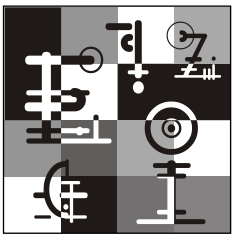
ГЛАВА 3



Часы, таймеры и периодические уведомления

Часы и таймеры

Пришло время рассмотреть все, что относится ко времени в Neutrino. Мы увидим, как и почему мы должны использовать таймеры, а также рассмотрим теоретические положения, которые этому сопутствуют. Далее мы обсудим способы опроса и настройки часов реального времени.

Давайте рассмотрим типовую техническую систему — скажем, автомобиль. В этом автомобиле у нас есть ряд программ, большинство из которых выполняются с различными приоритетами. Некоторые из этих программ необходимы для обеспечения реакции на внешние события (например, тормоза или радиоприемник), другие же должны срабатывать периодически (например, система диагностики).

Периодические процессы

Так как же обеспечивается "периодическая" работа системы диагностики? Можно вообразить себе некоторый процесс, выполняемый процессором нашего автомобиля и делающий нечто подобное следующему:

```
// Процесс диагностики
int main(void)      // Игнорируем аргументы
{
    for (;;) {
        perform_diagnostics();
        sleep(15);
    }

    // Сюда мы не дойдем
    return(EXIT_SUCCESS);
}
```

Видно, что процесс диагностики выполняется бесконечно. Он запускает цикл работ по диагностике, затем "засыпает" на 15 секунд, потом "просыпается", и все повторяется заново.

Если мысленно вернуться назад в мрачные и смутные однозадачные времена, когда один процессор обслуживал одного пользователя, то можно припомнить, что программы такого сорта реализовывались путем выполнения функцией *sleep()* активного ожидания. Для этого вам было необходимо узнать быстродействие вашего процессора и написать собственную функцию *sleep()*, например:

```
void sleep(int nseconds)
{
    long  i;

    while (nseconds--) {
        for (i = 0; i < CALIBRATED_VALUE; i++) ;
    }
}
```

В те дни, поскольку в машине не выполнялись никакие другие задачи, такие программы не составляли большой проблемы, т. к. никакой другой процесс не беспокоило, что вы используете своей функцией *sleep()* все 100% ресурсов процессора.

Внимание

Даже в наши дни мы иногда отдаем все 100% ресурсов процессора, чтобы отмерить время. В частности, функция *nanospin()* применяется для отсчета времени с очень большой точностью, но делает это за счет монопольного захвата процессора на своем приоритете. Пользуйтесь с осторожностью!

Если вы должны были реализовать некоторое подобие "многозадачного режима", то это обычно делалось путем применения процедуры прерывания, которая либо срабатывала от аппаратного таймера, либо выполнялась в пределах периода "активного ожидания", оказывая при этом некоторое воздействие на калибровку отсчета времени. Это обычно не вызывало беспокойство.

К счастью, в решении этих проблем мы уже ушли далеко вперед. Вспомните *разд. "Диспетчеризация и реальный мир" главы 1*, там описываются причины, по которым ядро выполняет перепланирование потоков. Причины могут быть следующие:

- аппаратное прерывание;
- системный вызов;
- сбой (исключение).

В данной главе мы подробно проанализируем две первые причины из вышеуказанного списка — аппаратные прерывания и системные вызовы.

Когда поток вызывает функцию *sleep()*, код, содержащийся в Си-библиотеке, в конечном счете делает системный вызов. Этот вызов приказывает ядру отложить вы-

полнение данного потока на заданный интервал времени. Ядро удаляет поток из рабочей очереди и включает таймер.

Все это время ядро принимает регулярно поступающие аппаратные прерывания таймера. Положим для определенности, что эти аппаратные прерывания происходят ровно каждые 10 мс.

Давайте немного переформулируем это утверждение: каждый раз, когда такое прерывание обслуживается соответствующей подпрограммой обработки прерывания (ISR) ядра, это значит, что истек очередной 10-миллисекундный интервал. Ядро отслеживает время суток путем увеличения специальной внутренней переменной на значение, соответствующее 10 мс, с каждым вызовом обработчика прерывания.

Так что, реализуя 15-секундный таймер, ядро в действительности выполняет следующее:

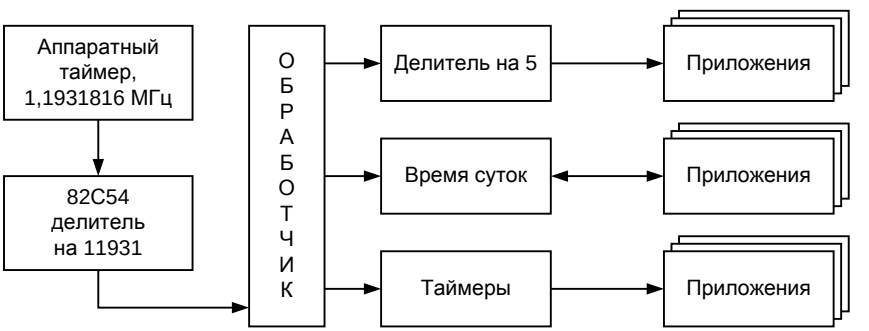
1. Устанавливает переменную в текущее время плюс 15 секунд.
2. В обработчике прерываний сравнивает эту переменную с текущим временем.
3. Когда текущее время станет равным (или больше) данной переменной, поток снова ставится в очередь готовности.

При использовании множества параллельно работающих таймеров — например, когда необходимо активизировать несколько потоков в различные моменты времени, — ядро просто ставит запросы в очередь, отсортировав таймеры по возрастанию их времени истечения (в голове очереди при этом окажется таймер с минимальным временем истечения). Обработчик прерывания будет анализировать только переменную, расположенную в голове очереди.

Источники прерывания таймера

На этом мы, пожалуй, закончим наш краткий экскурс по стране таймеров и перейдем к вещам, которые уже не так очевидны.

Откуда возникают прерывания таймера? На рисунке далее приведены аппаратные компоненты (и некоторые характерные для PC значения параметров), отвечающие за генерацию этих прерываний.



Источники прерываний таймера в PC

Из рисунка видно, что в PC используется высокочастотный аппаратный генератор синхроимпульсов (мегагерцового диапазона). Высокочастотный меандр делится при помощи аппаратного счетчика (на рисунке — микросхема Intel 82C54), который понижает частоту импульсов до сотен килогерц или сотен герц (диапазон, в котором их уже может обработать ISR). ISR таймера входит в состав ядра и взаимодействует непосредственно с его кодом и внутренними структурами данных. В процессорах архитектуры не-x86 (MIPS, PowerPC) тоже происходит подобная последовательность событий; в некоторых микросхемах аппаратный таймер может быть непосредственно встроен в процессор.

Отметим, что импульсы высокой частоты делятся на целочисленный делитель. Это означает, что результирующий период импульсов не будет точно равен 10 мс, потому что исходный период не кратен 10 мс. Поэтому ISR ядра из вышеприведенного примера будет реально вызываться по истечении каждых 9,9999296004 мс.

Большое дело, скажете вы, ну и что? Ну ладно, для нашего 15-секундного счетчика это годится. 15 секунд — это 1500 отсчетов таймера; расчеты показывают, что погрешность будет в районе 106 мкс:

$$15 \text{ с} - 1500 \times 9{,}9999296004 \text{ мс} = 15000 \text{ мс} - 14999{,}8944006 \text{ мс} =$$
$$= 0{,}1055994 \text{ мс} = 105{,}5994 \text{ мкс}.$$

К сожалению, продолжая наши математические выкладки, приходим к выводу, что при таких раскладах погрешность составляет 608 мс в день, что равняется приблизительно 18,5 секунд в месяц, или почти 3,7 минут в год!

Можно предположить, что при использовании делителей другого типа ошибка может быть либо меньше, либо больше, в зависимости от погрешности округления. К счастью, ядро это знает и вводит соответствующие поправки.

Ключевой момент всей этой истории состоит в том, что независимо от красивого округленного значения, реальное значение выбирается в сторону ускорения отсчета.

Разрешающая способность отсчета времени

Пусть отсчеты времени таймера генерируются чуть чаще, чем раз в 10 мс. Смогу ли я надежно обеспечить ожидание длительностью в 3 мс?

Не-а.

Подумайте, что происходит в ядре. Мы вызываем стандартную библиотечную функцию *delay()* для задержки на 3 мс. Ядро должно присвоить внутренней переменной ISR какое-то значение. Если оно присвоит ей значение текущего времени, то это будет означать, что таймер уже истек, и надо активизироваться немедленно. Если оно присвоит ей значение на один отсчет больше текущего времени, это будет означать, что надо активизироваться на следующем отсчете (т. е. с задержкой *вплоть* до 10 мс).

