ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА (САМАРСКИЙ УНИВЕРСИТЕТ)»

ИНСТИТУТ ИНФОРМАТИКИ И КИБЕРНЕТИКИ
Кафедра программных систем

А.В. Баландин

Основы программирования приложений реального времени

Учебное пособие

Самара
2025

УДК 681.3.066
ББК 32.973.26-018.2

Рецензенты: докт. техн. наук, проф. Самарского университета С. А. Прохоров,
зам. технического директора по ПО и НИОКР, ООО НВФ «Сенсоры,
Модули, Системы», канд. техн. наук В. Е. Захарченко.

Баландин, Александр Васильевич
Основы программирования приложений реального времени: Учебное пособие / А. В. Баландин. - Самара. Издательство Самарского университета, 2025. - 206 с.: ил.

В учебном пособии изложены теоретические аспекты параллельных вычислений над оперативными данными с ограниченной во времени актуальностью (темпоральными данными), составляющие основу разработки приложений реального времени. Рассмотрены средства API операционной системы QNX Neutrino, которые используются как базовые средства программирования на языке C или C++ процессно-нитевой структуры многопоточного, параллельного и распределённого приложения, осуществляющего параллельные вычисления над разделяемыми темпоральными данными в режиме реального времени.

Пособие подготовлено на кафедре программных систем в качестве курса лекций для магистров, обучающихся по направлению 02.04.02 «Фундаментальная информатика и информационные технологии», осваивающих основы и базовые средства программирования приложений реального времени при изучении дисциплины "Технологии промышленного программирования".

УДК 681.3.066
ББК 32.973.26-018.2

ОГЛАВЛЕНИЕ

ЧАСТЬ 1. ОНТОЛОГИЯ ПРИЛОЖЕНИЙ РЕАЛЬНОГО ВРЕМЕНИ

Введение

Существует класс технических кибернетических систем, к которым, в частности, относятся системы промышленной автоматизации, робототехнические устройства, встроенные в технологическое оборудование цифровые устройства управления и т.п., осуществляющих контроль и управление состоянием физических объектов в реальном времени. Такие системы называют *системами реального времени*, или сокращённо – СРВ [1][2][3][4]. В общем случае СРВ может быть распределённой, фрагменты которой взаимодействуют посредством передачи данных.

Программные системы, обеспечивающие работу систем реального времени, составляют особый класс программных приложений, называемых *приложениями реального времени*, или сокращённо – ПРВ. Особенностью функционирования приложения реального времени в составе СРВ является его непрерывный во времени обмен данными с физическим объектом, представленным датчиками и приводами (механизмы управления объектом), с которыми приложение связано специальными аппаратно-программными интерфейсами получения с датчиков значений физических параметров объекта для контроля и анализа его текущего состояния, а также доставку управляющих данных приводам для изменения значений физических параметров управления объектом. Вместе с этим ПРВ может в реальном времени параллельно выполнять, аналитическую обработку над множеством оперативно изменяющихся данных, репликацию данных в его распределённых фрагментах, архивирование данных в системной базе данных СРВ, а также осуществлять обмен данными с объектами своего окружения. Если абстрагироваться от семантики обработки данных, то *онтологическая модель*[1] (схема) фрагмента ПРВ выглядит так, как представлено на Рис. 1.

Концептуально функционирование приложения осуществляется таким образом, что каждый фрагмент ПРВ посредством периферийных *служб* (подсистем) в режиме реального времени выполняет обмен данными с внешними *источниками* или *приёмниками* данных, в качестве которых выступают: *физический объект*, *оператор*, другие *распределённые фрагменты* ПРВ, *системная база данных СРВ*. Все вместе они составляют так называемое *окружение* фрагмента ПРВ. При этом на периферийные подсистемы возлагаются следующие функции:

- *Служба сбора данных и управления* - осуществляет *транспонирование*[2] данных между ПРВ и физическим объектом.
- *Служба обмена данными с оператором* - посредством человеко-машинного интерфейса обеспечивает оператора информацией о состоянии как физического объекта, так и системы в целом, а также предоставляет необходимые средства управления объектом.

[1] Онтологическая модель — это информационная модель концептуального описания предметной области, которая включает описание множества объектов и связей между ними.

[2] **Транспонирование данных** – конвертация (преобразование) значений параметров физического объекта из аналоговой формы представления в программный тип данных или наоборот, используя датчики или приводы.

- *Служба репликации данных* - осуществляет синхронизацию копий темпоральных данных в распределённых фрагментах ПРВ.
- *Служба архивирования данных* - сохраняет в базе данных СРВ тренды данных для анализа динамики темпоральных данных, а также историю изменения состояния физического объекта и/или системных параметров.

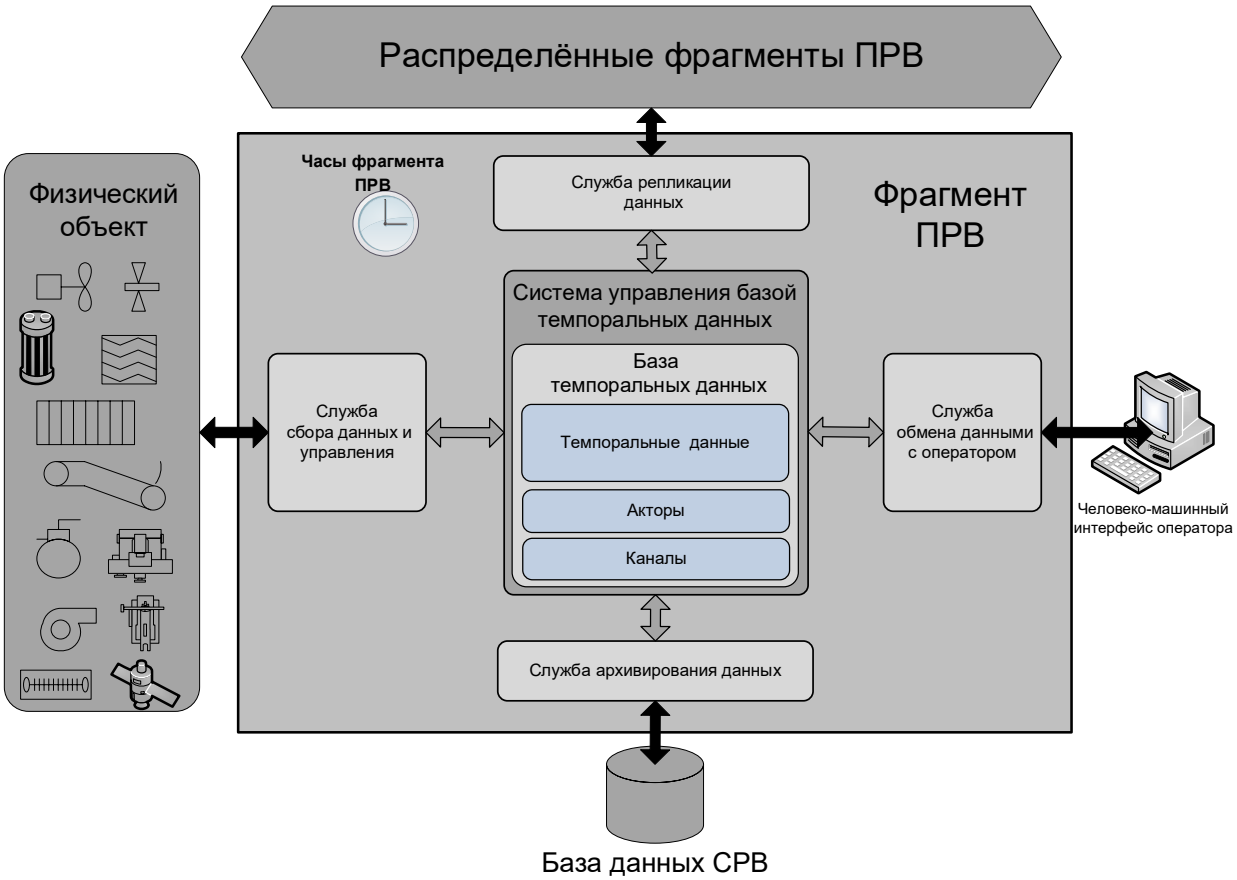


Рис. 1 Схема фрагмента ПРВ

Система управления базой темпоральных данных является вычислительным ядром фрагмента ПРВ, которое посредством *акторов* (вычислительных процедур) осуществляет оперативную обработку и актуализацию *темпоральных данных*, взаимодействуя посредством *каналов* обмена данными с *периферийными службами*, связывающими фрагмент ПРВ с его *окружением* как с источниками или потребителями данных.

Фрагмент ПРВ масштабируется как по составу необходимого набора периферийных служб, так и по составу базы темпоральных данных.

1. Служба реального времени ПРВ

1.1. Системное время и реальное время ПРВ

Актуальность значений темпоральных данных напрямую связана со временем, поэтому получение и использование значений данных, соответствующих текущему времени является для ПРВ важным условием эффективного функционирования. Приложение реального времени для *датирования*[3] полученного значения темпорального данного использует текущее показание реального времени, формируемое глобальной службой относительного реального времени всего ПРВ (далее коротко – *служба времени* ПРВ), формирующей показания часов относительного времени ПРВ с точностью заданного тика и начинающая отсчёт времени с нуля в момент запуска службы (начало штатного режима функционирования ПРВ). Показания часов распределённых локальных фрагментов ПРВ реплицируется с показаниями часов службы времени ПРВ (Рис. 2).

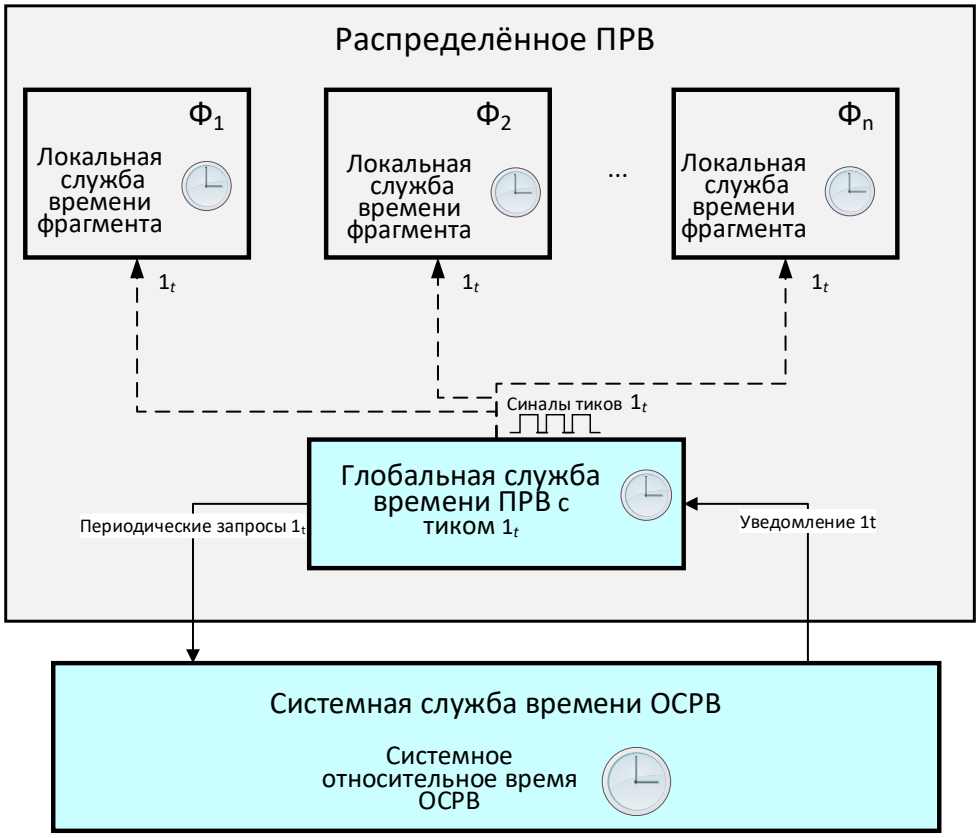


Рис. 2 Служба реального времени ПРВ

Необходимая точность шкалы часов службы времени ПРВ определяется требуемой скоростью реакции ПРВ на изменение контролируемого состояния внешнего окружения (значения данных разных состояний не должны фиксироваться с одинаковой меткой времени). При этом предельная точность определяется точностью шкалы часов службы относительного системного времени той ОСРВ, в среде которой функционирует ПРВ [9][10]. Далее, текущее

[3] Датирование темпорального данного – это процесс присвоения значениям изменяющегося во времени темпорального данного временных характеристик, позволяющих оценивать в текущий момент реального времени актуальность использования его значения в темпоральных вычислениях.

значение относительного системного времени часов ОСРВ будем обозначать как t, а текущее значение относительного времени локальных часов службы времени ПРВ будем обозначать $\tilde{t}$ и полагать, что величина тика - $1t$шкалы времени $\tilde{t}$ задаётся по шкале системного времени t службы времени используемой ОСРВ.

ПРВ для задания значения протяжённости тика $1t$ шкалы реального времени $\tilde{t}$ своей службы времени в качестве аналога "*непрерывного времени*" t использует шкалу относительного времени службы времени ОСРВ, под которой ПРВ функционирует. Служба времени ПРВ посылает периодические запросы в системную службу времени ОСРВ об истечении интервала системного времени протяжённостью $1t$ - *единица шкалы* времени ПРВ, и ожидает уведомления. По истечении интервала протяжённости $1t$ относительного системного времени служба времени ОСРВ посылает уведомление об этом службе времени ПРВ. В результате показание часов относительного реального времени ПРВ, начиная с нуля, периодически скачком возрастает на 1. Событие изменения показания часов ПРВ одновременно транслируется во все распределённые фрагменты ПРВ, локальные часы которых используются для датирования значений темпоральных данных. Локальные часы фрагментов ПРВ постоянно находятся в состоянии ожидания сигнала уведомления службы времени ПРВ об истечении интервала $1t$, после чего их показание увеличивается на 1 и они вновь переходят в состояние ожидания очередного сигнала.

Текущее показание локальных часов реального времени $\tilde{t}$ с тиком $1t$ формально обозначим - $time_{1t}$. Очевидно, что текущее показание часов ПРВ - $\tilde{t}$, в тиках 1_t будет связано с текущим показанием часов относительного системного времени ОСРВ – t, формулой: $\tilde{t} = \left\lfloor \frac{t}{1_t} \right\rfloor$ – целая от деления. Протяжённость тика 1_t часов ПРВ программно задаётся в единицах и долях секунды с помощью системного объекта ОСРВ – *относительный таймер*.

Так как очередной тик $\tilde{t}$ фактически является подмножеством оси системного времени ОСРВ, то запись $t \in \tilde{t}$ означает, что текущий момент системного относительного времени t находится в пределах текущего тика службы времени ПРВ - $\tilde{t}$.

1.2. Одномоментность в реальном времени

Заданная величина тика $1t$ часов ПРВ характеризует способность ПРВ *одномоментно* актуализировать базу темпоральных данных в темпе своего реального времени - $\tilde{t} = time_{1t}$, т.е. в течение интервала системного времени $1t$. Это означает, что все темпоральные данные, утратившие валидность в некоторый момент времени $\tilde{t}_i, i = 0,1,2, \ldots$, должны в этот же момент времени и обновиться в течение интервала системного времени протяжённостью $1t$, т.е. одномоментно.

Одномоментность является фундаментальным понятием реального времени $time_{1t}$. Так как практически актуализация темпоральных данных в ПРВ не может быть осуществлена мгновенно, то все события и значения данных, полученные или сформированные в течение текущего тика часов ПРВ, считаются одномоментными, т.е. их результаты будут иметь одну и ту же метку времени - $\dot{t} = time_{1t}$, полученную по часам ПРВ.

