANDLE_T *handle,
                    io_rename_extra_t *extra);
    int (*mknod)    (resmgr_context_t *ctp, io_mknod_t *msg,
                    RESMGR_HANDLE_T *handle, void *reserved);
    int (*readlink) (resmgr_context_t *ctp, io_readlink_t *msg,
                    RESMGR_HANDLE_T *handle, void *reserved);
    int (*link)     (resmgr_context_t *ctp, io_link_t *msg,
                    RESMGR_HANDLE_T *handle,
                    io_link_extra_t *extra);
    int (*unblock)  (resmgr_context_t *ctp, io_pulse_t *msg,
                    RESMGR_HANDLE_T *handle, void *reserved);
    int (*mount)    (resmgr_context_t *ctp, io_mount_t *msg,
                    RESMGR_HANDLE_T *handle,
                    io_mount_extra_t *extra);
} resmgr_connect_funcs_t;
```

Таблица векторов ввода/вывода *resmgr_io_funcs_t*

Структура управления *resmgr_io_funcs_t* содержит, по меньшей мере, следующие компоненты:

```
typedef struct _resmgr_io_funcs {
    unsigned nfuncs;
    int (*read)     (resmgr_context_t *ctp, io_read_t *msg,
                    RESMGR_OCB_T *ocb);
    int (*write)    (resmgr_context_t *ctp, io_write_t *msg,
                    RESMGR_OCB_T *ocb);
    int (*close_ocb) (resmgr_context_t *ctp, void *reserved,
                    RESMGR_OCB_T *ocb);
}
```

```

int (*stat)      (resmgr_context_t *ctp, io_stat_t *msg,
                 RESMGR_OCB_T *ocb);
int (*notify)   (resmgr_context_t *ctp, io_notify_t *msg,
                 RESMGR_OCB_T *ocb);
int (*devctl)   (resmgr_context_t *ctp, io_devctl_t *msg,
                 RESMGR_OCB_T *ocb);
int (*unblock)  (resmgr_context_t *ctp, io_pulse_t *msg,
                 RESMGR_OCB_T *ocb);
int (*pathconf) (resmgr_context_t *ctp, io_pathconf_t *msg,
                 RESMGR_OCB_T *ocb);
int (*lseek)    (resmgr_context_t *ctp, io_lseek_t *msg,
                 RESMGR_OCB_T *ocb);
int (*chmod)    (resmgr_context_t *ctp, io_chmod_t *msg,
                 RESMGR_OCB_T *ocb);
int (*chown)    (resmgr_context_t *ctp, io_chown_t *msg,
                 RESMGR_OCB_T *ocb);
int (*utime)    (resmgr_context_t *ctp, io_utime_t *msg,
                 RESMGR_OCB_T *ocb);
int (*fdopen)   (resmgr_context_t *ctp, io_fdopen_t *msg,
                 RESMGR_OCB_T *ocb);
int (*fdinfo)   (resmgr_context_t *ctp, io_fdinfo_t *msg,
                 RESMGR_OCB_T *ocb);
int (*lock)     (resmgr_context_t *ctp, io_lock_t *msg,
                 RESMGR_OCB_T *ocb);
int (*space)    (resmgr_context_t *ctp, io_space_t *msg,
                 RESMGR_OCB_T *ocb);
int (*shutdown) (resmgr_context_t *ctp, io_shutdown_t *msg,
                 RESMGR_OCB_T *ocb);
int (*mmap)     (resmgr_context_t *ctp, io_mmap_t *msg,
                 RESMGR_OCB_T *ocb);
int (*msg)      (resmgr_context_t *ctp, io_msg_t *msg,
                 RESMGR_OCB_T *ocb);
int (*umount)   (resmgr_context_t *ctp, void *msg,
                 RESMGR_OCB_T *ocb);
int (*dup)      (resmgr_context_t *ctp, io_dup_t *msg,
                 RESMGR_OCB_T *ocb);
int (*close_dup) (resmgr_context_t *ctp, io_close_t *msg,
                 RESMGR_OCB_T *ocb);
int (*lock_ocb) (resmgr_context_t *ctp, void *reserved,
                 RESMGR_OCB_T *ocb);
int (*unlock_ocb) (resmgr_context_t *ctp, void *reserved,
                  RESMGR_OCB_T *ocb);
} resmgr_io_funcs_t;

```