Мораль: не следует рассчитывать на то, что разрешающая способность ваших таймеров будет лучше, чем у системного отсчета.

Достижение более высокой точности

В Neutrino у приложений есть возможность программной подстройки аппаратного делителя и ядра вместе с ним (чтобы ядро знало, с какой частотой вызывается ISR таймера). Мы поговорим об этом в *разд. "Опрос и установка часов реального времени и кое-что еще" далее в этой главе*.

Флуктуации отсчета времени

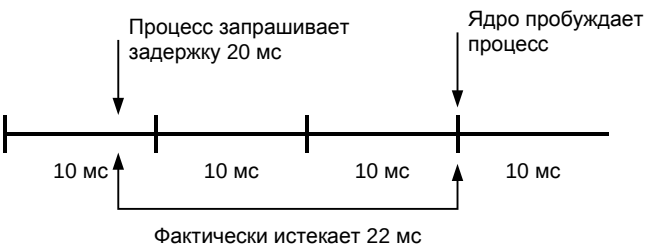
Существует еще одно явление, которое вы должны принимать во внимание. Предположим, что разрешающая способность у вас равна 10 мс, а вы желаете сформировать задержку длительностью в 20 мс.

Всегда ли вы можете быть уверены, что от момента вызова функции *delay()* до возврата из нее пройдет ровно 20 мс?

Никогда.

На это есть две серьезные причины. Первая причина довольно проста: при блокировании поток изымается из очереди готовности. Это означает, что процессор может перейти к другому потоку вашего приоритета. Когда ваши 20 мс истекут, ваш поток будет помещен в *конец* очереди готовности по этому приоритету и будет таким образом оставлен на милость потока, выполняющегося в данный момент. Это относится также к обработчикам прерываний и к потокам более высокого приоритета — то, что ваш поток перешел в состояние READY, еще не означает, что ему сразу предоставят процессор.

Вторая причина несколько более хитрая. Чтобы понять ее смысл, посмотрите на приведенный далее рисунок.



Флуктуации отсчета времени

Проблема здесь состоит в том, что ваш запрос является асинхронным по отношению к источнику отсчетов. У вас нет никакой возможности синхронизировать аппаратный таймер с вашим запросом. Поэтому в итоге вы получите интервал задержки где-то в диапазоне от 20 до 30 мс — в зависимости от того, в какой момент между отсчетами аппаратных часов возник ваш запрос.

Внимание

Это очень важный момент. Флуктуации отсчета времени — одна из печальных жизненных реалий. Способ обойти эту проблему заключается в увеличении разрешающей способности так, чтобы получающиеся погрешности укладывались в пределы установленных допусков. (Как это делается, мы рассмотрим в *разд. "Опрос и установка часов реального времени и кое-что еще" далее в этой главе*.) Имейте в виду, что флуктуации проявляются только на первом отсчете таймера — задержка на 100 секунд, реализуемая с помощью таймера с разрешением в 10 мс, попадет в интервал между 100 и 100,01 секундами.

Типы таймеров

Таймер, работу которого мы только что обсудили, называют *относительным таймером*. Для такого таймера период ожидания задается относительно текущего времени. Если бы вы пожелали задержать выполнение вашего потока до 12 часов 4 минут 33 секунд EDT (Eastern Daylight Time — восточное поясное время — *прим. ред.*) 20 января 2005 года, вам пришлось бы сначала рассчитать точное число секунд от "сейчас" до выбранного вами момента и включить относительный таймер с задержкой на это число секунд. Поскольку это довольно часто встречающаяся операция, в Neutrino реализованы *абсолютные* таймеры, которые обеспечивают задержку *до* заданного времени (а не *на* заданное время, как в случае относительного таймера).

А что, если вы захотите сделать что-нибудь полезное, пока поток ожидает наступления установленной даты? Или делать что-либо и получать "синхроимпульс" каждые 27 секунд? Здесь нельзя просто так позволить себе спать!

Как мы уже обсуждали в *главе 1*, вы можете просто запустить другой поток, и пусть он выполняет работу, пока ваш поток спит. Однако, поскольку мы говорим сейчас о таймерах, посмотрим, как это можно сделать другим способом.

В зависимости от выбранной цели, вы можете сделать это с помощью либо периодического, либо однократного таймера. *Периодический таймер* — это таймер, который срабатывает периодически, уведомляя поток (снова и снова), что истек некоторый временной интервал. *Однократный таймер* — это таймер, который срабатывает только один раз.

Реализация этих таймеров в ядре основана на том же самом принципе, что и в случае с таймером задержки из нашего первого примера. Ядро запоминает абсолютное значение времени (если вы укажете сделать именно так) и хранит его. Обработчик прерываний таймера сравнивает сохраненное значение времени с текущим.

Однако вместо удаления из очереди на выполнение после системного вызова на этот раз ваш поток продолжит работу. И в момент, когда суточное время достигнет заданного вами и хранимого в памяти момента времени, ядро уведомит ваш поток о том, что назначенное время пришло.

Схема уведомления

Как получить уведомление о тайм-ауте? При использовании таймера задержки вы получаете уведомление просто посредством возвращения в состояние READY.

При использовании периодических и однократных таймеров у вас появляется выбор:

- послать импульс;
- послать сигнал;
- создать поток.

Импульсы мы уже обсуждали в *главе 2*; сигналы — стандартный для UNIX механизм. Здесь же мы кратко рассмотрим уведомления при помощи создания потока.

Как заполнять структуру *struct sigevent*

Независимо от выбранной вами схемы уведомления, вам обязательно придется заполнять структуру **`struct sigevent`**. Давайте вкратце посмотрим, как это делается.

```
struct sigevent {
    int                 sigev_notify;

    union {
        int             sigev_signo;
        int             sigev_coid;
        int             sigev_id;
        void            (*sigev_notify_function) (union sigval);
    };

    union sigval        sigev_value;

    union {
        struct {
            short       sigev_code;
            short       sigev_priority;
        };
        pthread_attr_t *sigev_notify_attributes;
    };
};
```

Внимание

Обратите внимание, что в приведенной декларации используются неименованные объединения и структуры. Внимательное изучение файла заголовка покажет вам, как этот

трюк проходит с компиляторами, не поддерживающими такую особенность. По существу, там есть директива **#define**, которая заставляет именованные объединения и структуры выглядеть неименованными. Подробнее см. **<sys/siginfo.h>**.

Первое поле, которое вы должны заполнить, — это элемент *sigev_notify*, который определяет выбранный вами тип уведомления:

- SIGEV_PULSE — будет передан импульс;
- SIGEV_SIGNAL, SIGEV_ SIGNAL _CODE или SIGEV_SIGNAL_THREAD — будет передан сигнал;
- SIGEV_UNBLOCK — в данном случае не используется; предназначен для тайм-аутов ядра *(см. разд. "Тайм-ауты ядра" далее в этой главе)*;
- SIGEV_INTR — в данном случае не используется; предназначен для прерываний *(см. главу 4)*;
- SIGEV_THREAD — будет создан поток.

Поскольку мы намерены использовать структуру **struct sigevent** для таймеров, нас будут интересовать только такие значения *sigev_notify* как SIGEV_PULSE, SIGEV_SIGNAL* и SIGNAL_THREAD; остальные мы рассмотрим в соответствующих их применению разделах.

Уведомление при помощи импульса

Чтобы передать импульс при срабатывании таймера, присвойте полю *sigev_notify* значение SIGEV_PULSE и обеспечьте немного дополнительной информации:

Поле	Значение и смысл
sigev_coid	Идентификатор соединения (connection ID), по каналу которого будет передан импульс
sigev_value	32-разрядное значение (данные импульса — *см. разд. "Что внутри импульса?" главы 2 — прим. ред.*), которое будет передано по заданному полем *sigev_coid* соединению
sigev_code	8-разрядное значение (код импульса — *см. разд. "Что внутри импульса?" главы 2 — прим. ред.*), которое будет передано по заданному полем *sigev_coid* соединению
sigev_priority	Приоритет доставки импульса. Нулевое значение не допускается — слишком уж много людей пострадало от переключения на нулевой приоритет после получения импульса, а поскольку на этом приоритете приходится конкурировать за процессор со спецпроцессом IDLE, много процессорного времени там точно не светит

Отметим, что *sigev_coid* может описывать соединение на любом канале (обычно, хотя и не обязательно, этот канал связан с процессом, который инициирует событие).