Очевидно, что заданная в системном времени ОСРВ протяжённость тика $1t$ часов ПРВ принципиально влияет на способность ПРВ одномоментно актуализировать набор

темпоральных данных, у которых завершился или запланированно, или спорадически[4] по событию период репрезентативности в момент времени $\dot{t} = time_{1t}$. Чем меньше протяжённость тика $1t$ часов ПРВ, тем ниже вероятность того, что, при имеющихся ресурсах вычислительной системы, ПРВ выполнит все активированные акторы в течение текущего тика и успеет актуализировать в БТД все темпоральные данные, утратившие валидность в момент времени $\dot{t} = time_{1t}$. Утратившие валидность темпоральные данные не могут быть использованы активированными акторами в вычислениях в качестве исходных данных (*темпоральный прецедент* потери данного). И наоборот, с увеличением протяжённости тика $1t$ вероятность темпоральных прецедентов стремится к нулю. Поэтому для ПРВ, выполняющегося в конкретной вычислительной системе, существует минимальная величина тика часов ПРВ - $1t_{min}$, ниже которой приложение теряет способность функционировать в заданном реальном времени $time_{1t<1t_{min}}$ часов ПРВ без темпоральных прецедентов.

[4] **Спорадически**: непостоянно, изредка, с непредсказуемыми перерывами.

2. База темпоральных данных

База темпоральных данных (БТД) любого фрагмента ПРВ включает в свой состав следующие программные объекты:

- *Темпоральные данные* – конечный набор данных абстрактных типов, актуальность значений которых в реальном времени (валидность) обеспечивается системой управления БТД фрагмента.
- *Акторы* - конечный набор хранимых процедур, используемых системой управления БТД фрагмента ПРВ: для вычисления и обновления значений темпоральных данных в локальной базе (актуализации), репликации значений в распределённых фрагментах, согласования значений с окружением фрагмента.
- *Каналы* – конечный набор программных объектов, используемых системой управления БТД для *коммуникации данных* между темпоральными данными БТД и периферийными службами.

Система управления базой темпоральных данных (СУБТД) представляет собой службу оперативного управления актуализацией темпоральных данных БТД фрагмента ПРВ, используя каналы взаимодействия с периферийными службами и акторы. Работа СУБТД во времени осуществляется циклически в темпе *Часов* фрагмента ПРВ – системный объект фрагмента, управляемый *службой глобального относительного реального времени* всего ПРВ в целом.

2.1. Схема и управление базой темпоральных данных

Схема базы темпоральных данных представлена тремя множествами объектов БТД (**Ошибка! Источник ссылки не найден.**):

- множество *темпоральных данных*,
- множество *каналов*,
- множество *акторов.*

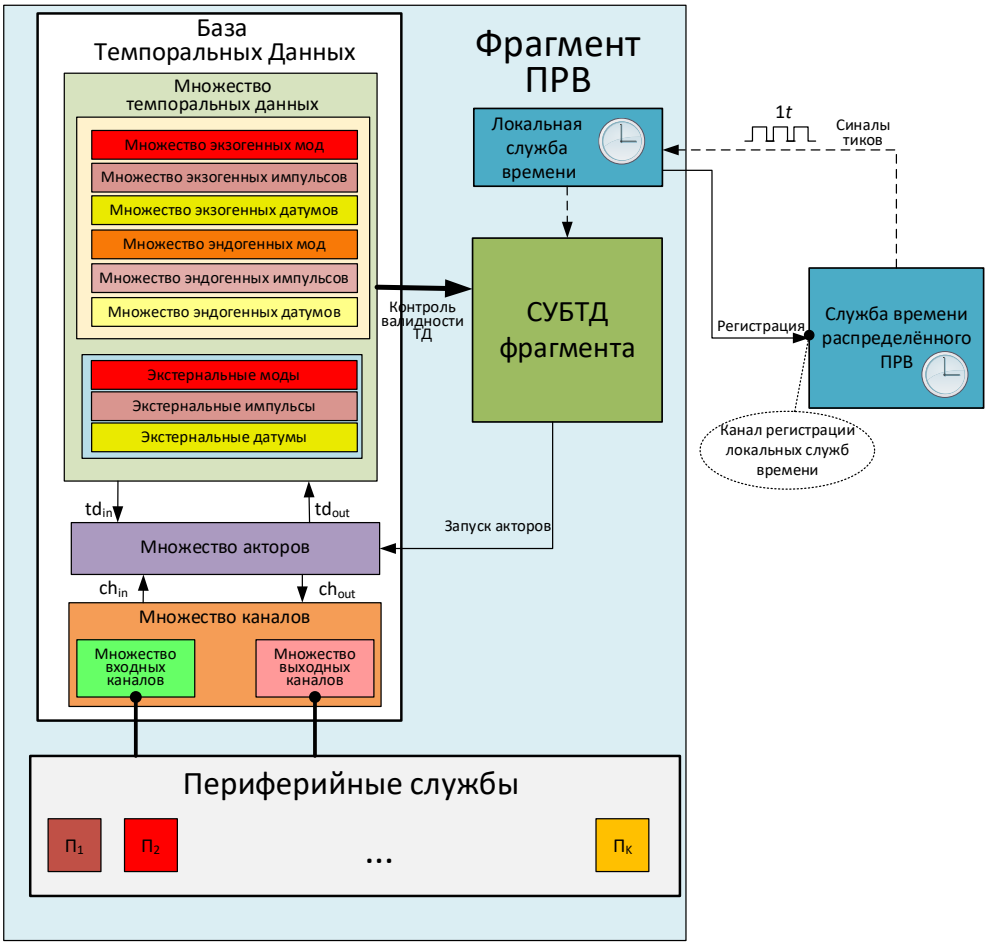


Рис. 3 Схема базы темпоральных данных

Множество *темпоральных данных* предназначены для хранения изменяющиеся в реальном времени значений параметров предметной области ПРВ, поставляемых периферийными службами или вычисляемых акторами.

Множество *каналов* предназначено для коммуникации данных между периферийными службами ПРВ и буферами каналов, связанными с темпоральными данными БТД. Каналы делятся на входные и выходные. *Входные каналы* принимают в свой буфер актуальные данные от периферийных служб, которые используются акторами для обновления не валидных значений *экзогенных* данных БТД. *Выходные каналы* наоборот полученные в свой буфер от акторов обновлённые значения темпоральных данных БТД отправляют периферийным службам.

Множество *акторов* используются СУБТД как хранимые процедуры для актуализации соответствующих темпоральных данных значением, полученным актором либо из буфера входного канала, либо вычисленным актором над соответствующим набором валидных темпоральных данных БТД. Кроме этого, акторы используются для обновления буферов выходных каналов актуализированными значениями экстернальных данных БТД. С формальной точки зрения акторы реализуют в БТД жёсткие направленные связи: между буферами входных каналов и экзогенными данными, входными и выходными темпоральными данными, экстернальными данными и буферами выходных каналов. Связанные таким образом объекты БТД представляют собой *схему базы темпоральных данных* ПРВ.

Формально схема базы темпоральных данных представляется как конечное множество $TDB = TD \cup CH \cup P$, являющееся объединением трёх попарно не пересекающихся множеств объектов: $TD \cap CH = \emptyset$, $TD \cap P = \emptyset$, $CH \cap P = \emptyset$; где TD - множество *темпоральных данных*, CH - множество *каналов*, P – множество *акторов*.

Множество TD представимо как: $TD = D \cup E$, $D \cap E = \emptyset$; где D – подмножество *экзогенных* данных, E – подмножество *эндогенных* данных.

Множество TD представимо также как объединение двух не пересекающихся подмножеств *интернальных* и *экстернальных* данных: $TD = TD_{internal} \cup TD_{external}$, $TD_{internal} \cap TD_{external} = \emptyset$.

Множество CH представляет собой объединение двух не пересекающихся подмножеств: $CH = CH_{IN} \cup CH_{OUT}$; CH_{IN} - подмножество *входных* каналов, CH_{OUT} - подмножество *выходных* каналов, $CH_{IN} \cap CH_{OUT} = \emptyset$.

Множество акторов P объединяет четыре не пересекающихся друг с другом подмножеств $P = P_D \cup P_E \cup P_T \cup P_C$: P_D – подмножество акторов актуализации экзогенных данных значениями из буферов входных каналов; P_E - подмножество акторов актуализации эндогенных данных значениями, вычисленными над наборами темпоральных данных; P_T - подмножество акторов обновления буферов выходных каналов значениями актуализированных экстернальных данных; P_C – акторы репликации семантически связанных в распределённых фрагментах БТД экстернальных и соответствующих им экзогенных темпоральных данных.

Система управления базой темпоральных данных в начале каждого очередного тика часов ПРВ контролирует валидность всех темпоральных данных БТД и активирует параллельное выполнение акторов для актуализации не валидных ТД, а также акторов обновления буферов выходных каналов значениями актуализированных в текущем тике экстернальных данных.

2.2. Темпоральные данные

В обычной практике мы редко задумываемся по поводу актуальности во времени значений данных, полученных тем или иным способом. Это объясняется тем, что практическая значимость значений данных либо вовсе не зависит от времени, либо после их практического использования необходимость в них сразу пропадает. Например, полученное когда-то значение числа π со временем вообще не теряет своей актуальности. А вот значение температуры тела больного, зафиксированное в некоторый момент времени, не теряет актуальности (практической значимости для врача) лишь в течение предсказуемого промежутка времени, и при его завершении требует обновления. Если актуальность значений данных ограничена во времени и их необходимо периодически обновлять для поддержания в актуальном состоянии, то такие данные называются *темпоральными* [6]. Специфика разработки приложений реального времени заключается в наличии и использовании в вычислениях значительного количества темпоральных данных, особенностью которых является то, что помимо текущего значения они характеризуются так называемыми *периодом репрезентативности* и *валидностью* [7][8].

Период репрезентативности - период времени, в течение которого полученное в начале этого периода значение темпорального данного считается актуальным до его окончания. Протяжённость периода репрезентативности называется **интервалом репрезентативности**.

Вали́дность — показатель актуальности или допустимости использования полученного ранее значения темпорального данного в темпоральных вычислениях в текущий момент времени.

На практике полагают, что в течение периода репрезентативности значение темпорального данного в рамках заданной погрешности его измерения в непрерывном времени не меняется и для каждого момента времени в течение периода репрезентативности по определению является валидным для использования в вычислениях над темпоральными данными. Например, температура тела здорового человека, измеряемая с точностью до градуса, будет иметь значение 36° в течение продолжительного интервала времени при изменении температуры в диапазоне [36°,37°), и изменится только при достижении значения 37° или снижении температуры ниже 36°. Поэтому интересоваться каждый час маловероятным изменением температуры тела здорового человека в градусах не имеет смысла. Очевидно, интервал репрезентативности температуры тела здорового человека может быть продолжительным, например, сутки. Но в момент окончания периода репрезентативности возрастает вероятность изменения ранее измеренного значения температуры, оно становится во времени всё менее вероятным ("нечётким"), и не может в текущий момент времени с уверенностью считаться актуальным и использоваться либо непосредственно, либо в вычислениях, по результатам которых принимаются ответственные решения. Поэтому полагают, что в момент завершения заданного периода репрезентативности значение темпорального данного теряет актуальность и его требуется обновить (актуализировать).

2.2.1. Модель темпоральных данных

Так как в общем случае текущее фиксированное значение темпорального данного в некоторый момент системного времени теряет свою актуальность, то формально обозначим темпоральное данное как $td^{\nu(t)}$, где $\nu(t) \in \{0,1\}$ индекс, который характеризует текущую валидность значения темпорального данного $td^{\nu(t)}$ в момент t системного времени:

$$\nu(t) = \begin{cases} 1 - \text{валидное}, \\ 0 - \text{не валидное} \end{cases}.$$

Модель изменяющегося в системном времени t темпорального данного $td^{\nu(t)}$ обозначим кортежем: $td^{\nu(t)} \equiv \langle \dot{z}, \dot{t}, \dot{\tau} \rangle^{\nu(t)}$, где $\dot{z}$ – значение абстрактного типа, актуализированное в момент времени $\dot{t} := time_{1_t}$ часов ПРВ; $\dot{\tau}$ - величина интервала репрезентативности в тиках часов ПРВ. Значение $\dot{z}$ валидного темпорального данного td^1 характеризуется во времени периодом репрезентативности $[\dot{t}, \dot{t} + \dot{\tau}]$ (*Period of Representativeness* - *PR*), в течение которого значение $\dot{z}$ остаётся неизменным, а темпоральное данное $td^{\nu(t)=1}$ валидным в текущий момент системного времени t, при условии, что $\tilde{t} = \left\lfloor \frac{t}{1_t} \right\rfloor \in [\dot{t}, \dot{t} + \dot{\tau}]$. Далее формально период репрезентативности темпорального данного $td^{\nu(t)}$ в момент t системного времени будем обозначать:

$$PR\left(td^{\nu(t)}\right) = \begin{cases} [\dot{t}, \dot{t} + \dot{\tau}], \ \nu(t) = 1 - \text{валидное}, \\ \emptyset, \ \nu(t) = 0 \qquad\quad - \text{не валидное} \end{cases}.$$

Период репрезентативности характеризуется интервалом репрезентативности - своей протяжённостью $\dot{\tau}$ в тиках часов ПРВ (*Interval of Representativeness*), который формально будем обозначать:

$$IR\left(td^{\nu(t)}\right) = \begin{cases} \dot{\tau}, \ \nu(t) = 1 - \text{валидное}, \\ 0, \ \nu(t) = 0 - \text{не валидное} \end{cases}.$$

В общем случае разные актуализированные значения темпорального данного могут иметь разные интервалы репрезентативности. При истечении у валидного темпорального данного $td^1 = \langle \dot{z}, \dot{t}, \dot{\tau} \rangle^1$ последнего тика периода репрезентативности (в момент времени $\ddot{t} = \dot{t} + \dot{\tau}$ по часам ПРВ) темпоральное данное становится не валидным - $td^0 = \langle \dot{z}, \dot{t}, 0 \rangle^0$, $IR(td^0) = 0$, $PR(td^0) = \emptyset$ - пустое множество, и должно актуализироваться в соответствии с особенностями изменения темпорального данного во времени: *непрерывно* – динамическое данное, *событийно* – статическое данное. Актуализация темпорального данного должна осуществиться одномоментно по часам ПРВ, то есть начаться и завершиться в течение одного тика $\ddot{t}$.

2.2.2. Категории темпоральных данных

Каждый тик часов реального времени система управления БТД фрагмента ПРВ контролирует текущую валидность темпоральных данных. Если выявляется не валидное темпоральное данное, то СУБТД активирует соответствующий актор, который выполняет необходимые действия для актуализации не валидного данного. По способу актуализации и последующего использования темпоральные данные БТД делятся на экзогенные или эндогенные, экстернальные или интернальные.

2.2.2.1. Экзогенные и эндогенные данные

Множество темпоральных данных БТД по способу актуализации активированным актором разделяется на две категории - *экзогенные* (первичные - канальные) или *эндогенные* (вторичные - вычисляемые).

При актуализации не валидное экзогенное данное используется активированным актором в качестве *выходного* аргумента, обновляемого актуальным значением из *буфера* входного канала – *входной* аргумент актора. Не валидное эндогенное данное используется актором как *выходной* аргумент обновляемый значением *вычисленным* актором на заданном наборе валидных темпоральных данных БТД - *входной* набор аргументов.