Уведомление при помощи сигнала

Чтобы передать сигнал, укажите в поле *sigev_notify* одно из перечисленных далее значений:

- SIGEV_SIGNAL — процессу будет передан обычный сигнал;
- SIGEV_SIGNAL_CODE — процессу будет передан сигнал, содержащий 8-битный код;
- SIGEV_SIGNAL_THREAD — сигнал, содержащий 8-битный код, будет передан определенному потоку.

При выборе уведомления типа SIGEV_SIGNAL* нужно будет заполнить ряд дополнительных полей:

Поле	Значение и смысл
sigev_signo	Номер сигнала для передачи (берется из **<signal.h>**, например SIGALRM)
sigev_code	8-разрядный код (для уведомления типа SIGEV_SIGNAL_CODE или SIGEV_SIGNAL_THREAD)

Уведомление созданием потока

Для создания потока по срабатыванию таймера установите поле *sigev_notify* в значение SIGEV_THREAD и заполните следующие поля:

Поле	Значение и смысл
sigev_notify_function	Адрес функции, возвращающей **`void*`** и принимающей **`void*`**, которая будет вызвана при возникновении события
sigev_value	Значение, которое будет передано функции *sigev_notify_function()* в качестве параметра
sigev_notify_attributes	Атрибутная запись потока *(см. разд. "Атрибутная запись потока" главы 1)*

Этот тип уведомления воистину страшен. Если ваш таймер будет срабатывать слишком часто, и при этом будут готовы к выполнению потоки с более высоким приоритетом, чем вновь создаваемые, то у вас быстро вырастет огромная куча заблокированных потоков, и они съедят все ресурсы вашей машины. Пользуйтесь этим типом уведомления с осторожностью!

Общие приемы программирования уведомлений

В файле **<sys/siginfo.h>** есть ряд удобных макросов для упрощения заполнения полей в структурах:

- *SIGEV_SIGNAL_INIT(eventp, signo)* — установите *eventp* в SIGEV_SIGNAL и впишите соответствующий номер сигнала *signo*;
- *SIGEV_SIGNAL_CODE_INIT(eventp, signo, value, code)* — установите поле *eventp* в SIGEV_SIGNAL_CODE, укажите номер сигнала в *signo*, а также задайте значения полей *value* и *code*;
- *SIGEV_SIGNAL_THREAD_INIT(eventp, signo, value, code)* — установите eventp в SIGEV_SIGNAL_THREAD, укажите номер сигнала в *signo*, а также задайте значения полей *value* и *code*;
- *SIGEV_PULSE_INIT(eventp, coid, priority, code, value)* — установите *eventp* в SIGEV_SIGNAL_PULSE, укажите идентификатор соединения в *coid*, а также параметры *priority*, *code* и *value*. Отметьте, что для *priority* есть специальное значение SIGEV_PULSE_PRIO_INHERIT, которое предотвращает изменение приоритета принимающего потока;
- *SIGEV_UNBLOCK_INIT(eventp)* — установите *eventp* в SIGEV_UNBLOCK;
- *SIGEV_INTR_INIT(eventp)* — установите *eventp* в SIGEV_INTR;
- *SIGEV_THREAD_INIT(eventp, func, attributes)* — задайте значения *eventp*, функции потока *func* и атрибутной записи *attributes*.

Уведомление при помощи импульса

Предположим, что вы разрабатываете сервер, который будет обречен провести большую часть своей жизни в RECEIVE-блокированном состоянии, ожидая сообщение. Идеальным вариантом здесь было бы принять специальное сообщение, указывающее, что момент, которого мы так долго ждали, наконец настал.

Как раз при таком сценарии и надо использовать импульсы в качестве схемы уведомления. В *разд. "Применение таймеров" далее в этой главе* я приведу пример кода, который можно использовать для периодического получения импульсов.

Предположим, что, с другой стороны, вы выполняете некоторую работу, но не желаете, чтобы она продолжалась вечно. Например, вы ожидаете возврата из некоторой функции, но не можете точно предсказать, сколько времени на это потребуется.

В этом случае оправданным выбором является использование уведомления при помощи сигнала — возможно, даже с обработчиком. (Другой вариант, который мы обсудим позже, заключается в использовании тайм-аутов ядра; см. также *разд. "Флаг _NTO_CHF_UNBLOCK" главы 2*.) В следующем разделе мы рассмотрим пример, использующий сигналы.

Если вы вообще не собираетесь принимать сообщения, то использование сигнала и функции *sigwait()* является более экономной альтернативой созданию канала для принятия импульсного сообщения.

Применение таймеров

Изучив все красоты теории, давайте теперь переключим наше внимание на конкретные образцы кода, чтобы посмотреть, что можно сделать при помощи таймеров.

Чтобы работать с таймером, вам потребуется:

1. Создать объект типа "таймер".
2. Выбрать схему уведомления (сигнал, импульс или создание потока) и создать структуру уведомления (**`struct sigevent`**).
3. Выбрать нужный тип таймера (относительный или абсолютный и однократный или периодический).
4. Запустить таймер.

Давайте теперь рассмотрим все это по порядку.

Создание таймера

Первый этап — это создание таймера с помощью функции *timer_create()*:

```
#include <time.h>
#include <sys/siginfo.h>

int timer_create(clockid_t clock_id, struct sigevent *event,
                 timer_t *timerid);
```

Аргумент *clock_id* сообщает функции *timer_create()*, на каком временном базисе вы формируете таймер. Это вещь из области POSIX — стандарт утверждает, что на различных платформах вы можете использовать различные типы временных базисов, но любая платформа должна, по меньшей мере, поддерживать базис CLOCK_REALTIME. В Neutrino есть три базиса:

- CLOCK_REALTIME;
- CLOCK_SOFTTIME;
- CLOCK_MONOTONIC.

Оставим пока на время варианты CLOCK_SOFTTIME и CLOCK_MONOTONIC. Мы вернемся к ним позднее в *разд. "Другие источники времени" далее в этой главе*.

Сигнал, импульс или поток?

Вторым параметром является указатель на структуру **`struct sigevent`**. Эта структура применяется для того, чтобы сообщить ядру о типе события, которое таймер должен сгенерировать при срабатывании. Мы уже обсуждали порядок заполнения **`struct sigevent`**, когда говорили о выборе схемы уведомления.

Итак, мы вызываем функцию *timer_create()* с временным базисом CLOCK_REALTIME и указателем на структуру **`struct sigevent`**, и ядро создает

объект типа "таймер" (он возвращается в последнем аргументе). Этот объект представляет собой небольшое целое число, которое является номером таймера в таблице таймеров ядра. Считайте его просто "дескриптором".

На этот момент никаких событий пока не происходит. Вы просто создали таймер, но ведь вы еще не включали его.

Какой таймер выбрать?

Создав таймер, вы должны решить, какого типа будет этот таймер. Это осуществляется путем комбинирования аргументов функции *timer_settime()*, которая обычно применяется для собственно запуска таймера:

```
#include <time.h>

int timer_settime(timer_t timerid, int flags, struct itimerspec *value,
                  struct itimerspec *oldvalue);
```

Аргумент *timerid* — это число, которое вы получите обратно по вызову функции *timer_create()*. Вы можете создать множество таймеров, а затем вызывать *timer_settime()* для них по отдельности, когда вам это будет необходимо.

С помощью аргумента *flags* вы определяете тип таймера — абсолютный или относительный.

Если вы передаете константу TIMER_ ABSTIME, получается абсолютный таймер, как вы и могли бы предположить. Затем вы передаете реальные дату и время срабатывания таймера.

Если вы передаете ноль, таймер предполагается относительным.

Давайте посмотрим, как определяется время. Вот ключевые фрагменты двух структур данных из **<time.h>**:

```
struct timespec {
    long    tv_sec,
            tv_nsec;
};

struct itimerspec {
    struct timespec it_value,
                    it_interval;
};
```

В структуре **`struct itimerspec`** есть два поля:

- *it_value* — однократно используемое значение;
- *it_interval* — перезагружаемое значение.

Параметр *it_value* задает либо интервал времени от настоящего момента до момента срабатывания таймера (в случае относительного таймера), либо собственно время срабатывания (в случае абсолютного таймера). После срабатывания таймера значение величины *it_interval* задает относительное время для повторной загрузки

таймера, чтобы он мог сработать снова. Заметим, что задание для *it_interval* нулевого значения преобразует данный таймер в однократный. Вы можете предположить, что, чтобы создать "исключительно периодический" таймер, вам следует установить параметр *it_interval* в значение интервала перезагрузки, а параметр *it_v*alue — в ноль. К сожалению, последнее неверно — установка параметра *it_value* в ноль выключает таймер. Если вы хотите создать "исключительно периодический" таймер, присвойте *it_value* и *it_interval* одинаковые значения и создайте таймер как относительный. Такой таймер сработает один раз (с задержкой *it_value*), а затем будет циклически перезагружаться с задержкой *it_interval*.

Оба параметра *it_value* и *it_interval* фактически являются структурами типа **`struct timespec`** — еще одного POSIX-объекта. Эта структура позволяет вам обеспечить разрешающую способность на уровне долей секунд. Первый ее элемент, *tv_sec*, — это число секунд, второй элемент, *tv_nsec*, — число наносекунд в текущей секунде. (Это означает, что никогда не следует устанавливать параметр *tv_nsec* в значение, превышающее 1 млрд, — это будет подразумевать смещение на более чем 1 секунду.)

Несколько примеров:

```
t_value.tv_sec = 5;
it_value.tv_nsec = 500000000;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

Это сформирует однократный таймер, который сработает через 5,5 секунды. (5,5 секунды складывается из 5 секунд и 500 000 000 нс.)

Мы предполагаем здесь, что этот таймер используется как относительный, потому что если бы это было не так, то его время срабатывания уже давно бы истекло (5,5 секунд с момента 00:00 по Гринвичу, 1 января 1970).

Другой пример:

```
it_value.tv_sec = 987654321;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

Данная комбинация параметров сформирует однократный таймер, который сработает в четверг, 19 апреля 2001 года в 00:25:21 по EDT. (Существует множество функций, которые помогут вам преобразовать воспринимаемый человеком интервал времени в "число секунд, истекшее с 00:00:00 по Гринвичу, 1 января 1970 года". См. функции *time(), asctime(), ctime(), mktime(), strftime()* и т. д.)

В данном примере мы предполагаем, что это абсолютный таймер, поскольку в противном случае ждать пришлось бы достаточно долго (987 654 321 секунд — приблизительно 31,3 года).

Отметьте, что в двух приведенных выше примерах я говорил: "мы предполагаем". В коде функции *timer_settime()* нет никаких проверок на правильность аргументов! Вы должны самостоятельно доопределить, является таймер абсолютным

или относительным. Что до ядра, то оно будет просто счастливо запланировать какое-нибудь событие на 31,3 года вперед.

И еще один пример:

```
it_value.tv_sec = 1;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_interval.tv_nsec = 500000000;
```

Если предположить, что это относительный таймер, он сработает через одну секунду и далее каждые полсекунды. Не существует никаких требований какого бы то ни было подобия значений интервала перезагрузки значениям задержки однократного срабатывания.

Сервер с периодическими импульсами

Первое, что следует рассмотреть, — это сервер, который желает получать периодические сообщения. Типовыми применениями такой схемы являются:

- поддерживаемые сервером тайм-ауты клиентских запросов;
- внутренние периодические события серверов.

Конечно, есть и другие, специализированные, применения для таких вещей — например, периодические подтверждения готовности узлов сети ("я жив"), запросы на повторную передачу и т. п.

Поддерживаемые сервером тайм-ауты

В таком сценарии сервер предоставляет клиенту некоторую услугу и клиент способен задать тайм-аут. Это может использоваться в самых разнообразных приложениях. Например, вы можете сказать серверу "выдай мне данные за 15 секунд" или "дай мне знать, когда истекут 10 секунд", или "жди прихода данных, но в течение не более чем 2 секунд".

Все это — примеры поддерживаемых сервером тайм-аутов. Клиент посылает сообщение серверу и блокируется. Сервер принимает периодические сообщения от таймера (раз в секунду, реже или чаще) и подсчитывает, сколько этих сообщений он получил. Когда число сообщений о тайм-аутах превышает время ожидания, указанное клиентом, сервер отвечает клиенту с сообщением о тайм-ауте или, возможно, с данными, которые он успел накопить на данный момент, — это зависит от того, как структурированы отношения клиента и сервера.

Далее приведен полный пример сервера, который принимает одно из двух сообщений от клиентуры и сообщения о тайм-ауте в виде импульса. Первое клиентское сообщение говорит серверу: "Дай мне знать, есть ли для меня данные, но не блокируй меня более чем на 5 секунд". Второе клиентское сообщение говорит: "Вот, возьми данные". Сервер должен позволить нескольким клиентам блокироваться на себе в ожидании данных, и поэтому обязан сопоставить клиентам тайм-ауты. Тут-то и нужен импульс; он информирует сервер: "Истекла одна секунда".

Чтобы программа не выглядела излишне громоздкой, перед каждым из основных разделов я прерываю исходный текст небольшими пояснениями. Скачать эту программу вы можете на FTP-сайте компании PARSE (**ftp://ftp.parse.com/pub/book_v3.tar.gz**), файл называется **timel.c**.

Декларации

В первом разделе программы определяются различные именованные константы и структуры данных. В нем также подключаются все необходимые заголовочные файлы. Оставим это без комментариев.

```
/*  time1.c
 *
 *  Пример сервера, получающего периодические сообщения от таймера
 *  и обычные сообщения от клиента.
 *  Иллюстрирует использование функций таймера с импульсами.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/siginfo.h>
#include <sys/neutrino.h>

// Получаемые сообщения

// Сообщения
#define MT_WAIT_DATA  2    // Сообщение от клиента
#define MT_SEND_DATA  3    // Сообщение от клиента

// Импульсы
#define CODE_TIMER    1    // Импульс от таймера

// Отправляемые сообщения
#define MT_OK         0    // Сообщение клиенту
#define MT_TIMEDOUT   1    // Сообщение клиенту

// Структура сообщения
typedef struct
{
```

```
    int messageType;   // Содержит сообщение от клиента и клиенту
    int messageData;   // Опциональные данные, зависят от сообщения
} ClientMessageT;

typedef union
{
    ClientMessageT  msg;   // Сообщение может быть либо обычным,
    struct _pulse  pulse;  //   либо импульсом
} MessageT;

// Таблица клиентов
#define MAX_CLIENT 16  // Максимум клиентов одновременно
struct
{
    int in_use;            // Элемент используется?
    int rcvid;             // Идентификатор отправителя клиента
    int timeout;           // Оставшийся клиенту тайум-аут
} clients [MAX_CLIENT];    // Таблица клиентов
int    chid;               // Идентификатор канала (глобальный)
int    debug = 1;          // Режим отладки, 1 == вкл, 0 == выкл
char    *progname = "time1.c";

// Предопределенные прототипы
static void setupPulseAndTimer(void);
static void gotAPulse(void);
static void gotAMessage(int rcvid, ClientMessageT *msg);
```

Функция *main()*

Следующий раздел кода является основным и отвечает за:

- создание канала с помощью функции *ChannelCreate()*;
- вызов подпрограммы *setupPulseAndTimer()* (для настройки периодического таймера, срабатывающего раз в секунду и использующего импульс в качестве способа доставки события;
- бесконечный цикл ожидания импульсов и сообщений и их обработки.

Обратите внимание на проверку значения, возвращаемого *MsgReceive()* — ноль указывает, что был принят импульс (здесь мы не делаем никакой дополнительной проверки, *наш* ли это импульс), ненулевое значение говорит о том, что было принято сообщение.

Обработка импульсов и сообщений выполняется функциями *gotAPulse()* и *gotAMessage()*.

```
int main(void)    // Игнорировать аргументы командной строки
{
    int rcvid;      // PID отправителя
    MessageT msg;   // Само сообщение

    if ((chid = ChannelCreate(0)) == -1) {
        fprintf(stderr, "%s: не удалось создать канал!\n", progname);
        perror(NULL);
        exit(EXIT_FAILURE);
    }

    // Настроить импульс и таймер
    setupPulseAndTimer();

    // Прием сообщений
    for (;;) {
        rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);

        // Определить, от кого сообщение
        if (rcvid == 0) {
            // Здесь неплохо бы еще проверить поле code...
            gotAPulse();
        } else {
            gotAMessage(rcvid, &msg.msg);
    }
}

   // Сюда мы никогда не доберемся
   return(EXIT_SUCCESS);
}
```

Функция *setupPulseAndTimer()*

В функции *setupPulseAndTimer()* вы видите код, в котором определяется тип таймера и схема уведомления. Когда мы рассуждали о таймерных функциях ранее, я говорил, что таймер может выдать сигнал или импульс либо создать поток. Решение об этом принимается именно здесь, в функции *setupPulseAndTimer()*. Обратите внимание, что здесь мы использовали макроопределение SIGEV_PULSE_INIT(). Используя это макроопределение, мы реально присвоили элементу *sigev_notify* значение SIGEV_PULSE. (Если бы мы использовали одно из макроопределений семейства SIGEV_SIGNAL*_INIT(), то получили бы уведомление при помощи соответствующего сигнала.) Отметьте, что при настройке импульса мы с помощью вызова

ConnectAttach() устанавливаем соединение с самим собой и даем ему уникальный код (здесь — константа CODE_TIMER; мы ее определили сами).