2.2.2.2. Экстернальные и интернальные данные

Темпоральные данные БТД в отношении с периферийными службами делятся на две категории – *интернальные* или *экстернальные*.

Темпоральное данное является экстернальным, если в результате его актуализации СУБТД активирует актор, обновляющий *буфер* соответствующего выходного канала - *выходной* аргумент актора, для передачи каналом обновлённого значения присоединённой периферийной службе. Темпоральное данное является интернальным, если оно не связано ни с одним из буферов выходных каналов (его обновлённое значение не требуется передавать периферийным службам).

2.2.3. Классы темпоральных данных

Класс темпорального данного определяет порядок обновления и способ формирования периода репрезентативности при актуализации темпорального данного.

По изменению во времени в множестве темпоральных данных $TD \in TDB$ различают три класса темпоральных данных:

- *датум* (datum),
- *импульс* (pulse),

– *мода* (mode).

Формально темпоральное данное с указанием класса представляется в виде - $td^{\nu(t)}_{class\in\{datum,pulse,mode\}}$.

2.2.3.1. Датум

Темпоральные данные класса *датум* - $td^{\nu(t)}_{datum}$, являются моделью динамических данных, непрерывно изменяющихся во времени. Принципиальным для датума является то, что интервал репрезентативности $IR(td^1) \equiv \dot{\tau}$ формируемого нового значения априори известен. Такими данными являются, например, параметры физического объекта: температура, давление, обороты, координаты нахождения в пространстве, скорость перемещения, расход топлива, объём заполнения ёмкости и т.п., мониторинг которых во времени осуществляется с заданной частотой.

Датум является экзогенным, если он актуализируется значением, взятым актором непосредственно из буфера входного канала, и эндогенным, если он актуализируется значением вычисленным актором над соответствующим входным набором темпоральных данных. Для экзогенного датума величина интервала репрезентативности определяется заданным порогом чувствительности к изменениям значения соответствующего физического параметра во времени относительно предыдущего замера (апертура). Поэтому по истечении интервала репрезентативности СУБТД должна одномоментно обновить датум значением, полученным по входному каналу от периферийной службы (Рис. 1). Следовательно формально, валидное темпоральное данное класса *датум* - $td^1_{datum} = \langle \dot{z}, \dot{t}, \dot{\tau} \rangle^1_{datum}$, в текущий момент системного времени $t \in PR(td^1_{datum}) \equiv [\dot{t}, \dot{t} + \dot{\tau}]$ имеет значение $\dot{z}$, полученное в момент времени ПРВ - $\dot{t}$, и является репрезентативным в течение конечного априори заданного периода репрезентативности $[\dot{t}, \dot{t} + \dot{\tau}]$ с интервалом репрезентативности $\dot{\tau} \equiv IR(td^1_{datum})$. Датум должен одномоментно обновиться не позднее, чем в течение тика времени ПРВ - $\ddot{t} = \dot{t} + \dot{\tau}$, следующего за периодом репрезентативности. Исключением являются эндогенные датумы, зависящие от темпоральных данных класса $mode$, у которых период репрезентативности может спорадически завершиться раньше запланированного для датума момента $\ddot{t}$. В этом случае эндогенный датум должен обновиться одномоментно с модой, от которой зависит его значение.

В момент времени $\ddot{t}$, когда датум перестаёт быть валидным, индикатор валидности сбрасывается в 0 - $\nu(\ddot{t}) = 0$, интервал репрезентативности обнуляется - $IR(td^0) = 0$, период репрезентативности становится неопределённым - $PR(td^0) = \emptyset$. Формально датум принимает значение - $td^0_{datum} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{datum}$ – не валидный датум. Для корректной актуализации датума, она должна начаться и завершиться в момент потери датумом валидности (запланировано или спорадически), т.е. должна выполниться одномоментно. В момент завершения актуализации происходит замена соответствующим актором текущего значения $\dot{z}$ на вновь полученное значение $\ddot{z}$, нулевое значение интервала репрезентативности заменяется априори заданным значением интервала репрезентативности $\ddot{\tau} > 0$, индикатор валидности устанавливается в единицу. В результате актуализированный датум становится валидным и принимает значение - $td^1_{datum} = \langle \ddot{z}, \ddot{t}, \ddot{\tau} \rangle^1_{datum}$.

2.2.3.2. Импульс

Темпоральные данные класса *импульс* предназначены для представления в БТД различного рода сигналов, отображающих спорадически возникающие на объекте мониторинга контролируемые события, при возникновении которых требуется своевременное выполнение активированными акторами необходимых действий в течение периода репрезентативности импульса, после чего импульс перестаёт быть валидным. Такими сигналами являются, например, различные сигналы контроля аварийных ситуаций, сигналы срабатывания реле или датчиков замыкания/размыкания (концевиков), сигналы срабатывания оптических датчиков, фиксирующих событие распознавания контролируемого объекта, и т.п. Это предупреждающие или тревожные сигналы, поступающие от периферийных служб в буферы входных каналов, для актуализации экзогенных импульсов БТД. Такие сигналы могут поступать и от периферийной службы обмена данными с оператором, как реакция оператора на некое контролируемое им особое состояние объекта. Например, неадекватное возрастание температуры или давления может потребовать от оператора послать сигнал о переводе объекта в аварийный режим работы. Кроме того, тревожные или предупреждающие сигналы могут формироваться непосредственно и в самой БТД в виде актуализации эндогенных импульсов специальными акторами, которые распознают на заданном входном наборе темпоральных данных возникновение системных критических состояний, требующих для их устранения соответствующих ограниченных во времени реакций ПРВ.

Импульсы составляют класс темпоральных данных БТД, для которых с точки зрения ПРВ отсутствие актуальности текущих значений является «естественным», т.е. $td^0_{pulse} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{pulse}$ — это нормальное состояние импульса в текущий момент времени. Импульсы выражают спорадическое возникновение контролируемых событий, с «продолжительной», но ограниченной во времени актуальностью в течение априори заданного периода репрезентативности, но в отличие от датума, импульс не имеет непрерывное во времени валидное значение. Быть до момента возникновения контролируемого события не валидным — $td^0_{pulse} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{pulse}$ — это практически "желаемое" значение импульса. В момент возникновения контролируемого события - $\ddot{t}$, значение импульса td^0_{pulse} актуализируется соответствующим актором - $td^1_{pulse} = \langle \ddot{z}, \ddot{t}, \ddot{\tau} \rangle^1_{pulse}$, и импульс становится валидным в течение вновь сформированного интервала репрезентативности $IR(td^1) = \ddot{\tau}$, по истечении которого в момент $\dot{t} = \ddot{t} + \ddot{\tau}$ текущее значение импульса вновь теряет валидность - $td^0_{pulse}(t) = \langle \ddot{z}, \ddot{t}, 0 \rangle^0_{pulse}$, оставаясь таким в течение неопределённого интервала времени до момента очередного спорадически возникающего события, актуализирующего импульс.

2.2.3.3. Мода

Темпоральные данные класса *мода* отражают в БТД абстрактные статические параметры, изменяющиеся во времени спорадически, и как частный случай - константы. Это, могут быть параметры конфигурации как физического объекта (например, различные уставки границ изменения значений физических параметров), так и системы в целом (например, системный параметр, определяющий текущий режим работы). Установка в БТД исходного значения моды может осуществляться до начала работы приложения (до запуска часов ПРВ), например вручную оператором при загрузке приложения. А также и в режиме функционирования приложения, когда, например, изменение в реальном времени на объекте значения

переключателя режима работы доставляется периферийной службой, обновляя буфер входного канала, связанного с экзогенной модой. В результате активированный в текущем тике актор СУБТД и актуализирует моду значением из обновлённого буфера входного канала. Таким образом, в общем случае значение моды во времени обновляется спорадически, и остаётся актуальным в течение априори неопределённого, но по величине большего нуля, интервала репрезентативности - $\dot{\tau} = \not\approx \gg 0$, до момента очередного спорадического события потери валидности, индикатор валидности текущего значения моды равен единице - $\nu(t) = 1$.

Теоретически, в непрерывном времени, мода должна актуализироваться мгновенно в момент спорадической потери валидности. Это означает, что контролировать событие спорадической потери модой валидность требуется в каждом тике в реального времени часов ПРВ, а необходимым условием корректной актуализации моды является её обновление в том же тике, в котором валидность была потеряна.

Таким образом, моды составляют особый класс темпоральных данных БТД, для которых интервал репрезентативности является неопределённым. Формально, валидная в текущий момент времени часов ПРВ - $\ddot{t} > \dot{t}$, мода представляется в виде $td^1_{mode} = \langle \dot{z}, \dot{t}, \not\approx \rangle^1_{mode}$, где $\not\approx > 0$ – обозначение неопределённого интервала репрезентативности. Но, так как изменение значения моды происходит спорадически, то СУБТД в начале каждого очередного наступившего тика $\ddot{t}$ временно делает моду не валидной - $td^0_{mode} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{mode}$, и активирует соответствующий актор актуализации моды. Если актор не выявил событие обновления моды, то он сразу восстанавливает валидность текущего значения моды - $td^1_{mode} = \langle \dot{z}, \dot{t}, \not\approx \rangle^1_{mode}$. Иначе, мода остаётся не валидной до момента завершения её актуализации. С момента завершения актуализации моды в некотором тике и до его окончания обновлённое значение моды остаётся валидным - $td^1_{mode} = \langle \ddot{z}, \ddot{t}, \not\approx \rangle^1_{mode}$, что позволяет использовать её другими активированными в этом тике акторами в качестве входного значения. Далее всё повторяется.

2.3. Каналы

Каналы – это интерфейсные объекты базы темпоральных данных, связывающие БТД с периферийными службами фрагмента ПРВ. Входные каналы используются СУБТД либо для получения от периферийных служб актуальных значений для обновления экзогенных данных – *входные каналы*, либо для экспорта актуализированных значений экстернальных данных в периферийные службы. В соответствии с этим каналы делятся на входные - ch_{in}, и выходные - ch_{out}.

Каждый канал имеет интерфейсную точку для присоединения периферийной службы к фрагменту ПРВ, и буфер для размещения передаваемых значений, размер которого соответствует типу соответствующего темпорального данного. Класс обновляемого экзогенного или реплицируемого экстернального данного определяет соответствующий тип и протокол работы связанного с ними канала. При этом входные каналы могут быть двух типов - *инициативные* или *пассивные*, а выходные – только *пассивными*.

Каждый канал управляется собственной процедурой, реализующей протокол управления каналом. Инициативный входной канал управляется по протоколу $proactive_{in}$, пассивный входной канал управляется по протоколу $inactive_{in}$, а все выходные каналы являются пассивными и управляется по протоколу $inactive_{out}$.

Далее подмножества входных и выходных каналы БТД с учётом их протоколов работы обозначим:

$CH_{IN} = \left\{ ch_{in_1}^{prot_{in_1}}, ch_{in_2}^{prot_{in_2}}, \dots, ch_{in_{N_{in}}}^{prot_{in_{N_{in}}}} \right\}$ – подмножество из N_{in} *входных* каналов $ch_{in_i}^{prot_{in_i}}, prot_{in_i} \in \{proactive_{in}, inactive_{in}\}, \ i \in \overline{1, N_{in}}$;

$CH_{OUT} = \left\{ ch_{out_1}^{inactive_{out}}, ch_{out_2}^{inactive_{out}}, \dots, ch_{out_{N_{out}}}^{inactive_{out}} \right\}$ – подмножество из N_{out} пассивных *выходных* каналов $ch_{out_j}^{inactive_{out}}, \ j \in \overline{1, N_{N_{out}}}$.

2.3.1. Схема инициативного входного канала

Инициативный входной канал используется СУБТД для актуализации экзогенного данного БТД класса *datum*. Периферийная служба, связанная с входным каналом БТД соединением типа $coid$ играет пассивную роль. Инициатором посылки периферийной службой в канал сообщения выступает канал БТД, который активируется сигналом инициации работы канала, поступившим от СУБТД при наступлении последнего тика периода репрезентативности датума, и посылает запрос периферийной службе на получение от неё нового значения в буфер канала. В следующем тике СУБТД активирует актор актуализации уже не валидного экзогенного датума обновлённым значением, полученным в предыдущем тике в буфер канала от периферийной службы. Принципиальная схема инициативного входного канала представлена на Рис. 4.

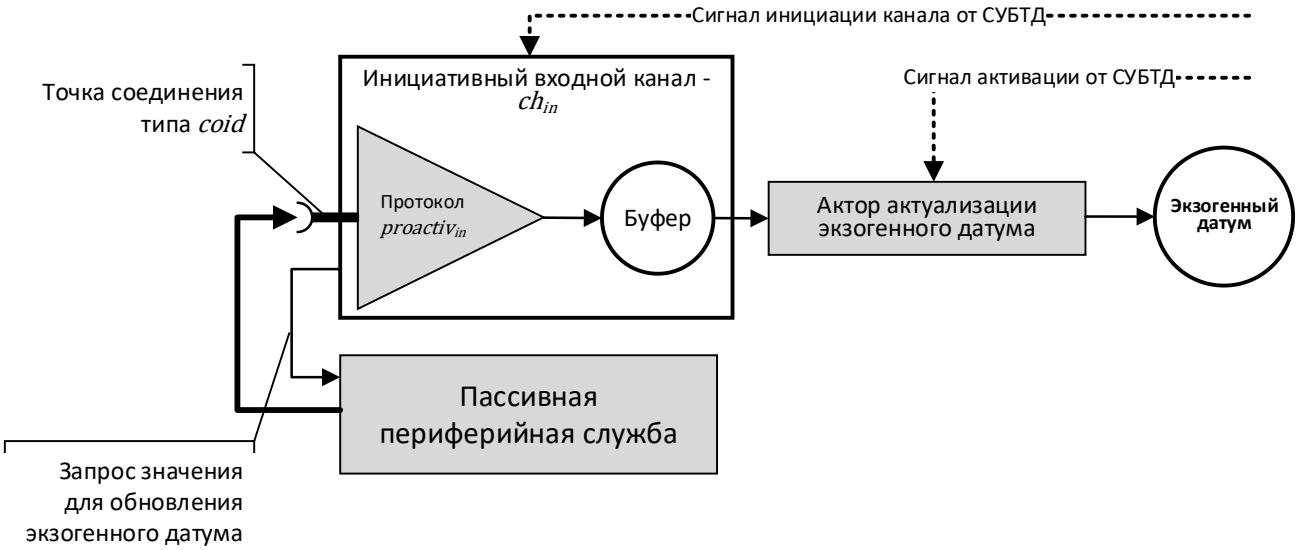


Рис. 4 Схема инициативного входного канала

По соединению типа $coid$ инициативный входной канал работает по протоколу $proactive_{in}$, в соответствии с которым он находится в состоянии ожидания сигнала инициации работы от СУБТД для обновления буфера входного канала. Запущенный инициативный канал посылает периферийной службе сообщение-запрос на посылку в канал сообщения со значением для актуализации связанного с входным каналом экзогенного датума. Поступившее сообщение обновляет содержимое буфера входного канала, которое в дальнейшем будет использовано актором актуализации экзогенного датума.

2.3.2. Схема пассивного входного канала

Пассивный входной канал используется СУБТД для актуализации экзогенного данного БТД класса *pulse* или *mode*. Инициатором посылки в канал сообщения выступает *периферийная служба*, присоединённая к точке соединения с пассивным каналом типа $chid$, когда возникает контролируемое ею событие. Принципиальная схема пассивного входного канала представлена на Рис. 5.