Последний параметр в инициализации структуры события — это приоритет импульса; здесь мы выбрали SIGEV_PULSE_PRIO_INHERIT (константа, равная –1). Это предписывает ядру не изменять приоритет принимающего импульс потока.

В конце описания функции мы вызываем *timer_create()* для создания таймера в ядре, после чего настраиваем его на срабатывание через одну секунду (поле *it_value*) и на периодическую перезагрузку односекундными интервалами (поле *it_interval*). Отметим, что таймер включается только по вызову *timer_settime()*, а не при его создании.

Внимание

Схема уведомления по типу SIGEV_PULSE — расширение, свойственное только Neutrino. Концепция импульсов в POSIX отсутствует.

```
/*  setupPulseAndTimer
 *
 *  Эта подпрограмма отвечает за настройку импульса, чтобы тот отправлял
 *  сообщение с кодом MT_TIMER. Затем устанавливается
 *  таймер с периодом в одну секунду.
*/

void setupPulseAndTimer(void)
{
    timer_t             timerid; // Идентификатор таймера
    struct sigevent     event;   // Генерируемое событие
    struct itimerspec   timer;   // Структура данных таймера
    int                 coid;    // Будем соединяться с собой

    // Создать канал к себе
    coid = ConnectAttach(0, 0, chid, 0, 0);
    if (coid == -1) {
        fprintf(stderr, "%s:  ошибка ConnectAttach!\n", progname);
        perror(NULL);
        exit(EXIT_FAILURE);
    }

    // Установить, какое событие мы хотим сгенерировать - импульс
    SIGEV_PULSE_INIT(&event, coid, SIGEV_PULSE_PRIO_INHERIT, CODE_TIMER, 0);
    // Создать таймер и привязать к событию
    if (timer_create(CLOCK_REALTIME, &event, &timerid) == -1) {
        fprintf(stderr, "%s: не удалось создать таймер, errno %d\n",
                progname, errno);
```

```
        perror(NULL);
        exit(EXIT_FAILURE);
    }

    // Настроить таймер (задержка 1 с, перезагрузка через 1 с)...
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_nsec = 0;
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_nsec = 0;

    // ...и запустить его!
    timer_settime(timerid, 0, &timer, NULL);
}
```

Функция *gotAPulse()*

В функции *gotAPulse()* вы можете видеть, как мы реализовали способность сервера обеспечивать тайм-ауты для клиентов. Мы последовательно просматриваем список клиентуры и, поскольку мы знаем, что импульс выдается один раз в секунду, просто уменьшаем число секунд, которое остается клиенту до тайм-аута. Если эта величина достигает нулевого значения, мы отвечаем этому клиенту сообщением "Извините, тайм-аут" (тип сообщения MT_TIMEDOUT). Обратите внимание, что мы подготавливаем это сообщение заранее (вне цикла **for**), а затем посылаем его по мере необходимости. Этот прием — по существу, вопрос стиля: если вы предполагаете отвечать часто, возможно, имело бы смысл выполнить настройку однажды и загодя. Если же множество ответов не ожидается, то имело бы больший смысл делать настройки по мере необходимости.

Если значение оставшегося времени еще не достигло нуля, мы не делаем ничего — клиент по-прежнему заблокирован в ожидании сообщения.

```
/*  gotAPulse
 *
 *  Эта подпрограмма отвечает за обработку тайм-аутов.
 *  Она проверяет список клиентов на предмет тайм-аута и отвечает
 *  соответствующим сообщением тем клиентам, у которых тайм-аут произошел.
 */

void gotAPulse(void)
{
    ClientMessageT  msg;
    int             i;
    if (debug) {
        time_t  now;
```

```
            time(&now);
            printf("Получен импульс, время %s", ctime(&now));
        }

        // Подготовить ответное сообщение
        msg.messageType = MT_TIMEDOUT;

        // Просмотреть список клиентов
        for (i = 0; i < MAX_CLIENT; i++) {

            // Элемент используется?
            if (clients [i].in_use) {
                // Тайм-аут?
                if (—clients [i].timeout == 0) {
                    // Ответить
                    MsgReply(clients [i].rcvid, EOK, &msg, sizeof(msg));

                    // Освободить элемент
                    clients [i].in_use = 0;
                }
            }
        }
    }
```

Функция *gotAMessage()*

В функции *gotAMessage()* вы видите другую половину заданной функциональности, где мы добавляем клиента в список клиентуры, ожидающей данные (если получено сообщение типа MT_WAIT_DATA), или сопоставляем клиента с сообщением, которое было только что получено (если это сообщение типа MT_SEND_DATA). Заметьте, что для простоты мы здесь не реализуем очередь клиентов, находящихся в ожидании передачи данных, получатель для которых еще не доступен — это вопрос управления очередями, оставьте его для себя в качестве упражнения.

```
/*  gotAMessage
 *
 *  Эта подпрограмма вызывается при каждом приеме сообщения. Проверяем тип
 *  сообщения (либо "жду данных", либо "вот данные") и действуем
 *  соответственно. Для простоты предположим, что данные никогда не ждут.
 *  Более подробно об этом см. в тексте.
 */
```

```
void gotAMessage(int rcvid, ClientMessageT *msg)
{
    int i;
    // Определить тип сообщения
    switch (msg -> messageType) {
        // Клиент хочет ждать данных
        case    MT_WAIT_DATA:
            // Посмотрим, есть ли пустое место в таблице клиентов
            for (i = 0; i < MAX_CLIENT; i++) {
                if (!clients [i].in_use) {
                    // Нашли место - пометить как занятое, сохранить rcvid
                    //   и установить тайм-аут
                    clients [i].in_use = 1;
                    clients [i].rcvid = rcvid;
                    clients [i].timeout = 5;
                    return;
                }
            }
            fprintf(stderr, "Таблица переполнена, сообщение от rcvid %d
            игнорировано, клиент заблокирован\n", rcvid);
            break;

        // Клиент с данными
        case    MT_SEND_DATA:
            // Посмотрим, есть ли другой клиент, которому можно ответить
            // данными от этого клиента
            for (i = 0; i < MAX_CLIENT; i++) {
                if (clients [i].in_use) {
                    // Нашли - использовать полученное сообщение
                    // в качестве ответного
                    msg -> messageType = MT_OK;

                    // Ответить ОБОИМ КЛИЕНТАМ!
                    MsgReply(clients [i].rcvid, EOK, msg, sizeof(*msg));
                    MsgReply(rcvid, EOK, msg, sizeof *msg));

                    clients [i].in_use = 0;
                    return;
                }
            }
            fprintf(stderr, "Таблица пуста, сообщение от rcvid %d
            игнорировано, клиент заблокирован\n", rcvid);
            break;
    }
}
```

Примечание

Несколько общих замечаний по тексту программы.

- Если сообщение с данными прибывает, когда либо никто не ждет, либо список ожидающих клиентов переполнен, в стандартный поток ошибок выводится сообщение, но клиенту при этом мы не отвечаем ничего. Это означает, что ряд клиентов может оказаться в REPLY-блокированном состоянии навсегда — идентификаторы отправителей мы потеряли, а значит, и ответ им дать не можем.
- Это сделано намеренно. Вы можете изменить это, добавив соответственно сообщения MT_NO_WAITERS и MT_NO_SPACE, которыми можно было бы отвечать всякий раз при обнаружении ошибок данного типа.
- Когда клиент-обработчик ждет, а клиент-поставщик пересылает ему данные, мы отвечаем обоим клиентам. Это критично, поскольку мы должны разблокировать обоих клиентов.
- Мы повторно использовали буфер клиента-поставщика для обоих ответов. Этот прием программирования — опять же, вопрос стиля: в большом приложении у вас, вероятно, было бы много типов возвращаемых значений, и вы могли бы и не захотеть повторно использовать одни и те же буферы.
- В приведенном примере используется "щербатый" массив фиксированной длины с флагом "элемент задействован" (`clients[i].in_use`). Поскольку моей целью здесь является отнюдь не демонстрация хитростей программирования односвязных списков, я использовал простейший для понимания вариант. В конечном же программном продукте, разумеется, имело бы смысл использовать динамический список.
- Когда функция *MsgReceive()* получает импульс, наше решение относительно того, действительно ли это "наш" импульс, фактически является весьма слабо аргументированным — мы просто предполагаем (согласно комментариям), что все входящие импульсы имеют тип CODE_TIMER. Опять же, в конечном продукте следовало бы проверять значение кода импульса и сообщать о наличии каких-либо аномалий.

Отметим, что в приведенном примере демонстрируется только один способ реализации тайм-аутов клиентуры. Позже, в этой же главе *(см. разд. "Тайм-ауты ядра")* мы поговорим о тайм-аутах ядра. Это еще один способ делать почти то же самое, только управление на этот раз осуществляется клиентом, а не таймером.

Внутренние переодические события серверов

Здесь мы имеем несколько другое применение для периодических сообщений о тайм-аутах, когда эти сообщения предназначены сервером исключительно для внутреннего использования и не имеют никакого отношения к клиенту вообще.

Например, некоторые аппаратные средства могут требовать, чтобы сервер опрашивал их периодически, — например, такое может быть в случае сетевого соединения: сервер должен периодически проверять, является ли данное подключение доступным, и это не зависит от команд клиентуры.

Другой вариант — если, например, в аппаратных средствах предусмотрен таймер "выключения по неактивности". Например, если длительное пребывание какого-то аппаратного модуля во включенном состоянии может приводить к неоправданным затратам электроэнергии, то если его никто не использует в течение, скажем, 10 секунд, его можно было бы выключить (или переключить в режим низкого энергопотребления — *прим. ред.*). Опять же, к клиенту это не имеет никакого отношения (за исключением того, что запрос от клиента отменит режим ожидания) — это просто функция, которую сервер должен уметь предоставлять "своим" аппаратным средствам.

Код в этом случае сильно бы напоминал приведенный выше пример, за исключением того, что вместо списка ожидающих клиентов у вас была бы только одна переменная тайм-аута. С каждым событием от таймера ее значение уменьшалось бы, но пока оно больше нуля, ничего бы не происходило. Когда оно стало бы равным нулю, это вызывало бы отключение аппаратных средств (или какое-либо другое соответствующее действие).

Единственный трюк здесь заключается в том, что всякий раз, когда поступает сообщение от клиента, использующего данные аппаратные средства, вы должны восстановить первоначальное значение этой переменной, поскольку обращение к ресурсу должно сбрасывать "обратный отсчет". И наоборот, аппаратным средствам может потребоваться определенный промежуток времени "на разогрев" после включения. В этом случае после выключения аппаратных средств вам придется при поступлении запроса от клиента организовать еще один таймер, чтобы "придержать" запрос до того момента, пока аппаратные средства не станут готовы.

Таймеры, посылающие сигналы

На настоящий момент мы уже рассмотрели практически все, что относится к таймерам, за исключением одного небольшого момента. Мы обеспечивали отправку импульса, но у нас также есть возможность посылать POSIX-сигналы. Давайте посмотрим, как это делается:

```
timer_create(CLOCK_REALTIME, NULL, &timerid);
```

Это простейший способ создать таймер, который будет посылать вам сигнал. Он обеспечивает выдачу сигнала SIGALRM при срабатывании таймера. Если бы мы предоставили **`struct sigevent`**, мы могли бы определить, какой именно сигнал мы хотим получить:

```
struct sigevent event;
SIGEV_SIGNAL_INIT(&event, SIGUSR1);
timer_create(CLOCK_REALTIME, &event, &timerid);
```

Это обеспечит нам выдачу сигнала SIGUSR1 вместо SIGALRM.

Сигналы таймера перехватываются обычными обработчиками сигналов, здесь нет ничего необычного.

Таймеры, создающие потоки

Если вы хотите по каждому срабатыванию таймера создавать новый поток, то можете это сделать с помощью **struct sigevent** и всех остальных таймерных штук, которые мы только что обсудили:

```
struct sigevent event;
SIGEV_THREAD_INIT(&event, maintenance_func, NULL);
```

Однако пользоваться этим надо очень осторожно, потому что если вы определите слишком короткий интервал, вы можете просто утонуть в создаваемых потоках. Они просто поглотят все ресурсы вашего процессора и оперативной памяти.

Опрос и установка часов реального времени и кое-что еще

Независимо от применения таймеров, вы можете также опрашивать и устанавливать часы реального времени, а также и плавно подстраивать их. Для этих целей можно использовать следующие функции:

Функция	Тип	Описание
ClockAdjust()	*Neutrino*	Плавная регулировка времени
ClockCycles()	*Neutrino*	Опрос с высоким разрешением
clock_getres()	*POSIX*	Выборка базового разрешения
clock_gettime()	*POSIX*	Получение текущего времени суток
ClockPeriod()	*Neutrino*	Получение/установка базового разрешения
clock_settime()	*POSIX*	Установка текущего времени суток
ClockTime()	*Neutrino*	Получение/установка текущего времени суток

Опрос и установка

Функции *clock_gettime()* и *clock_settime()* являются POSIX-функциями, основанными на системном вызове *ClockTime()*. Эти функции могут применяться для получения и установки текущего времени суток. К сожалению, установка здесь является "жесткой", т. е. независимо от того, какое время вы указываете в буфере, оно немедленно делается текущим. Это может иметь пугающие последствия, особенно когда получается, что время "повернуло вспять", потому что устанавливаемое время оказалось меньше "реального". Вообще настройка часов таким способом

должна выполняться только при включении питания или когда время сильно не соответствует "реальному".

Если нужна плавная корректировка текущего времени, ее можно реализовать с помощью функции *ClockAdjust():*

```
int ClockAdjust(clockid_t id,
                const struct _clockadjust *new,
                const struct _clockadjust *old);
```

Параметрами здесь являются источник синхроимпульсов (всегда используйте CLOCK_REALTIME) и параметры *new* и *old*. Оба эти параметра являются необязательными и могут быть заданы как NULL. Параметр *old* просто возвращает текущую корректировку. Работа по корректировке часов управляется параметром *new*, который является указателем на структуру, содержащую два элемента, *tick_nsec_inc* и *tick_count*. Действует функция *ClockAdjust()* очень просто — каждые *tick_count* отсчетов системных часов к существующему значению системного времени добавляется корректировка *tick_nsec_inc*. Это означает, что, чтобы передвинуть время вперед ("догоняя" реальное), вы задаете для *tick_nsec_inc* положительное значение. Заметьте, что не надо переводить время назад — вместо этого, если ваши часы спешат, задайте для *tick_nsec_inc* небольшое отрицательное значение, и ваши часы соответственно замедлят ход. Таким образом, вы немного замедляете часы, пока их показания не будут соответствовать действительности. Существует эмпирическое правило, гласящее, что не следует корректировать системные часы значением, превышающим 10% от базового разрешения вашей системы (см. функцию *ClockPeriod()* и ее "друзей", о них мы поговорим в следующем разделе).

Регулировка разрешающей способности

Как мы и говорили на протяжении всей этой главы, нельзя сделать ничего с большей точностью, чем принятая в системе базовая разрешающая способность по времени. Напрашивается вопрос: а как настроить эту базовую разрешающую способность? Для этого вы можете использовать следующую функцию:

```
int ClockPeriod(clockid_t id,
                const struct _clockperiod *new,
                struct _clockperiod *old,
                int reserved);
```

Как и в случае с описанной ранее функцией *ClockAdjust()*, с помощью параметров *new* и *old* вы получаете и/или устанавливаете значения базовой разрешающей способности по времени. Параметры *new* и *old* являются указателями на структуры типа **`struct _clockperiod`**, которые, в свою очередь, содержат два элемента — *nsec* и *fract*. На настоящий момент элемент *fract* должен быть равен нулю (это число фемтосекунд (миллиардная доля микросекунды — *прим. ред.*); нам, вероятно, это еще не скоро потребуется). Параметр *nsec* указывает, сколько наносекунд содержится в интервале между двумя базовыми отсчетами времени. Значение этого интервала времени по умолчанию — 10 мс, поэтому значение *nsec* (если вы исполь-

зуете функцию для получения базового разрешения) будет приблизительно равно 10 млн нс. (Как мы уже упоминали в *разд. "Источники прерывания таймера" ранее в этой главе*, это не будет в точности равняться 10 мс.)

При этом вы можете, конечно, не стесняться и попробовать назначить базовой разрешающей способности какое-нибудь смехотворно малое значение, но тут вмешается ядро и эту вашу попытку пресечет. В общем случае, в большинстве систем допускаются значения от 1 мс до сотен микросекунд.