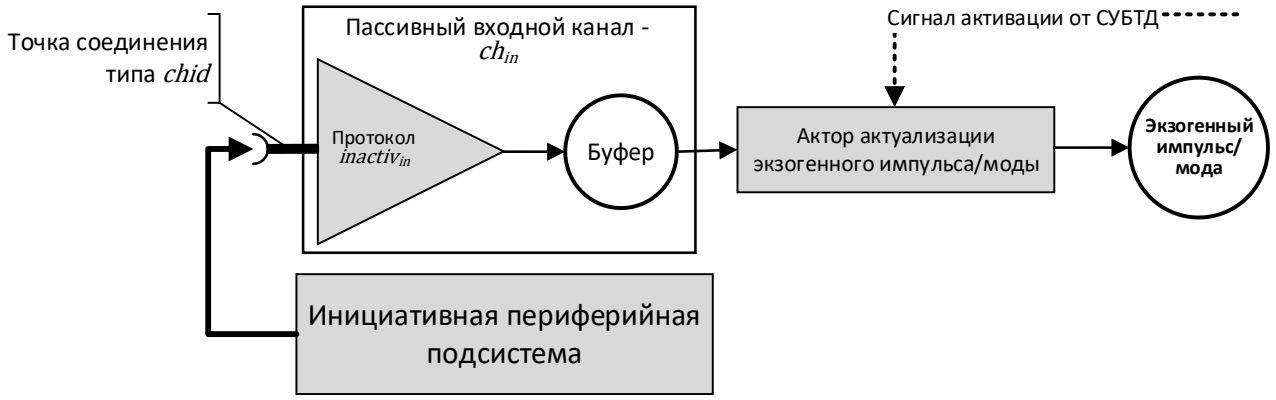


Рис. 5 Схема пассивного входного канала

По соединению типа $chid$ пассивный входной канал работает по протоколу $inactive_{in}$, в соответствии с которым он находится в состоянии постоянного ожидания спорадического прихода от инициативной периферийной службы сообщения со значением для актуализации соответствующего экзогенного импульса или моды. Когда сообщение спорадически поступает в канал, то полученное значение обновляет содержимое буфера канала.

СУБТД использует пассивный входной канал так, что в начале каждого тика часов ПРВ она активирует соответствующий актор актуализации импульса или моды. Работа актора начинается с того, что в начале он проверяет признак обновления буфера канала. Если буфер канала обновлён, то актор использует значение буфера для обновления соответствующего темпорального данного БТД. В противном случае он просто завершает свою работу до следующей активизации при наступлении очередного тика часов ПРВ.

2.3.1. Схема выходного канала

Выходные каналы используются СУБТД для экспорта обновлённых экстернальных данных БТД фрагмента ПРВ в периферийные службы или репликации темпоральных данных в распределённых фрагментах ПРВ.

Посылка выходным каналом сообщения периферийной службе инициируется поступившим от неё запросом на получение обновлённого значения экстернального данного. Принципиальная схема выходного канала представлена на Рис. 6.

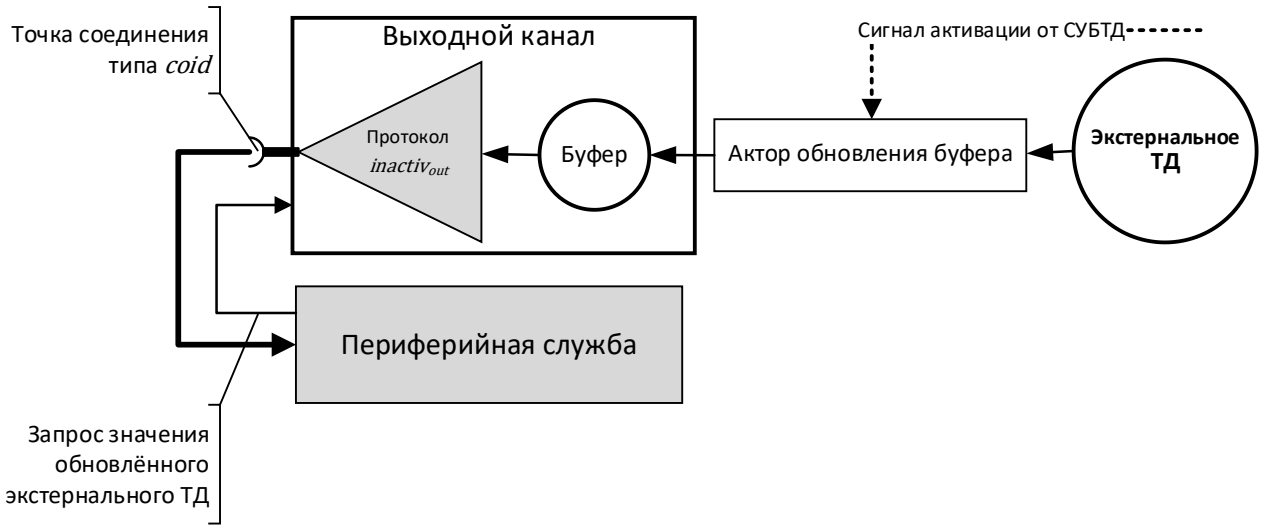


Рис. 6 Схема выходного канала

Буфер выходного канала обновляется актором каждый раз при обновлении экстернального данного БТД. При этом канальная процедура реализует протокол пассивного канала - $inactive_{out}$, ожидая прихода запроса от периферийной службы на получение сообщения с обновлённым значением соответствующего экстернального данного. Следует заметить, что обновление актором буфера выходного канала зависит только от состояния экстернального данного, и никак не связано с наличием или отсутствием запроса от периферийной службы на передачу сообщения с обновлённым значением буфера канала.

2.4. Акторы

В базе темпоральных данных акторы составляют множество хранимых вычислительных процедур – P, которое представляет собой объединение четырёх не пересекающихся подмножеств однотипных акторов - $P = D \cup E \cup T \cup C$:

D – акторы актуализации экзогенных ТД значениями, полученными из буферов входных каналов, связанных с периферийными службами;

E – акторы актуализации эндогенных ТД значениями, вычисленными над входными наборами темпоральных данных БТД;

T – акторы экстернальных ТД, актуализирующие значения данных в буферах выходных каналов, связанных с периферийными службами;

C – акторы репликации темпоральных данных в распределённых фрагментах БТД.

2.4.1. Транзакции базы темпоральных данных

В общем случае выполнение акторов рассматривается как параллельно выполняемые транзакции актуализации БТД. Транзакции базы темпоральных данных инициируются при завершении каждого тика часов ПРВ. При выполнении любой транзакции актор p может находится в одном из следующих состояний:

- "*Блокирован*",
- "*Ожидает*",
- "*Активен*".

После завершения транзакции актор переходит в состояние "*Блокирован*" до конца текущего тика. В состоянии "*Блокирован*" актор p деактивирован, иными словами – "не используется". Переход актора в очередном тике из состояния "*Блокирован*" в состояние "*Ожидает*" происходит вследствие его активации СУБТД для выполнения транзакции. Состояние "*Ожидает*" является для актора началом выполнения транзакции, в котором проверяется условие готовности перехода актора в состояние "*Активен*". Если условие выполняется, то актор переходит в состояние "*Активен*", в котором реализует процедуру актуализации, соответствующую классу ТД, или процедуру обновления буфера выходного канала актуализированным значением экстернального ТД. При завершении процедуры актор переходит в состояние "*Блокирован*" до окончания текущего тика часов ПРВ.

2.4.2. Транзакции актуализации экзогенных темпоральных данных

Различают следующие транзакции актуализации экзогенных темпоральных данных БТД:

- Актуализация экзогенного датума.
- Актуализация экзогенного импульса.
- Актуализация экзогенной моды.

Актуальное значение экзогенного данного берётся актором из буфера инициативного или пассивного входного канала. В общем случае принятое каналом в буфер значение может соответствовать исходному объекту сложной структуры, представленному в виде битовой последовательности (содержащей в битовом виде значения всех полей структуры), формат которой позволяет соответствующему актору актуализации выполнить операцию *десериализации* (восстановления структуры исходного объекта из битовой последовательности) при обновлении экзогенного данного.

2.4.2.1. Актуализация экзогенного датума

Новое актуальное значение для обновления экзогенного датума $td^1 = \langle \dot{z}, \dot{t}, \dot{\tau} \rangle^1$ принимается от периферийной службы в буфер инициативного входного канала инициированного СУБТД в последний момент периода репрезентативности датума - $\dot{t} + \dot{\tau}$, предшествующий очередному моменту времени $\ddot{t} = \dot{t} + \dot{\tau} + 1$, при наступлении которого текущее значение датума в БТД теряет валидность - $td^0_{datum} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{datum}$. Так как инициатива получения каналом нового значения от периферийной службы исходит от СУБТД, то транзакция обновления буфера канала реализуется по протоколу $proactive_{in}$ (Рис. 4). В соответствии с протоколом $proactive_{in}$ канал $ch_{in}^{proactive_{in}}$ посылает периферийной службе запрос на получение нового значения и интервала репрезентативности датума - пары $\langle \dot{z}, \dot{\tau}_{system} \rangle$, значение $\dot{\tau}_{system}$ периферийная служба предоставляет каналу в единицах относительного системного времени ОСРВ.

Транзакция актуализации актором экзогенного датума d_{in}^{datum} формально представляется в виде:

$$d_{in}^{datum} : ch_{in}^{proactive_{in}} \xrightarrow{\left(\Delta t^{system}_{d_{in}^{datum}}\right)} td^0_{datum}.$$

Выполнение актором d_{in}^{datum} транзакции выглядит как последовательность нахождения актора в одном из трёх состояний: "*Блокирован*", "*Ожидает*", "*Активен*". Если в наступившем

тике экзогенный датум td^1_{datum} является валидным, СУБТД не активирует актор d^{datum}_{in} и он остаётся в состоянии "*Блокирован*". В момент потери датумом валидности - $\ddot{t} = \dot{t} + \dot{\tau} + 1$, актор d^{datum}_{in} из состояния "*Блокирован*" переходит в состояние "*Ожидает*", в котором проверяет получение в буфер входного канала $ch^{proactive_{in}}_{in}$ от периферийной службы обновлённого значения пары - $\langle \ddot{z}, \ddot{\tau}_{system} \rangle$. Если буфер канала не обновился - $\langle \dot{z}, \dot{\tau}_{system} \rangle$, то актор d^{datum}_{in} сразу переходит в состояние "*Блокирован*", и датум $\langle \dot{z}, \dot{t}, 0 \rangle^0_{datum}$ не актуализируется. Если буфер канала обновился, то актор d^{datum}_{in} из состояния "*Ожидает*" переходит в состояние "*Активен*" и в течение интервала системного времени $\Delta t^{system}_{d^{datum}_{in}}$ обновляет в БТД значение экзогенного датума. Так как полученный интервал репрезентативности $\dot{\tau}_{system}$ предоставляется периферийной службой в системном времени ОСРВ, то он преобразуется актором d^{datum}_{in} в целое значение тиков по шкале часов реального времени ПРВ - $time_{1t}$, т.е. $\dot{\tau} := \left\lfloor \frac{\dot{\tau}_{system}}{1t} \right\rfloor$, дробная часть отбрасывается. Метку времени начала периода репрезентативности нового значения экзогенного датума актор d^{datum}_{in} получает по шкале часов ПРВ как $\ddot{t} := time_{1t}$. В общем случае - $\ddot{t} \geq \dot{t} + \dot{\tau} + 1$. Завершив обновление, индекс валидности устанавливается в единицу, и актор d^{datum}_{in} вновь переходит в состояние "*Блокирован*", транзакция завершается.

Заметим, что в нулевой момент времени - $\dot{t} = time_{1t} = 0$, датум является изначально не валидным с неопределённым значением и нулевым интервалом репрезентативности - $td^0_{datum}(0) = \langle \nexists, 0, 0 \rangle^0_{datum}$. Поэтому в нулевой момент времени СУБТД сразу инициирует транзакцию актуализации не валидного датума.

2.4.2.2. Актуализация экзогенного импульса

Утрата валидности экзогенного импульса по истечении интервала репрезентативности является его "естественным" состоянием - $td^0_{pulse} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{pulse}$, а инициатива обновления исходит от периферийной службы. Поэтому входной канал является пассивным и ожидает прихода нового значения импульса. Транзакция актуализация экзогенного импульса актором d^{pulse}_{in} значением из буфера пассивного входного канала, формально представится в виде:

$$d^{pulse}_{in} : ch^{inactive_{in}}_{in} \xrightarrow{\left(\Delta t^{system}_{d^{pulse}_{in}}\right)} td^0_{pulse},$$

Если экзогенный импульс является валидным, то в каждый момент его периода репрезентативности актор актуализации импульса d^{pulse}_{in} находится в состоянии "*Блокирован*". При завершении периода репрезентативности импульса в начале наступившего очередного тика часов ПРВ актор d^{pulse}_{in} переходит в состояние "*Активен*", проверяет поступление в буфер пассивного канала $ch^{inactive_{in}}_{in}$ от периферийной службы пары $\langle \dot{z}, \dot{\tau}_{system} \rangle$ - нового значения $\dot{z}$ и интервала репрезентативности $\dot{\tau}_{system}$ импульса. Если буфер канала не обновился, то актор d^{pulse}_{in} переходит в состояние "*Блокирован*". Если в момент $\dot{t} := time_{1t}$ буфер канала обновился, то актор актуализирует импульс в течение интервала системного времени $\Delta t^{system}_{d^{pulse}_{in}}$ значением $\langle \dot{z}, time_{1t}, \left\lfloor \frac{\dot{\tau}_{system}}{1t} \right\rfloor \rangle^1$, преобразуя интервал репрезентативности $\dot{\tau}_{system}$ из системного времени в целое значение тиков по шкале часов реального времени ПРВ. Индекс валидности

устанавливается в единицу. Завершив обновление, актор d_{in}^{pulse} переходит в состояние "*Блокирован*", оставаясь в нём в течение всего периода репрезентативности импульса до момента потери импульсом валидности, после чего транзакция актуализации импульса повторяется.

Заметим, что в нулевой момент времени $time_{1t} = 0$ импульс изначально является не валидным с неопределённым значением - $td_{datum}^{0}(0) = \langle \not\equiv ,0,0\rangle_{datum}^{0}$.

2.4.2.3. Актуализация экзогенной моды

В отличие от датума и импульса мода $td_{mode}^{1} = \langle \dot{z}, \dot{t}, \not\equiv\rangle_{mode}^{1}$ является валидной в течение неопределённого, но большего нуля, интервала репрезентативности. Момент завершения периода репрезентативности моды определяется спорадически возникающим событием доставки каналом $ch_{in}^{inactive_{in}}$ нового значения, свидетельствующего о завершении интервала репрезентативности текущего значения моды. В этот момент реального времени мода "мгновенно" теряет валидность (принимает значение $td_{mode}^{0} = \langle \dot{z}, \dot{t}, 0\rangle_{mode}^{0}$) и должна актуализироваться:

$$d_{in}^{mode} : ch_{in}^{inactive_{in}} \xrightarrow{\left(\Delta t_{d_{mode}}^{system}\right)} td_{mode}^{0},$$

Транзакция актуализации моды имеет сходство с транзакцией актуализации импульса, но принципиальное отличие в том, что момент завершения периода репрезентативности моды является неопределённым, поэтому состояние " *Блокирован* " для актора d_{mode} становится вырожденным, требующим принудительного завершения для инициации транзакция актуализации моды. Поэтому в начале очередного тика часов ПРВ индекс валидности и интервал репрезентативности моды принудительно сбрасываются в ноль, делая её не доступной для использования в вычислениях. При этом актор d_{mode} переходит в состояние "*Активен*" и проверяет доставку в буфер входного канала $ch_{in}^{inactive_{in}}$ нового значения моды - $\langle z, \not\equiv\rangle$, и либо в течение интервала системного времени $\Delta t_{d_{mode}}^{system}$обновляет и восстанавливает валидность моды $\langle \dot{z}, time_{1t}, \not\equiv\rangle^{1}$, либо просто восстанавливает валидность текущего значения моды.

2.4.3. Транзакции актуализации эндогенных темпоральных данных

В отличие от экзогенных данных, не валидное эндогенное темпоральное данное БТД $td_{out_{class}}^{0} = \langle \dot{z}, \dot{t}, 0\rangle_{out_{class}}^{0}$), ставшее не валидным в момент времени часов ПРВ $\ddot{t}$, актуализируется значением, которое вычисляется актором $e_{class} \in E$ над набором входных темпоральных данных БТД - $\left\{td_{in_{class_1}}^{v_1}, td_{in_{class_2}}^{v_2}, \dots, td_{in_{class_N}}^{v_N}\right\}$. Актор e_{class} вычисляет не только новое значение эндогенного темпорального данного - $\ddot{z}$, но и определяет новый интервал его репрезентативности - $\ddot{\tau}$, и результат должен быть получен одномоментно, то есть в течение тика $\ddot{t}$ часов ПРВ, в котором началась актуализация эндогенного данного. При этом каждый актор e_{class} характеризуется конечным интервалом системного времени $\Delta t_{e_{class}}^{system}$ вычисления нового значения $\ddot{z}$ и нового интервала репрезентативности $\ddot{\tau}$. Очевидно, что $\Delta t_{e_{class}}^{system} \ll 1_t$.

Активируя в момент времени $\ddot{t}$ актор e_{class}, СУБТД тем самым инициирует *транзакцию актуализации* не валидного эндогенного данного $td_{out_{class}}^{0} = \langle \dot{z}, \dot{t}, 0\rangle_{out_{class}}^{0}$. Формально

транзакцию актуализации не валидного эндогенного данного актором $e_{class} \in E$ обозначим в виде:

$$e_{class}: \left\{td^{v_1}_{in_{class_1}}, td^{v_2}_{in_{class_2}}, \dots, td^{v_N}_{in_{class_N}}\right\} \xRightarrow{\left(\Delta t^{system}_{e_{class}}\right)} td^{0}_{out_{class}},$$

где $\left\{td^{v_1}_{in_{class_1}}, td^{v_2}_{in_{class_2}}, \dots, td^{v_N}_{in_{class_N}}\right\}$ - набор входных темпоральных данных БТД, в общем случае разных классов - $class_i \in \{datum, pulse, moda\}$, валидные значения которых используются актором e_{class} для вычисления актуального значения $\ddot{z}$. В каждом тике часов ПРВ выполнение транзакции представляется как последовательное нахождение актора e_{class} в одном из трёх состояний: "*Блокирован*", "*Ожидает*", "*Активен*".

Актор e_{class} находится в состоянии "*Блокирован*", если актуализируемое им темпоральное данное в текущем тике является валидным - $td^{1}_{out_{class}}$, набор входных данных является *темпорально целостным* - $\left\{td^{1}_{in_{class_1}}, td^{1}_{in_{class_2}}, \dots, td^{1}_{in_{class_N}}\right\}$, и ни одно из входящих в него темпоральных данных не обновило своего значения. Актор e_{class} из состояния "*Блокирован*" переводится в состояние "*Ожидает*", если в наступившем тике эндогенное данное является не валидным - $td^{0}_{out_{class}} = \langle \dot{z}, \dot{t}, 0\rangle^{0}_{out_{class}}$.

В состоянии "*Ожидает*" актор e_{class} находится до момента времени, когда набор входных темпоральных данных в любой своей части обновился, и в результате все темпоральные данные набора являются валидными - $\left\{td^{1}_{in_{class_1}}, td^{1}_{in_{class_2}}, \dots, td^{1}_{in_{class_N}}\right\}$, такой набор назовём *темпорально целостным*. В этот момент - $\dot{t} := time_{1t}$, актор e_{class} переходит в состояние "*Активен*", выполняет вычисление новой пары $\langle \dot{z}, \dot{\tau}\rangle$, $\dot{\tau} = IR\left(td^{1}_{out_{class}}(\dot{t})\right)$, и в течение интервала системного времени $\Delta t^{system}_{e_{class}}$ обновляет не валидное значение эндогенного данного. При этом необходимо найти момент завершения периода репрезентативности обновлённого эндогенного данного такой, когда условие темпоральной целостности входного набора $\left\{td^{1}_{in_{class_1}}, td^{1}_{in_{class_2}}, \dots, td^{1}_{in_{class_N}}\right\}$ нарушалось бы в следующий за ним момент. Поэтому период репрезентативности обновлённого значения эндогенного данного $td^{1}_{out_{class}}$ вычисляется по формуле:

$$PR\left(td^{1}_{out_{class}}\right) = \left(\bigcap_{i=1}^{N} PR\left(td^{1}_{in_{class_i}}\right)\right) \bigcap [\dot{t}, \infty] = \left[\dot{t}, \dot{t} + IR\left(td^{1}_{out_{class}}\right)\right] \neq \emptyset,$$

где $\bigcap_{i=1}^{N} PR\left(td^{1}_{in_{class_i}}\right)$ – пересечение периодов репрезентативности $PR\left(td^{1}_{in_{class_i}}\right)$ всех темпоральных данных входного набора $\left\{td^{1}_{in_{class_1}}, td^{1}_{in_{class_2}}, \dots, td^{1}_{in_{class_N}}\right\}$, которое в свою очередь пересекается с частью оси времени - $[\dot{t}, \infty]$, для формирования начала периода репрезентативности нового значения эндогенного данного, равного моменту времени перехода актора e_{class} в состояние "*актив*ен" - $\dot{t}$. В итоге, начало периода репрезентативности $PR\left(td^{1}_{out_{class}}\right)$ будет совпадать с моментом времени $\dot{t} := time_{1t}$ завершения обновления значения эндогенного датума, а момент завершения определяемого периода репрезентативности будет совпадать с ближайшим моментом потери темпоральной целостности набором входных данных $\left\{td^{1}_{in_{class_1}}, td^{1}_{in_{class_2}}, \dots, td^{1}_{in_{class_N}}\right\}$.

Важно отметить, что теоретически возможно получение для обновляемого значения эндогенного данного пустого периода репрезентативности $PR\left(td^1_{out_{class}}(\dot{t})\right) = \emptyset$ с нулевым интервалом - $IR\left(td^1_{out_{class}}(\dot{t})\right) = 0$. Это возможно в том случае, если во входном наборе $\left\{td^1_{in_{class_1}}, td^1_{in_{class_2}}, \dots, td^1_{in_{class_N}}\right\}$ присутствует темпоральное данное, у которого последний тик периода репрезентативности совпадает с моментом активизации актора обновления не валидного эндогенного данного - $\dot{t} := time_{1t}$. Поэтому индекс валидности нового значения следует устанавливать в единицу, только если $IR\left(td^1_{out_{class}}\right) > 0$. В противном случае индекс валидности сохраняет значение ноль. То есть, в общем случае результат актуализации следующий:

$$td^{v(\dot{t})}_{out_{class}} = \begin{cases} td^1_{out_{class}}, IR\left(td^1_{out_{class}}(\dot{t})\right) > 0 \\ td^0_{out_{class}}, IR\left(td^0_{out_{class}}(\dot{t})\right) = 0 \end{cases}.$$

Различают следующие транзакции актуализации эндогенных темпоральных данных БТД:

- Актуализация эндогенного датума.
- Актуализация эндогенного импульса.
- Актуализация эндогенной моды.

2.4.3.1. Актуализация эндогенного датума

Транзакция актуализации не валидного эндогенного датума формально представляется в виде:

$$e_{datum}: \left\{td^{v_1}_{in_{class_1}}, td^{v_2}_{in_{class_2}}, \dots, td^{v_N}_{in_{class_N}}\right\} \xrightarrow{\left(\Delta t^{system}_{e_{class}}\right)} td^0_{out_{datum}}.$$

Набор входных данных актора e_{datum} может содержать темпоральные данные только класса $class_i = datum$. Заметим, что набор входных данных не может содержать мод. Так как интервал репрезентативности моды не определён, то и результат вычисления над ними интервала репрезентативности результата будет неопределённым, а это противоречит определению датума, у которого интервал репрезентативности всегда априори задан. Невозможность нахождения в наборе входных данных импульсов связана с тем, что датум представляет собой данное, теоретически непрерывно изменяющееся во времени, а наличие во входном наборе импульса этому бы противоречило из-за того, что импульс по определению не является непрерывным во времени. А так как актор e_{datum} по определению выполняется только над темпорально целостным набором входных данных - $\left\{td^1_{in_{class_1}}, td^1_{in_{class_2}}, \dots, td^1_{in_{class_N}}\right\}$, то при завершении периода репрезентативности импульса, включённого во входной набор, темпоральная целостность набора прерывается на неопределённый интервал времени, что не соответствует теоретической непрерывности во времени значения темпорального данного класса $datum$.

Так как набор входных данных $\left\{td^{v_1}_{in_{class_1}}, td^{v_2}_{in_{class_2}}, \dots, td^{v_N}_{in_{class_N}}\right\}$ состоит только из одних датумов (экзогенных или эндогенных), то транзакция актуализации эндогенного датума, предполагает последовательное нахождение актора e_{datum} в одном из трёх состояний: "*Блокирован*", "*Ожидает*", "*Активен*".

Актор e_{datum} находится в состоянии "*Блокирован*" до момента потери датумом валидности - $td_{out_{datum}}^{\nu(\dot{t})=0}$. В этот момент СУБТД инициирует транзакцию актуализации, переводя актор состояние "*Ожидает*". Состояние "*Ожидает*" сохраняется до тех пор, пока набор входных темпоральных данных $\left\{td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N}\right\}$ не обновится в наступивший момент времени $\ddot{t} := time_{1t}$, $\ddot{t} \geq \dot{t}$ станет *темпорально целостным*, т.е. в наборе все темпоральные данные стали валидными. В этот момент актор e_{datum} переходит в состояние "*активен*", и в течение ограниченного интервала системного времени $\Delta t_{e_{datum}}^{system}$ вычисляет $\ddot{z}$ и $\ddot{\tau}$, формирует $\langle \ddot{z}, \ddot{t}, \ddot{\tau} \rangle$[1], и обновляет им не валидный датум:

$$e_{datum}: \left\{td_{in_{class_1}}^{1}, td_{in_{class_2}}^{1}, \dots, td_{in_{class_N}}^{1}\right\} \xRightarrow{\left(\Delta t_{e_{datum}}^{system}\right)} td_{out_{datum}}^{1},$$

после чего вновь переходит в состояние "*Блокирован*" до очередной транзакции актуализации.

Заметим, что в нулевой момент времени - в момент старта ПРВ, все эндогенные датумы БТД не являются валидными, и акторы e_{datum} изначально находятся в состоянии "*Ожидает*".

2.4.3.2. Актуализация эндогенного импульса

Транзакция актуализации эндогенного импульса формально представляется в виде:

$$e_{pulse}: \left\{td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N}\right\} \xRightarrow{\left(\Delta t_{e_{pulse}}^{system}\right)} td_{out_{pulse}}^{0},$$

где не валидный выходной импульс $td_{out_{pulse}}^{0}$ актуализируется значением, вычисленным актором e_{pulse} над набором входных темпоральных данных $\left\{td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N}\right\}$, который может состоять:

- либо только из темпоральных данных класса *datum* или *mode*,
- либо из темпоральных данных класса *datum* или *mode*, и может включать в себя только один импульс,
- либо только из одного импульса.

В первом варианте импульс формируется спорадически как результат анализа актором e_{pulse} над набором темпоральных данных класса *datum* или *mode* $\left\{td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N}\right\}$.условия возникновения события, порождающего импульс.

Во втором варианте иметь во входном наборе $\left\{td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N}\right\}$ более одного импульса не имеет смысла, так как практически не возможно достичь для такого набора входных данных темпоральной целостности, если импульсы в наборе будут независимыми, то есть не будут семантически связанными. Иметь в наборе зависимые импульсы также практически не целесообразно, так как они будут семантически эквивалентными.

Третий вариант является вырожденным случаем второго.

Состав входного набора определяет специфику выполнения транзакции актуализации выходного импульса актором e_{pulse}.

Если входной набор процедуры e_{pulse} состоит только из одного импульса - $\left\{td_{in_{pulse_1}}^{\nu_1}\right\}$, то выявив валидный импульс система управления БТД, - $td_{in_{pulse_1}}^{1}$, активирует актор e_{pulse} для актуализации выходного импульса $td_{out_{pulse}}^{0}$, переводя его из состояния "*Блокирован*" сразу в

состояние "*Активен*", который в течение системного времени $\Delta t_{e_{pulse}}^{system}$ формирует новое значение импульса $\langle \ddot{z}, \dot{t}, \dot{\tau} \rangle_{out_{pulse}}^{1}$, у которого момент $\dot{t}$ и интервал репрезентативности $\dot{\tau}$ наследуются от импульса входного набора - $td_{in_{pulse_1}}^{1}$, и актуализирует не валидный эндогенный импульс:

$e_{pulse}: \left\{ td_{in_{pulse_1}}^{1} \right\} \xRightarrow{\left(\Delta t_{e_{pulse}}^{system}\right)} td_{out_{pulse}}^{1}$.

После этого актор переходит в состояние "*Блокирован*" до завершения периода репрезентативности актуализированного импульса.

Отличие в реализации транзакция актуализации эндогенного импульса для входного набора темпоральных данных, включающего в себя кроме импульса и темпоральные данные других классов - $\left\{ td_{in_{pulse_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N} \right\}$ заключается лишь в том, что система управления БТД переводит процедуру e_{pulse} в состояние "*Ожидает*" наступления темпоральной целостности набора входных темпоральный данных, а в результате либо сразу переходит в состояние "*Блокирован*" до конца тика (если входной набор не темпорально целостный), либо переходит в состояние "*Активен*", актуализирует выходной импульс и переходит в состояние "*Блокирован*" до завершения периода репрезентативности нового значения эндогенного импульса - $td_{out_{pulse}}^{1}$.