Точные временные метки

Существует одна система отсчета времени, которая не подчиняется описанным выше правилам "базовой разрешающей способности по времени". Некоторые процессоры оборудованы встроенным высокочастотным (высокоточным) счетчиком, к которому Neutrino обеспечивает доступ при помощи функции *ClockCycles()*. Например, в процессоре Pentium, работающем с частотой 200 МГц, этот счетчик увеличивается тоже с частотой в 200 МГц, и поэтому он может обеспечить вам значение времени с точностью до 5 нс. Это особенно полезно, когда вы хотите точно выяснить, сколько времени затрачивается на выполнение конкретного фрагмента кода (в предположении, конечно, что он не будет вытеснен). В этом случае вы должны вызвать функцию *ClockCycles()* перед началом вашего фрагмента и после его окончания, а потом просто подсчитать разность полученных отсчетов. Более подробно это описано в руководстве по Си-библиотеке.

Внимание

Обратите внимание, что в SMP-системе можно сталкнуться с одной проблемкой. Если поток считывает значение *ClockCycles()* из одного ЦПУ, а затем выполняется на другом ЦПУ, то результаты могут быть недостоверными. Причина этого заключается в том, что каждый ЦПУ имеет собственный счетчик, используемый функцией *ClockCycles()*, и эти счетчики не синхронизируются между ЦПУ. В таком случае решением является принудительная привязка потока к одному из ЦПУ.

Дополнительные возможности

Теперь, после ознакомления с основами таймеров, мы можем взглянуть на некоторые дополнительные вопросы:

- типы таймеров CLOCK_SOFTTIME и CLOCK_MONOTONIC;
- тайм-ауты ядра.

Другие источники времени

Мы рассмотрели источник времени CLOCK_REALTIME и упомянули, что реализация, соответствующая POSIX, может поддерживать самые разные источ-

ники времени, которые ей нравятся, но должна как минимум поддерживать CLOCK_REALTIME.

Что вообще значит "источник времени"? Считайте, что это просто абстрактный источник информации о времени. Если хотите аналогию из повседневной жизни, то ваши личные часы являются источником времени; они измеряют, как быстро течет время. Точность ваших часов может отличаться от точности часов кого-то другого. Вы можете забыть завести часы или сменить батарейку, при этом время будет казаться остановившимся. Или вы можете перевести стрелки ваших часов, и время как будто сделает неожиданный скачок. Все это — характеристики источника времени.

В Neutrino источник CLOCK_REALTIME основан на часах "текущего времени суток", предоставляемых микроядром Neutrino. (В дальнейших примерах мы будем называть его "временем Neutrino".) Это означает, что если во время работы системы кто-либо внезапно переведет время на 5 секунд вперед, то это изменение может повредить или не повредить вашей программе (в зависимости от того, что она делает). Давайте рассмотрим, что произойдет при вызове функции *sleep(30)*:

Астрономическое время	**Время Neutrino**	**Действия**
11:22:05	11:22:00	*sleep(30)*
11:22:15	11:22:15	Часы переведены на 11:22:15; они отставали на 5 секунд!
11:22:35	11:22:35	*sleep (30)*; просыпаемся

Прекрасно! Поток сделал то, что вы ожидали: в 11:22:00 он заснул на тридцать секунд и проснулся в 11:22:35 (тридцать секунд спустя). Заметьте, что кажется, будто сон на функции *sleep()* длился 35 секунд вместо 30; на самом деле прошло только 30 секунд, просто часы Neutrino были переведены на пять секунд вперед (в 11:22:15).

Ядро знает, что вызов *sleep()* использует относительный таймер, поэтому оно заботится о том, чтобы на самом деле прошел заданный интервал времени.

Так, а что было бы, если бы мы использовали, наоборот, абсолютный таймер, и в 11:22:00 по "времени Neutrino" сказали ядру разбудить нас в 11:22:30?

Астрономическое время	**Время Neutrino**	**Действия**
11:22:05	11:22:00	Проснуться в 11:22:30
11:22:15	11:22:15	Часы переведены как в прошлый раз
11:22:30	11:22:30	Просыпаемся

И снова все происходит так, как вы того ожидали — вы хотели, чтобы вас разбудили в 11:22:30, и, несмотря на перевод времени, вас разбудили. Однако на самом деле таймер сработал через 25 секунд вместо полных тридцати. Для некоторых приложений это может быть проблемой.

CLOCK_MONOTONIC

Люди, пишущие POSIX, думали над этой проблемой и в качестве ее решения придумали другой источник времени, называемый CLOCK_MONOTONIC. Когда вы создаете таймерный объект с помощью функции *timer_create()*, вы можете сказать ей, какой источник времени нужно использовать.

Течение времени CLOCK_MONOTONIC происходит таким способом, что оно *никогда* не корректируется. В результате этого, независимо от того, сколько сейчас на самом деле времени, если вы укажете таймеру с источником времени CLOCK_MONOTONIC сработать через 30 секунд, а затем выполните любые корректировки времени, которые желаете, то таймер сработает по истечении 30 секунд.

Источник времени CLOCK_MONOTONIC обладает следующими характеристиками:

- всегда увеличивает счетчик;
- основан на *реальном* течении времени;
- начинается с нуля.

Внимание

Важной особенностью часов, начинающихся с нуля, является то, что они принадлежат к иной "эпохе", нежели эпоха CLOCK_REALTIME, начинающаяся с 1 января 1970 года в 00:00:00 по Гринвичу. И хотя время обих часов течет в одном и том же темпе, но их значения *несовместимы*.

Функции, которые не позволяют вам выбирать источник времени — такие как *pthread_mutex_timedlock()* — используют CLOCK_REALTIME. Этот источник задан жестко и вы не можете его изменить.

Так что же делает CLOCK_SOFTTIME?

Если вы желаете отсортировать источники времени по "жесткости", то мы получим следующую последовательность. Вы можете рассматривать CLOCK_MONOTONIC как товарный поезд — он не делает остановок ни для кого. Следующим в списке окажется CLOCK_REALTIME, поскольку его можно немного сдвинуть (как мы видели на примере перевода времени). И в конце списка у нас будет источник времени CLOCK_SOFTTIME, который мы можем смещать *на много*.

Как и следует из названия, CLOCK_SOFTTIME в основном применяется для "мягких" вещей — вещей, невыполнение которых не приведет к критическому отказу. CLOCK_SOFTTIME является "активным" только, пока работает процессор.

(Да, это звучит понятно, но подождите!) Когда процессор останавливается по причине, когда система управления электропитанием обнаружила, что в ближайшее время ничего не случится, источник времени CLOCK_SOFTTIME тоже выключается!

Далее приведена таблица, иллюстрирующая эти три источника времени:

Астрономическое время	**Время Neutrino**	**Действия**
11:22:05	11:22:00	Разбудить в "сей момент" + 00:00:30 (см. ниже)
11:22:15	11:22:15	Перевели время как в прошлые разы
11:22:20	11:22:20	Управление электропитанием выключает ЦПУ
11:22:30	11:22:30	Срабатывает таймер CLOCK_REALTIME
11:22:35	11:22:35	Срабатывает таймер CLOCK_MONOTONIC
11:45:07	11:45:07	Управление электропитанием включает ЦПУ и срабатывает таймер CLOCK_SOFTTIME

По этому поводу есть пара замечаний.

- Мы рассчитали время побудки, как "сей момент" плюс 30 секунд, и использовали для побудки в расчетное время абсолютный таймер. Это *отличается* от побудки *через* 30 секнд с использованием относительного таймера.
- Обратитите внимание, что для удобства размещения всего примера в единой хронологии мы немного приврали. Дело в том, что если поток и в самом деле будет разбужен срабатыванием таймера CLOCK_REALTIME, (а позже так же CLOCK_MONOTONIC), то это приведет к активизации ЦПУ, что, в свою очередь, разбудит CLOCK_SOFTTIME.

Если CLOCK_SOFTTIME "пересыпает", то он просыпается, как только это возможно — его течение времени не останавливается при выключенном ЦПУ, он только не имеет возможности просыпаться до включения ЦПУ. В остальном CLOCK_SOFTTIME такой же, как CLOCK_REALTIME.

Использование разных источников времени

Для указания различных источников времени необходимо использовать POSIX-функции управления временем, принимающие в качестве параметра идентификатор времени (clock ID). Например:

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *rqtp,
                    struct timespec *rmtp);
```

Функция *clock_nanosleep()* принимает параметр *clock_id* (говорящий, какой источник времени использовать), флаг (который задает относительное или абсолютное время), "запрашиваемое время сна" *(rqtp)*, а так же указатель на область памяти, в которую функция может записывать количество оставшегося до срабатывания таймера времени (параметр *rmtp*, который может быть NULL, если вам все равно).

Тайм-ауты ядра

Neutrino позволяет вам получать тайм-ауты по всем блокированным состояниям. Мы обсуждали эти состояния в *разд. "Состояния потоков" главы 1*. Наиболее часто у вас может возникнуть потребность в этом при обмене сообщениями: клиент, посылая сообщение серверу, не желает ждать ответа "вечно". В этом случае было бы удобно использовать тайм-аут ядра. Тайм-ауты ядра также полезны в сочетании с функцией *pthread_join():* завершения потока тоже не всегда хочется долго ждать.

Далее приводится объявление функции *TimerTimeout()*, которая является системным вызовом, ответственным за формирование тайм-аутов ядра.