Особо реализуется транзакция актуализации импульса, если набор входных темпоральный данных $\left\{ td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N} \right\}$ процедуры e_{pulse} не содержит импульса, $class_i \in \{datum, mode\}$. В этом случае результат вычисления, полученный актором e_{pulse} над набором $\left\{ td_{in_{class_1}}^{1}, td_{in_{class_2}}^{1}, \dots, td_{in_{class_N}}^{1} \right\}$ может оказаться пустым - $\emptyset$. То есть актор e_{pulse}, анализируя темпорально целостный набор входных данных либо вырабатывает спорадический результат, свидетельствующий о возникновении некоторого контролируемого события, и обновляет выходной импульс значением - $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{out_{pulse}}^{1}$, либо оставляет не валидное значение импульса без изменения - $td_{out_{pulse}}^{0}$. Если в результате выполнения процедуры e_{pulse} сохраняется не валидное значение импульса, то актор e_{pulse} переводится в состояние "Блокирован" до завершения текущего тика часов ПРВ, а в начале следующего тика актор вновь переводится в состояние "*Ожидает*". Если при этом набор входных данных $\left\{ td_{in_{class_1}}^{\nu_1}, td_{in_{class_2}}^{\nu_2}, \dots, td_{in_{class_N}}^{\nu_N} \right\}$ окажется темпорально целостным, то актор переводится в состояние "*Активен*", анализирует наличие контролируемого события и либо актуализирует выходной импульс новым значением - $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{out_{pulse}}^{1}$, после чего переводится в состояние "*Блокирован*" до завершения периода репрезентативности нового значения импульса, либо формально вырабатывает результат с пустым периодом репрезентативности - $\emptyset$, т.е. событие не произошло, сохраняется не валидное значение, и актор e_{pulse} переводится в состояние "*Блокирован*" до завершения текущего тика. Если же в состоянии "*Ожидает*" выясняется, что набор входных данных в текущем тике не имеет темпоральной целостности, то актор сразу переводится из состояния "*Ожидает*" в состояние "*Блокирован*", в котором остаётся до завершения текущего тика. Таким образом актор e_{pulse} будет выполняться каждый тик часов ПРВ, пока не дождётся темпоральной целостности входного набора данных и контролируемое

событие не будет выявлено, после чего в течение системного времени $\Delta t_{e_{pulse}}^{system}$ формирует новое значение эндогенного импульса - $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{out_{pulse}}^{1}$, где $\dot{t}$ – текущий тик часов ПРВ, $\dot{\tau}$ – заданный актором e_{pulse} интервал репрезентативности, соответствующий контролируемому событию, и переходит в состояние "*Блокирован*" до завершения периода репрезентативности нового значения импульса.

Заметим, что в нулевой момент времени - в момент старта ПРВ, эндогенный импульс не является валидным и транзакция актуализации начинается в нулевом тике с перехода актора e_{pulse} из состояния "*Блокирован*" в состояние "*Ожидает*". Кроме того, в отличие от датума валидность импульса принудительно обнуляется в момент завершения СУБТД начатого цикла актуализации БТД, так как использованный импульс теряет актуальность.

2.4.3.3. Актуализация эндогенной моды

Транзакцию актуализации эндогенной моды формально представим в виде:

$$e_{mode}: \left\{ td_{in_{class_1}}^{v_1}, \dots, td_{in_{class_N}}^{v_N} \right\} \xrightarrow{\left(\Delta t_{e_{mode}}^{system}\right)} td_{out_{mode}}^{0}, class_i \in \{datum, mode\}, i = \overline{1, N}.$$

В отличие от датума или импульса, у которых момент завершения периода репрезентативности априори задан, момент завершения периода репрезентативности эндогенной моды не определён и семантически. При наступлении очередного тика эндогенная мода автоматически теряет валидность, активируя актор e_{mode}, который в каждом тике часов ПРВ контролирует спорадическое возникновение событием обновления набора $\left\{ td_{in_{class_1}}^{v_1}, \dots, td_{in_{class_N}}^{v_N} \right\}$, и при его возникновении, обновляет эндогенную моду. Таким событием может быть, например, обновление в наборе $\left\{ td_{in_{mode}}^{v} \right\}$ единственной экзогенной моды. Кроме этого, событие обновления может быть следствием обновления в наборе темпоральных данных $\left\{ td_{in_{class_1}}^{v_1}, \dots, td_{in_{class_N}}^{v_N} \right\}$, который, например, может состоять из темпоральных данных типа $mode$ или $datum$, или даже – только $datum$, если в наборе произошло обновление датума. Иными словами, в общем случае могут быть разные варианты состава набора входных темпоральных данных и событий его обновления. Именно при возникновении в очередном тике контролируемого актором e_{mode} события обновления набора завершается период репрезентативности текущего значения выходной эндогенной моды. В противном случае актор e_{mode} просто восстанавливает валидность текущего значения выходной эндогенной моды.

Заметим, что на актуальность значения импульса, ранее вычисленного с использованием значения моды, утратившей валидность в период репрезентативности импульса, это не влияет. Это следует из того, что валидность значения импульса связана непосредственно с моментом возникновения соответствующего семантике импульса события, а сформированный интервал репрезентативности импульса задаёт лишь предельное время реакции системы на это событие. Поэтому импульсу не требуется актуализация.

В связи с тем, что анализ спорадического события завершения периода репрезентативности эндогенной моды на наборе $\left\{ td_{in_{class_1}}^{v_1}, \dots, td_{in_{class_N}}^{v_N} \right\}$ необходимо контролировать каждый тик часов ПРВ, то транзакция актуализации начинается с того, что в

начале каждого очередного тика $\dot{t}$ индекс валидности и интервал репрезентативности выходной моды сбрасываются в ноль - $td^{0}_{out_{mode}}$, формально делая моду не валидной и инициируя тем самым переход процедуры e_{mode} из состояния "*Блокирован*" в состояние "*Ожидает*", в котором она находится (возможно вырождено) до момента наступления темпоральной целостности входного набора $\left\{td^{v_1}_{in_{class_1}}, \dots, td^{v_N}_{in_{class_N}}\right\}$. Если условие темпоральной целостности выполняется, то актор e_{mode} переводится в состояние "Активен", в течение системного времени $\Delta t^{system}_{e_{mode}}$ формирует новое значение моды - $\langle \dot{z}, \dot{t}, \not\equiv \rangle^{1}_{out_{pulse}}$, и переходит в состояние "*Блокирован*" до завершения текущего тика. Очевидно, что в большинстве случаев обновление значения моды будет носить фиктивный характер.

Теоретически моды в БТД изначально должны быть проинициализированы, и до старта часов ПРВ уже иметь валидные значения.

2.4.4. Транзакция экспорта экстернальных данных

Транзакция экспорта экстернального данного инициируется в результате завершения актором в текущем тике его актуализации, после чего СУБТД сразу активирует связанный с обновлённым экстернальным данным актор $t_{class} \in T$, который должен обновить результатом актуализации td^{v}_{class} буфер выходного канала, осуществляющего его доставку (экспорт) периферийной службе, связанной с этим каналом. Заметим, что если экстернальное данное не обновилось, то буфер выходного канала актором t_{class} не обновляется, и в текущем тике экспорт данного каналом не осуществляется.

Важно отметить, что если значение экстернального данного является объектом сложной структуры, содержащей динамические поля, то актор экстернального данного должен при его копировании в буфер выходного канала выполнить *сериализацию*[5] объекта.

Формально транзакция обновления актором t_{class} буфера пассивного выходного канала $ch^{inactive_{out}}_{out}$ значением экстернального данного td^{v}_{class}, реализуемая в течение системного времени $\Delta t^{system}_{t_{class}}$, выражается в виде:

$$t_{class}: td^{v}_{class} \xRightarrow{\left(\Delta t^{system}_{t_{class}}\right)} ch^{inactive_{out}}_{out}.$$

В общем случае в рамках транзакции актор t_{class} последовательно находится в одном из трёх состояний:

"*Блокирован*" – актор t_{class} деактивирован;

"*Ожидает*" – актор t_{class} активирован и проверяет условие обновления экстернального данного td^{v}_{class};

"*Активен*" – актор t_{class} активирован, и в течение конечного интервала системного времени $\Delta t^{system}_{p_{class}}$ обновляет буфер выходного канала текущим значением экстернального данного.

---

[5] **Сериализация** - процесс преобразования содержимого структуры данных объекта с динамическими полями в битовую последовательность (содержащую в битовом виде значения всех полей структуры), формат которой обеспечивает возможность **десериализации -** восстановление структуры данных исходного объекта из битовой последовательности, принятой периферийной службой.

Транзакция начинается с перехода актора t_{class} в текущем тике из состояния "*Блокирован*" в состояние "*Ожидает*", в котором актор проверяет условие необходимости обновления буфера выходного канала текущим значением экстернального данного. Условие выполняется, если состояние экстернального данного изменилось: значение экстернального данного обновилось, либо изменилась его валидность. Тогда актор переходит в состояние "*Активен*" и обновляет буфер выходного канала. В противном случае актор сразу переходит в состояние "*Блокирован*", и буфер канала не обновляется в текущем тике.

Если при завершении транзакции буфер выходного канала обновился, и канал в этот момент обслуживает запрос от периферийной службы на получение сообщения с обновлённым значением буфера, то канал в качестве ответа на запрос посылает сообщение периферийной службе. Если буфер канала обновился, но запрос от периферийной службы отсутствует, то новое значение буфера не экспортируется. Если выходной канал обслуживает запрос от периферийной службы, но буфер в текущем тике не обновился, то обслуживание запроса пролонгируется в следующий тик. Это типичное состояние для выходного канала, связанного с экстернальным импульсом или модой, обновление которых происходит спорадически.

Замечание. Пролонгация обслуживания каналом запроса от периферийной службы на получение значения экстернального датума в ряде случаев может быть следствием нарушения режима реального времени.

2.4.5. Репликация распределённой базы темпоральных данных

2.4.5.1. Распределённая база темпоральных данных

Системы реального времени, например в области промышленной автоматизации, могут строится на основе локальной или промышленной сети, связывающей отдельные подсистемы СРВ в единую распределённую систему. Следствием этого является то, что и программная составляющая распределённой СРВ должна строиться как распределённое приложение реального времени. В этом случае глобальное ПРВ представляется как конечное множество распределённых по узлам вычислительной сети параллельно функционирующих составных частей (фрагментов): $\text{ПРВ} = \{\Phi_1, \Phi_2, \dots, \Phi_N\}$. Важно, однако, то что каждый фрагмент $\Phi_i \in \text{ПРВ}$ строится по одной и той же принципиальной схеме (см. **Ошибка! Источник ссылки не найден.**). В результате и логически единая для распределённого ПРВ база темпоральных данных теперь представляется в виде конечного множества её распределённых фрагментов - $TDB_{\text{ПРВ}} = \{TDB_{\Phi_1}, TDB_{\Phi_2}, \dots, TDB_{\Phi_N}\}$, каждый со своими локальными множествами темпоральных данных, акторов, входных и выходных каналов.

Если разные фрагменты Φ_i и Φ_j имеют не пустые пересечения множеств темпоральных данных БТД фрагментов Φ_i и Φ_j: $TDB_{\Phi_i} \cap TDB_{\Phi_j} \neq \emptyset$, $i \neq j$, то они содержат идентичные темпоральные данные, находящиеся в разных фрагментах распределённой $TDB_{\text{ПРВ}}$. Их идентичность выражается в том, что некоторое изначально актуализируемое локально, например во фрагменте TDB_{Φ_i}, темпоральное данное является экстернальным данным по отношению к фрагменту TDB_{Φ_j}, в котором находится идентичное ему экзогенное данное, теоретически являющееся в каждый момент реального времени его полной копией. И наоборот, изначально актуализируемое локально темпоральное данное во фрагменте TDB_{Φ_j}, является экстернальным данным по отношению к фрагменту TDB_{Φ_i}, в котором находится идентичное

ему экзогенное данное. В результате обязанностью СУБТД таких фрагментов $\Phi_i, \Phi_j \in \text{ПРВ} = \{\Phi_1, \Phi_2, \dots, \Phi_N\}$ становится репликация[6] в реальном времени часов ПРВ значений идентичных темпоральных данных, во всех распределённых фрагментах TDB_{Φ_i} базы темпоральных данных $TDB_{\text{ПРВ}}$: $TDB_{\Phi_i} \in \{TDB_{\Phi_1}, TDB_{\Phi_2}, \dots, TDB_{\Phi_N}\}$.

Если фрагмент TDB_{Φ_i} содержит реплицируемые экстернальные данные, которым во фрагменте TDB_{Φ_j} соответствуют идентичные экзогенные данные, то во фрагмент TDB_{Φ_i} добавляются соответствующие выходные каналы, а во фрагмент TDB_{Φ_j} - соответствующие классу экзогенного данного входные каналы для приёма копий реплицируемых данных фрагмента TDB_{Φ_i}. В итоге распределённые фрагменты ПРВ Φ_i и Φ_j по отношению друг к другу, выступают в роли особых объектов окружения, с которыми фрагменты взаимодействуют посредством специальных периферийных служб – *служб репликации* распределённых темпоральных данных. Специфика их заключается в том, что взаимодействующие друг с другом службы репликации данных фрагментов Φ_i и Φ_j совместно реализуют протокол логической "склейки" буферов выходных каналов фрагмента TDB_{Φ_i}, которые обновляются в реальном времени акторами репликации значений экстернальных данных (*оригиналов*), с буферами инициативных входных каналов фрагмента TDB_{Φ_j}, используемыми для актуализации в реальном времени соответствующих экзогенных данных (*дубликатов*). Аналогично реплицируются значения экстернальных данных (*оригиналов*) фрагмента TDB_{Φ_j} и соответствующих им экзогенных данных (*дубликатов*) фрагмента TDB_{Φ_i}.

2.4.5.2. Транзакция репликации распределённых темпоральных данных

Подмножество акторов репликации экстернальных данных (*оригиналов*) фрагмента TDB_{Φ_i}, реализующих транзакции обновления буферов "склеенных" выходных каналов, обозначим $C_{TDB_{\Phi_i}} \subset P_{TDB_{\Phi_i}}$. Подмножество акторов, обновляющих экзогенные данные (*дубликаты*) фрагмента TDB_{Φ_j} значениями из буферов "склеенных" входных каналов, обозначим $C_{TDB_{\Phi_j}} \subset P_{TDB_{\Phi_j}}$. Обозначим $c_{out}^{class} \in C_{TDB_{\Phi_i}}$ актор репликации экстернального данного (*оригинала*) класса $class$ во фрагменте TDB_{Φ_i}, а актор репликации соответствующего экзогенного данного (*дубликата*) во фрагменте TDB_{Φ_j} обозначим $c_{in}^{class} \in C_{TDB_{\Phi_i}}$. Тогда транзакция репликации распределённого во фрагментах TDB_{Φ_i} и TDB_{Φ_j} темпорального данного td_{class}^{v} любого класса формально представляется в виде пары параллельно выполняемых акторов репликации: c_{out}^{class} в TDB_{Φ_i} и c_{in}^{class} в TDB_{Φ_j}, реализующих совместно протокол логической "склейки" буфера пассивного выходного канала $ch_{out}^{inactive_{out}}$ фрагмента TDB_{Φ_i} и буфера инициативного входного канала $ch_{in}^{proactive_{in}}$ фрагмента TDB_{Φ_j}:

[6] Репликация – это процесс, под которым понимается одномоментное копирование в реальном времени часов $\text{ПРВ}_{\text{СРВ}}$ результата актуализации экстернального данного в одном фрагменте распределённой БТД в его дубликат – экзогенное данное, в другом или множестве других распределённых фрагментах.

$$\begin{cases} c_{out}^{class} : td_{class}^{v} \xRightarrow{\left(\Delta t_{c_{out}^{class}}^{system}\right)} ch_{out}^{inactive_{out}} \quad - \text{ в } TDB_{\Phi_i}, \\ c_{in}^{class} : ch_{in}^{proactive_{in}} \xRightarrow{\left(\Delta t_{c_{in}^{class}}^{system}\right)} td_{class}^{v} \; - \text{ в } TDB_{\Phi_j} \end{cases}$$

Во фрагменте TDB_{Φ_i} актор c_{out}^{class} находится в состоянии "*Блокирован*" до события завершения в текущем тике часов ПРВ транзакции актуализации реплицируемого экстернального данного td_{class}^{v}, после чего СУБТД инициирует транзакцию репликации, активируя актор репликации c_{out}^{class}, который переходит в состояние "*Ожидает*" для проверки факта обновления экстернального данного td_{class}^{v}.