```
#include <sys/neutrino.h>

int TimerTimeout(clockid_t id, int flags,
                 const struct sigevent *notify,
                 const uint64_t *ntime,
                 uint64_t *otime);
```

Видно, что функция *TimerTimeout()* возвращает целое число (индикатор удачи/неудачи; 0 означает, что все в порядке, –1 — что произошла ошибка, и ее код записан в *errno*). Источник синхроимпульсов (CLOCK_REALTIME и т. п.) указывается в *id*, параметр *flags* задает соответствующее состояние (или состояния). Параметр *notify* всегда должен быть событием уведомления типа SIGEV_UNBLOCK; параметр *ntime* указывает относительное время, спустя которое ядро должно сгенерировать тайм-аут. Параметр *otime* показывает предыдущее значение тайм-аута и в большинстве случаев не используется (вы можете передать вместо него NULL).

Внимание

Важно отметить, что тайм-ауты "взводятся" функцией *TimerTimeout()*, а запускаются по входу в одно из состояний, указанных в параметре *flags*. Сбрасывается тайм-аут при возврате из любого системного вызова. Это означает, что вы должны заново "взводить" тайм-аут перед каждым системным вызовом, к которому вы хотите его применить. Сбрасывать тайм-аут после системного вызова не надо — это выполняется автоматически.

Тайм-ауты ядра и функция *pthread_join()*

Самый простой пример для рассмотрения — это использование тайм-аута с функцией *pthread_join()*.

Вот как это можно было бы сделать:

```
/* tt1.c */
#include <stdio.h>
#include <pthread.h>
#include <inttypes.h>
#include <errno.h>
#include <sys/neutrino.h>

#define SEC_NSEC 1000000000LL  // В одной секунде 1 биллион наносекунд

void *long_thread(void *notused)
{
    printf("Этот поток выполняется более 10 секунд\n");
    sleep(20);
}

int main(void) // Игнорировать аргументы
{
    uint64_t        timeout;
    struct sigevent event;
    int             rval;
    pthread_t       thread_id;

    // Настроить событие — это достаточно сделать однажды.
    // Либо так, либо event.sigev_notify = SIGEV_UNBLOCK:
    SIGEV_UNBLOCK_INIT(&event);

    // Создать поток
    pthread_create(&thread_id, NULL, long_thread, NULL);

    // Установить тайм-аут 10 секунд
    timeout = 10LL * SEC_NSEC;

    TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN, &event, &timeout, NULL);

    rval = pthread_join(thread_id, NULL);
    if (rval == ETIMEDOUT) {
        printf("Истекли 10 секунд, поток %d все еще выполняется!\n",
               thread_id);
    }

    sleep(5);

    TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN, &event, &timeout, NULL);
    rval = pthread_join(thread_id, NULL);
```

```
    if (rval == ETIMEDOUT) {
        printf("Истекли 25 секунд, поток %d все еще выполняется (нехоро-
шо)!\n",
                thread_id);
    } else {
        printf("Поток %d завершен (как и ожидалось!)\n", thread_id);
    }
}
```

Мы применили макроопределение *SIGEV_UNBLOCK_INIT()* для инициализации структуры события, но можно было установить *sigev_notify* в *SIGEV_UNBLOCK* и "вручную". Можно было даже сделать еще более изящно, передав NULL вместо **`struct sigevent`** — функция *TimerTimeout()* понимает это как знак, что нужно использовать SIGEV_UNBLOCK.

Если поток (заданный в *thread_id*) остается работающим более 10 секунд, то системный вызов завершится по тайм-ауту — функция *pthread_join()* возвратится с ошибкой, установив *errno* в ETIMEDOUT.

Вы можете использовать и другую "стенографию", указав NULL в качестве значения тайм-аута (параметр *ntime* в объявлении, приведенном ранее), что предпишет ядру не блокироваться в данном состоянии. Этот прием можно использовать для организации программного опроса. (Хоть программный опрос и считается дурным тоном, его можно весьма эффективно использовать в случае с *pthread_join(),* периодически проверяя, завершился ли нужный поток. Если нет, можно пока сделать что-нибудь другое.)

Далее представлен пример программы, в которой демонстрируется неблокирующий вызов *pthread_join()*:

```
int pthread_join_nb(int tid, void **rval)
{
    TimerTimeout(CLOCK_REALTIME, _NTO_TIMEOUT_JOIN, NULL, NULL, NULL);
    return(pthread_join(tid, rval));
}
```

Тайм-ауты ядра при обмене сообщениями

Все становится несколько сложнее, когда вы используете тайм-ауты ядра при обмене сообщениями. Вспомните *разд. "Обмен сообщениями и модель "клиент/сервер"" главы 2* — на момент отправки клиентом сообщения сервер может как ожидать его, так и нет. Это означает, что клиент может заблокироваться как по передаче (если сервер еще не принял сообщение), так и по ответу (если сервер принял сообщение, но еще не ответил). Основной смысл здесь в том, что вы должны предусмотреть оба блокирующих состояния в параметре *flags* функции *TimerTimeout()*, потому что клиент может оказаться в любом из них.

Чтобы задать несколько состояний, сложите их операцией ИЛИ (OR):

```
TimerTimeout(... _NTO_TIMEOUT_SEND | _NTO_TIMEOUT_REPLY, ...);
```

Это вызовет тайм-аут всякий раз, когда ядро переведет клиента в состояние блокировки по передаче (SEND) или по ответу (REPLY). В тайм-ауте SEND-блокировки нет ничего особенного — сервер еще не принял сообщение, значит, ничего для этого клиента он не делает. Это значит, что, если ядро генерирует тайм-аут для SEND-блокированного клиента, сервер об этом информировать не обязательно. Функция *MsgSend()* клиента возвратит признак ETIMEDOUT, и обработка тайм-аута завершится.

Однако, как было упомянуто в *разд. "Флаг _NTO_CHF_UNBLOCK" главы 2*, если сервер уже принял сообщение клиента и клиент желает разблокироваться, для сервера существуют два варианта реакции. Если сервер не указал флаг _NTO_CHF_UNBLOCK на канале, по которому было принято сообщение, клиент будет разблокирован немедленно, и сервер не получит об этом никакого оповещения. У большинства серверов, которые мне доводилось встречать, флаг _NTO_CHF_UNBLIOCK был всегда установлен. В этом случае ядро посылает серверу импульс, а клиент остается заблокированным до тех пор, пока сервер ему не ответит! Как было показано в вышеупомянутом разделе *главы 2*, это сделано для того, чтобы сервер мог узнать о запросе клиента на разблокирование и выполнить по этому поводу какие-то действия.

Резюме

Мы рассмотрели функции Neutrino, ответственные за манипулирование временем, включая таймеры и их примененение, а также тайм-ауты ядра. Относительные таймеры обеспечивают генерацию событий "через определенное число секунд", в то время как абсолютные таймеры генерируют события "в определенное время". Таймеры (и, вообще говоря, структура **`struct sigevent`**) могут обеспечить как выдачу импульса или сигнала, так и создание потока.

Ядро создает таймеры, сохраняя абсолютное время, представляющее последующее "событие", в отсортированной очереди и сравнивая текущее время (при помощи обработчика прерываний таймера) со значением, расположенным в голове этой очереди. Когда текущее время становится больше или равно времени, хранящегося в головном элементе очереди, очередь просматривается на предмет дополнительных совпадений, после чего ядро диспетчеризует события или потоки (в зависимости от типа элемента очереди) и, возможно, производит перепланирование.

Для обеспечения поддержки функций энергосбережения вы обязаны отключать периодические таймеры, когда в них нет необходимости, иначе энергосбережения как такового не произойдет — система будет все время думать, что у нее есть работа для периодического выполнения. Так же вам следует использовать CLOCK_SOFTTIME, за исключением, конечно, тех случаев, когда вы на самом деле хотите помешать энергосбережению.

Благодаря различным типам источников времени вы имеете возможность выбирать базис для часов и таймеров; начиная с "реального, текущего", до источников времени, основанных на управлении питанием.