Если обновление произошло, то актор c_{out}^{class} переходит в состояние "*Активен*", и в течение интервала системного времени $\Delta t_{c_{out}^{class}}^{system}$ обновляет содержимое буфера "склеенного" выходного канала $ch_{out}^{inactive_{out}}$ значением td_{class}^{v}, после чего переходит в состояние "*Блокирован*".

Если в результате транзакции актуализации экстернального реплицируемого данного td_{class}^{v} обновление не случилось, то из состояния "*Ожидает*" актор c_{out}^{class} сразу переходит в состояние "*Блокирован*", и буфер "склеенного" выходного канала не обновляется. Это типично для реплицируемых импульса или моды. Если реплицируемым является актуализированный экстернальный датум, то типичным результатом актуализации является его обновление, а в результате и обновление актором c_{out}^{class} буфера соответствующего выходного канала $ch_{out}^{inactive_{out}}$, "склеенного" с входными каналами реплицируемых экзогенных данных распределённых фрагментов. Если буфер выходного канала обновился, то значение буфера передаётся службой репликации во все "склеенные" с ним инициативные входные каналы $ch_{in}^{proactive_{in}}$ распределённых фрагментов.

Во фрагментеTDB_{Φ_j} актор c_{in}^{class} находится в состоянии "*Блокирован*" до момента потери реплицируемым экзогенным данным td_{class}^{v} валидности в текущем тике, в результате чего СУБТД сразу активирует актор c_{in}^{class}, он переходит в состояние "*Ожидает*", в котором проверяет факт обновления "склеенного" буфера инициативного входного канала $ch_{in}^{proactive_{in}}$. Если буфер обновился, то актор c_{in}^{class} сразу переходит в состояние "*Активен*", в течение интервала системного времени $\Delta t_{c_{in}^{class}}^{system}$ обновляет реплицируемый дубликат td_{class}^{v} значением полученным из буфера "склеенного" входного канала $ch_{in}^{proactive_{in}}$, и переходит в состояние "*Блокирован*". Если буфер не обновился, то актор c_{in}^{class} сразу переходит в состояние "*Блокирован*".

3. Схема управления базой темпоральных данных

Ранее были отдельно рассмотрены транзакции управления темпоральными данными различных категорий и классов, при необходимости инициируемые СУБТД в начале каждого тика часов ПРВ. В результате активизируются и параллельно выполняются акторы актуализации темпоральных данных и акторы обновления буферов выходных каналов значениями актуализированных экстернальных данных. При этом порядок инициирования транзакций влияет на корректность результатов выполнения активированных акторов. Это связано с тем, что для акторов, актуализирующих эндогенные данные, условием получения корректного результата их выполнения является темпоральная целостность набора входных темпоральных данных актора. разделяемым ресурсом параллельно имеет значение и на множестве акторов P устанавливается отношение частичного порядка, в соответствии с которым всё множество акторов P разбивается на непересекающиеся нумерованные подмножества, а на множестве нумерованных подмножеств устанавливается отношение доминирования.

4. Режимы жёсткого и мягкого реального времени

4.1. Темпоральные прецеденты

При выполнении в очередном тике часов ПРВ инициированных транзакций актуализации не валидных экзогенных датумов может оказаться так, что из-за не своевременного обновления, по какой-то причине, буфера соответствующего входного канала не все активированные акторы p_i перейдут из состояния "*Ожидает*" в состояние "*Активен*", а некоторые сразу вернутся в состояние "*Блокирован*", оставив не валидные экзогенные данные не обновлёнными.

Возможно также, что в наступившем тике обновилось значительное количество экзогенных данных различных классов, что как следствие, привело к инициации транзакций актуализации значительного числа зависящих от них эндогенных данных, а также транзакций экспорта или репликации обновившихся экстернальных данных. В результате, может оказаться так, что среди параллельно активированных в текущем тике акторов p_i, перешедших в состояние "*Активен*", не все успеют выполниться одномоментно из-за ограниченности вычислительных ресурсов, что приводит к невозможности одномоментного выполнения и завершения некоторых транзакции в текущем тике.

Начиная с момента *системного* времени $t = \ddot{t} \cdot 1t$ и в течение всего последующего интервала системного времени, не актуализированное актором темпоральное данное будет не валидным - $td^0(t) = \langle \dot{z}, \dot{t}, 0 \rangle^0$, и его значение не будет доступно акторам для использования в вычислениях. Если в последствии актуализация произойдёт в момент $time_{1t} > \ddot{t}$, т.е. не одномоментно, а с задержкой, то для ПРВ это является нарушением режима реального времени. На Рис. 7 иллюстрируется изменение валидности значения непрерывного во времени темпорального данного $td^v(t) = \langle \dot{z}, \dot{t}, \dot{\tau} \rangle^v$ в системном времени t, с отмеченными тиками часов ПРВ.

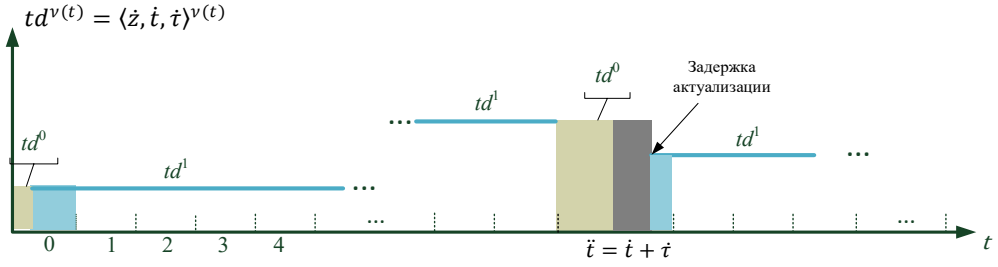


Рис. 7 Изменение валидности темпорального данного в системном времени

Системы реального времени, относительно возможности продолжения работы ПРВ при возникновении темпоральных прецедентов, делят на системы "жёсткого" и "мягкого" реального времени. Для СРВ жёсткого реального времени возникновение В ПРВ темпорального прецедента интерпретируется как фатальная ошибка, после чего осуществляется его аварийное завершается. В системах мягкого реального времени для ограниченного набора темпоральных данных допускаются редко возникающие темпоральные прецеденты с априори заданными ограниченными интервалами задержки актуализации, в течение которых ПРВ продолжает функционировать в режиме деградации.

Не зависимо от причины, не обновления не валидных данных или не завершения выполнения хотя бы одной из всех инициированных СУБТД транзакций в течение текущего тика это интерпретируется в ПРВ как нарушение режима реального времени или *темпоральный прецедент*.

4.1. Режим жёсткого реального времени

Если СРВ в течение всего времени работы абсолютно не допускает в ПРВ темпоральных прецедентов, то такой режим работы СРВ называют режимом *жёсткого реального времени*. Для таких СРВ условие одномоментности выполнения всех инициированных транзакций актуализации темпоральных данных БТД не может нарушаться в течение всего времени работы системы. При фиксированной вычислительной производительности – C_{const}, выполнение режима жёсткого реального времени требует установки часам ПРВ величины тика $1_t > 1_t^{min}$– минимальное значение тика часов ПРВ при котором вероятность темпоральных прецедентов гарантированно сводится к нулю. Если для заданной вычислительной производительности C_{const} оказалось, что найденное минимальное значение тика часов ПРВ равное $1_t^{min} > 1_t^{lim}$ – заданное предельное время реакции системы на все контролируемые события, то необходимо либо оптимизировать вычислительные потребности ПРВ, не меняя C_{const}, либо находить соответствующую 1_t^{lim} производительность C_{min}, и использовать вычислительную систему с производительностью $C_{const} > C_{min}$ – расчётная минимальная производительность вычислительной системы обеспечивающая работу СРВ в режиме жёсткого реального времени.

Если данные от периферийных служб своевременно доставляются канальными процедурами в буферы входных каналов, то очевидной причиной темпоральных прецедентов будет нехватка вычислительных ресурсов для всегда своевременного завершения всех активированных в тике акторов. Формально условие, гарантирующее функционирование ПРВ в режиме жёсткого реального времени в вычислительной системе с ограниченной производительностью C_{const} можно выразить следующим образом.

Положим, что для каждого актора $p_i \in P$ известна длительность конечного интервала системного времени его выполнения в монопольном режиме в вычислительной системе с производительностью C_{const}: в состоянии "*Ожидает*" - $\Delta t_{p_i^{\text{Ожидает}}}^{system}(C_{const})$, в состоянии "*Активен*" - $\Delta t_{p_i^{\text{Активен}}}^{system}(C_{const})$. Допустим, что для любого актора $p_i \in P$ при выполнении транзакции затраты системного времени на его переключение из состояния "*Блокирован*" в состояние "*Ожидает*", из состояния "*Ожидает*" в состояние "*Активен*" и из состояния "*Активен*" в состояние "*Блокирован*" пренебрежимо малы, т.е. можно считать равными нулю. Тогда очевидно, что для выполнения условия одномоментного выполнения всех инициированных в тике $\dot{t}$ транзакций, в каждый момент системного времени $t \in \dot{t}$ должно выполняться неравенство:

$$\forall \dot{t} = \left\lfloor \frac{t}{1_t} \right\rfloor \in [0,T]: \sum_{i=1}^{N} \delta_{p_i^{\text{Ожидает}}}(t) \cdot \Delta t_{p_i^{\text{Ожидает}}}^{system}(C) + \delta_{p_i^{\text{Активен}}}(t) \cdot \Delta t_{p_i^{\text{Активен}}}^{system}(C) < 1_t,$$

где:

N – количество всех акторов множества P ;

$[0,T]$ – общее время работы по часам ПРВ;

$$\delta_{p_i^{\text{Ожидает}}}(t) = \begin{cases} 1, & p_i \text{ в состоянии "Ожидает"} \\ 0, & p_i \text{ в состоянии "Активен"или "Блокирован"} \end{cases};$$

$$\delta_{p_i^{\text{Активен}}}(t) = \begin{cases} 1, & p_i \text{ в состоянии "Активен"} \\ 0, & p_i \text{ в состоянии "Ожидает"или "Блокирован"} \end{cases};$$

$t \in \dot{t}$ - текущий момент относительного системного времени.

Иными словами, активированные в текущем тике $\dot{t}$ акторы всех инициированных транзакций должны перейти в состояние "*Блокирован*" до окончания этого же тика. Если это условие не выполняется, то останется хотя бы один актор в состоянии "*Активен*", выполнение которого выходит за пределы текущего тика $\dot{t}$, и возникает темпоральный прецедент.

4.2. Режим мягкого реального времени

Возникающие темпоральные прецеденты могут оказывать разной степени отрицательное влияние на эффективность функционирования СРВ. Темпоральные прецеденты с одними данными БТД могут быть совершенно не допустимыми, так как приводят к потери контроля системы над критическими параметрами (импульсы, моды), что как правило вызывает её аварийную остановку. Однако для некоторых СРВ редкие устранимые во времени темпоральные прецеденты могут кратковременно снижать эффективность функционирования системы, сохраняя её приемлемую работоспособность. В этом случае прекращение работы СРВ крайне нежелательно. Например, кратковременная потеря контроля над некоторыми датумами может и не приводить к существенному ухудшению технологического процесса. Поэтому целесообразно избегать аварийной остановки работы системы, и предоставить ей возможность работы в режиме кратковременной функциональной деградации. Возможность работы СРВ в режиме кратковременной функциональной деградации обуславливается возможностью появления и кратковременного использования в вычислениях устаревших (*нечётких*) темпоральных данных БТД. Кратковременная задержка транзакций обновления датумов, которые обычно во множестве присутствуют в БТД, и возможность использования акторами их нечётких значений может не оказывать столь существенного влияния на эффективность функционирования СРВ как задержка с использованием импульса или тем более не своевременное обновление моды. При этом важно иметь оценку предела устаревания во времени значения темпорального данного, до которого нечёткое значение можно считать *условно валидным* для использования акторами в качестве исходных данных. Установка для каждого темпорального данного БТД своего предела условной валидности зависит от степени влияния использования акторами его устаревшего значения на функционирование СРВ. Для данных класса *mode*, например, использование их не валидного значения вообще недопустимо. Для данных класса *pulse* задержку использования значения импульса можно рассматривать как допустимую нечёткость окончания заданного интервала репрезентативности импульса (допустимая задержка доставки значения импульса в буфер выходного канала). Для датума допустимое увеличение заданного интервала репрезентативности можно устанавливать в зависимости от известной степени влияния их нечётких значений на практическую значимость результатов их использования для актуализации эндогенных данных БТД. Очевидно, что "правильно" настроенный режим использования акторами нечётких данных БТД повышает "устойчивость" работы ПРВ к вдруг спорадически возникающим пиковым нагрузкам выполнения значительного числа инициированных транзакций (актуализации темпоральных

данных, экспортирования экстернальных данных периферийным службам и репликации данных в распределённых фрагментах БТД), предотвращая аварийную остановку СРВ.

Если допускается возникновение контролируемых СУБТД темпоральных прецедентов, то такой режим работы ПРВ называется режимом *мягкого реального времени*. В режиме мягкого реального времени в случае возникновения темпорального прецедента возможны следующие практически значимые варианты действий СУБТД по управлению не завершившимся циклом актуализации БТД, прежде чем в новом тике начать новый цикл актуализации БТД:

1. В момент окончания тика целостно завершить транзакции, уже инициированные в текущем цикле актуализации БТД, и не инициировать новые.
2. Продолжить выполнение текущего цикла актуализации БТД до целостного завершения всех транзакций.

В первом варианте выполнение цикла актуализации БТД прерывается приходом сигнала нового тика. Инициация новых транзакций отменяется, а новый цикл актуализации БТД откладывается до завершения начатых транзакции, когда выполнение акторов завершается переходом из состояние "*Активен*" в состояние "*Блокирован*". После этого СУБТД начинает новый цикл актуализации БТД в наступившем новом тике в обычном порядке. В итоге отменённые в предыдущем тике транзакции заново инициируются в новом тике.

Во втором варианте приход сигнала нового тика не прерывает выполнение текущего цикла актуализации БТД до его полного завершения, после чего СУБТД ждёт сигнала нового тика. В обоих вариантах возникает необходимость принятия решения по использованию акторами нечётких значений с контролируемой степенью *условной валидности* для актуализации не валидных темпоральных данных.

4.2.1. Оценка деградации темпоральных данных

Режим мягкого реального времени допускает, что кратковременная задержка обновления некоторых формально потерявших валидность темпоральных данных класса *datum* или даже *pulse* не всегда является основанием для отказа временного использования их деградирующих значений акторами, если это не может привести к фатальным последствиям работы СРВ. Однако использование в вычислениях деградирующих значений должно быть ограничено во времени "разумным" пределом. Для этого необходим формальный метод оценки степени практической значимости кратковременного использования акторами в наборе входных данных своевременно не обновлённых значений темпоральных данных. При этом рассматриваются только импульсы или датумы. Моды по определению используются только валидные.

Для оценки степени деградации темпоральных данных вместо целочисленного индекса валидности $\nu \in \{0,1\}$, принимающего только два значения 0 или 1, вводится вещественный *коэффициент валидности* $\tilde{\nu} \in [0,1]$, характеризующий степень валидности (практической пригодности) использования акторами устаревшего значения импульса или датума при выполнении транзакции. Очевидно, что в режиме мягкого реального времени понятие коэффициента валидности расширяет понятие индекса валидности темпорального данного в режиме жёсткого реального времени. Если коэффициент валидности $\tilde{\nu} = 1$, то темпоральное данное является валидным. Если коэффициент валидности $\tilde{\nu} = 0$, то темпоральное данное не

является валидным и не может использоваться в вычислениях, как и в режиме жёсткого реального времени. Если коэффициент валидности $0 \ll \tilde{v} < 1$, то темпоральное данное является нечётким во времени, а величина коэффициента валидности (степень валидности) характеризует практическую значимость использования в вычислениях нечёткого во времени значения темпорального данного.

Естественно полагать, что возможность использования в вычислениях деградирующих с каждым пропущенным тиком часов ПРВ нечётких темпоральных данных должна ограничиваться. Поэтому каждому ТД устанавливается нижний предел коэффициента валидности: $\tilde{v} \geq \tilde{v}_{lim} \gg 0$. При этом выбор для темпорального данного $\tilde{v}_{lim}$ является результатом эвристической оценкой степени его влияния на эффективность работы системы в условиях ограниченной деградации темпоральных данных в БТД.

Коэффициент валидности $\tilde{v}$ должен отражать уменьшение степени валидности значения темпорального данного $\langle \dot{z}, \dot{t}, 0 \rangle_{class}^{\tilde{v}_{lim} \leq \tilde{v} \leq 1}$ при увеличении запаздывания обновления (для датума) или запаздывания использования в вычислениях (для импульса). При этом можно интуитивно полагать, что, чем больше в период валидности темпорального данного $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{class}^{1}$ был интервал валидности $\dot{\tau}$, тем вероятнее, что устаревшее значение $\dot{z}$ сохранить практическую пригодность его использования в очередном тике за пределами периода репрезентативности. Если обозначить $\tilde{t}$ текущее показание часов ПРВ, то справедливо полагать, что при увеличении задержки обновления - $\left(\tilde{t} - (\dot{t} + \dot{\tau})\right) \to \infty$, коэффициент валидности $\tilde{v}(\tilde{t}) \to 0$ и, следовательно, практическая значимость значения $\dot{z}$ для темпоральных вычислений должна монотонно снижаться. При этом разумно полагать очевидно, что чем меньше был интервал репрезентативности $\dot{\tau}$, тем быстрее условно валидное (нечёткое) значение $\dot{z}$ должно терять практическую значимость с каждым тиком запаздывания.

4.2.1.1. Деградация и использование нечёткого экзогенного датума

Формально результат выполнения транзакции актуализации экзогенного датума с учётом возможности неуспешного завершения, представится в виде:

$$d_{datum} : ch_{in}^{inactive} \xRightarrow{\left(\Delta t_{d_{datum}}^{system}\right)} td_{datum}^{\tilde{v}(\tilde{t}) \leq 1}.$$

Если в момент завершившегося периода репрезентативности экзогенного датума активированный актор d_{datum} в состоянии "*Ожидает*" обнаруживает, что буфер инициативного входного канала обновился, то актор d_{datum} переходит в состояние "*Активен*", обновляет экзогенный датум значением, полученным из буфера, возвращается в состояние "*Блокирован*", и транзакция актуализации завершается успешно.

Деградация значения экзогенного датума $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{datum}^{\tilde{v}(\tilde{t})}$ во времени начинается, когда активированный актор d_{datum} в состоянии "*Ожидает*" обнаруживает, что буфер инициативного входного канала не обновился, актор возвращается в состояние "*Блокирован*", а экзогенный датум сохраняет в текущем тике устаревшее значение. При этом степень валидности текущего значения экзогенного датума $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{datum}^{\tilde{v}(\tilde{t})}$ с заданным $\tilde{v}_{lim}$ при завершении транзакции его актуализации в текущем тике $\tilde{t}$ вычисляется по формуле:

$$\tilde{v}(\tilde{t}) = \begin{cases} 1, & \tilde{t} \in [\dot{t}, \dot{t} + \dot{\tau}]; \\ \dfrac{\dot{\tau}}{\tilde{t} - \dot{t}}, & (\tilde{t} > \dot{t} + \dot{\tau}) \wedge \left(\dfrac{\dot{\tau}}{\tilde{t} - \dot{t}} \geq \tilde{v}_{lim}\right); \\ 0, & (\tilde{t} > \dot{t} + \dot{\tau}) \wedge \left(\dfrac{\dot{\tau}}{\tilde{t} - \dot{t}} < \tilde{v}_{lim}\right). \end{cases}$$

Из формулы следует, что для своевременно обновлённого экзогенного датума значение его коэффициента валидности $\tilde{v}(\tilde{t}) = 1$ в течение его очередного периода репрезентативности Но если своевременное обновление не произошло, то по мере удаления текущего момента времени часов ПРВ $\tilde{t}$ от момента завершение периода репрезентативности устаревшего данного - $(\dot{t} + \dot{\tau})$, его степень валидности становится меньше единицы и стремится к нулю: $\tilde{v}(\tilde{t}) \to 0$ при $(\tilde{t} - \dot{t}) \to \infty$. При этом, чем продолжительнее был интервал репрезентативности $\dot{\tau}$ устаревшего значения $\dot{z}$, тем медленнее его степень валидности во времени будет стремится к своему нижнему пределу $\tilde{v}_{lim}$. Практически это означает, что чем больше был интервал репрезентативности $\dot{\tau}$ устаревшего экзогенного датума $td_{datum}^{\tilde{v}(\tilde{t})<1}$, тем дольше, при запаздывании с обновлением, его устаревающее значение $\dot{z}$ будет сохранять практическую значимость для темпоральных вычислений.

При успешном обновлении экзогенного датума актор d_{datum} будет находиться в состоянии "*Блокирован*" до истечения периода репрезентативности $[\dot{t}, \dot{t} + \dot{\tau}]$, а после его завершения в новом тике часов ПРВ актор d_{datum} переходит из состояния "*Блокирован*" в состояние "*Ожидает*", в котором проверяет наличие в буфере входного канала $ch_{in}^{inactive}$ нового значения. Если буфер не обновлён, то актор d_{datum} переходит в состояние "*Блокирован*" до конца текущего тика и транзакция завершается деградацией экзогенного датума, его степень валидности становится меньше единицы. Это приводит к тому, что при $\tilde{v}(\tilde{t}) < 1$ транзакция актуализации будет инициироваться в каждом последующем тике запаздывания обновления буфера канала $ch_{in}^{inactive}$. При этом, если текущая степень валидности экзогенного датума - $\tilde{v}(\tilde{t}) \geq \tilde{v}_{lim}$, то нечёткое значение датума $\dot{z}$ может быть использовано в вычислениях в качестве входного данного другими активированными акторами. После успешного обновления экзогенного датума в очередном тике он вновь становится валидным - $\tilde{v} = 1$, а его актуализация возобновится только по истечении периода репрезентативности.

Заметим, что теоретически из-за длительной задержки доставки в буфер входного канала нового значения, затраченное время на попытки актуализации экзогенного датума, может в итоге превысить интервал репрезентативности полученного нового значения. В этом случае новое значение экзогенного датума сразу же окажется нечётким.

4.2.1.2. Деградация и использование нечёткого экзогенного импульса

Деградация во времени значения обновлённого экзогенного импульса, полученного из буфера пассивного входного канала начинается, когда текущий тик часов ПРВ превысил момент завершения его периода репрезентативности, а импульс ещё не был использован акторами в качестве входного данного. Это может быть связано либо с запаздыванием его поступления в буфер входного канала, либо с задержкой его использования акторами в качестве входного данного из-за перегрузки вычислительной системы.

Транзакция актуализации экзогенного импульса штатно инициируется каждый тик, если в наступившем тике коэффициент валидности экзогенного импульса $\tilde{v}(\tilde{t}) = 0$. Актор d_{pulse}

переходит из состояния "*Блокирован*" в состояние "*Ожидает*", в котором он проверяет доставку в буфер канала $ch_{in}^{inactive_{in}}$ нового значения импульса. Если буфер канала $ch_{in}^{inactive_{in}}$ не обновился, то актор d_{pulse} из состояния "*Ожидает*" возвращается в состояние "*Блокирован*", а экзогенный импульс сохраняет своё устаревшее не валидное значение $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{pulse}^{0}$. Если же буфер канала обновился, то актор обновляет экзогенный импульс значением из буфера. Если период репрезентативности полученного из буфера нового значения ещё не истёк, то обновлённое значение экзогенного импульса будет валидным. Если же из-за задержки доставки в буфер канала нового значения его период репрезентативности уже истёк, то обновлённый экзогенный импульс получит деградирующее нечёткое значение. Поэтому в режиме мягкого реального времени актуализация экзогенного импульса с заданным $\tilde{\nu}_{lim}$ формально представится в виде:

$$d_{pulse} : ch_{in}^{inactive_{in}} \xrightarrow{\left(\Delta t_{d_{pulse}}^{system}\right)} td_{pulse}^{\tilde{\nu}(\tilde{t})\leq 1}.$$

В дальнейшем степень валидности значения экзогенного импульса $\langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{pulse}^{\tilde{\nu}(\tilde{t})}$ в момент оценки актором в состоянии "*Ожидает*" возможности его использования в качестве входного данного в текущем тике $\tilde{t}$ будет так же, как и для экзогенного датума, вычисляться по формуле:

$$\tilde{\nu}(\tilde{t}) = \begin{cases} 1, & \tilde{t} \in [\dot{t}, \dot{t} + \dot{\tau}]; \\ \dfrac{\dot{\tau}}{\tilde{t} - \dot{t}}, & (\tilde{t} > \dot{t} + \dot{\tau}) \wedge \left(\dfrac{\dot{\tau}}{\tilde{t} - \dot{t}} \geq \tilde{\nu}_{lim}\right); \\ 0, & (\tilde{t} > \dot{t} + \dot{\tau}) \wedge \left(\dfrac{\dot{\tau}}{\tilde{t} - \dot{t}} < \tilde{\nu}_{lim}\right). \end{cases}$$

Заметим, что актор d_{pulse} будет находиться в состоянии "*Блокирован*", пока $\tilde{\nu}(\tilde{t}) \geq \tilde{\nu}_{lim}$. Это означает, что практически нет оснований ожидать в ближайшее время поступления в буфер входного канала нового спорадического события. И лишь когда в очередном тике окажется, что $\tilde{\nu}(\tilde{t}) = 0$, актор d_{pulse} будет вновь в каждом тике переходить в состояние "*Ожидает*".

Заметим, что в отличие от датума коэффициент валидности импульса $\tilde{\nu}(\tilde{t})$ принудительно обнуляется СУБТД в момент завершения начатого цикла актуализации БТД, так как использованный импульс теряет актуальность.

4.2.1.3. Деградация и использование нечёткого эндогенного датума

Транзакцию актуализации не валидного эндогенного датума $td_{datum}^{\tilde{\nu}(\tilde{t})} = \langle \dot{z}, \dot{t}, \dot{\tau} \rangle_{datum}^{\tilde{\nu}(\tilde{t})<1}$, с заданным для него $\tilde{\nu}_{lim}$, формально представим в виде:

$$e_{datum} : \left\{ td_1|_{class_1}^{\tilde{\nu}_1(\tilde{t})\geq\tilde{\nu}_{lim}}, td_2|_{class_2}^{\tilde{\nu}_2(\tilde{t})\geq\tilde{\nu}_{lim}}, \dots, td_N|_{class_N}^{\tilde{\nu}_N(\tilde{t})\geq\tilde{\nu}_{lim}} \right\}_{in_{in}} \xrightarrow{\left(\Delta t_{e_{datum}}^{system}\right)} td_{datum}^{\tilde{\nu}(\tilde{t})<1}.$$

Заметим, что для данных класса $mode$, возможно входящих в набор входных данных актора e_{datum} коэффициент валидности должен быть строго равен единице. Поэтому во входном наборе актора нечёткими могут быть только датумы, при чём со степенями валидности не меньшими, чем $\tilde{\nu}_{lim}$.

Транзакция актуализации эндогенного датума $td_{datum}^{\tilde{\nu}(\tilde{t})}$ инициируется, когда в очередном тике его коэффициент валидности становится меньше единицы - $\tilde{\nu}(\tilde{t}) < 1$. В этот момент актор e_{datum} переходит из состояния "*Блокирован*" в состояние "*Ожидает*" для оценки нечёткой целостности набора входных темпоральных данных актора e_{datum} (данные в нём либо валидные, либо нечёткие с коэффициентом валидности $\tilde{\nu}_i(\tilde{t}) \geq \tilde{\nu}_{lim}$). Это требование вытекает из эвристики, что эндогенный датум, с нижним пределом коэффициента валидности равным $\tilde{\nu}_{lim}$, практически не целесообразно обновлять, если во входном наборе присутствовал хотя бы один датум с коэффициентом валидности меньше $\tilde{\nu}_{lim}$. А вот набор входных темпоральных данных $\left\{td_1|_{class_1}^{\tilde{\nu}_1(\tilde{t})\geq\tilde{\nu}_{lim}}, td_2|_{class_2}^{\tilde{\nu}_2(\tilde{t})\geq\tilde{\nu}_{lim}}, \dots, td_N|_{class_N}^{\tilde{\nu}_N(\tilde{t})\geq\tilde{\nu}_{lim}}\right\}_{in}$ определяется как *условно темпорально целостный*.

Если актор e_{datum} в состоянии "*Ожидает*" оценивает набор входных данных как условно темпорально целостный, то он переходит из состояния "*Ожидает*" в состояние "*Активен*". При завершении выполнения он обновляет эндогенный датум значением со степенью валидности равной $\tilde{\nu}(\tilde{t}) = min\{\tilde{\nu}_1(\tilde{t}), \tilde{\nu}_2(\tilde{t}), \dots, \tilde{\nu}_N(\tilde{t})\} \geq \tilde{\nu}_{lim}$.

Если окажется, что $\tilde{\nu}(\tilde{t}) < 1$, то эндогенный датум в текущем тике $\ddot{t}$ обновляется нечётким значением $\langle \ddot{z}, \ddot{t}, 0\rangle_{datum}^{\tilde{\nu}_{lim}\leq\tilde{\nu}(\tilde{t})<1}$ с нулевым интервалом репрезентативности. Актор e_{datum} переходит в текущем тике $\ddot{t}$ в состояние "*Блокирован*", и остаётся в нём лишь до конца этого тика, а в новом тике опять переходит в состояние "*Ожидает*", пытаясь вновь в очередном тике актуализировать эндогенный датум до валидного значения, когда $\tilde{\nu}(\tilde{t}) = 1$.

Если окажется, что входной набор является темпорально целостным - $\tilde{\nu}(\tilde{t}) = 1$, то транзакция актуализации эндогенного датума завершается формированием в текущем тике $\ddot{t}$ нового валидного значения $\langle \ddot{z}, \ddot{t}, \ddot{\tau}\rangle_{datum}^{1}$ с интервалом репрезентативности $\ddot{\tau}$, сформированным как для валидного датума. В этом случае после завершения обновления в текущем тике актор e_{datum} переходит в состояние "*Блокирован*" и остаётся в нём до завершения сформированного периода репрезентативности.

Для полученного актором нечёткого эндогенного датума $\langle \ddot{z}, \ddot{t}, 0\rangle_{datum}^{\tilde{\nu}_{lim}\leq\tilde{\nu}(\tilde{t})<1}$ старый интервал валидности интерпретируется как неопределённый (используется для вычисления $\tilde{\nu}(\tilde{t})$), что заставляет в начале очередного тика переводить процедуру e_{datum} в состояние "*Ожидает*" и пытаться актуализировать его. Если в состоянии "*Ожидает*" набор входных данных процедуры e_{datum} не оказывается нечётко темпорально целостным, то актор просто делает $\tilde{\nu}(\tilde{t}) = 0$ и переходит в состояние "*Блокирован*", остаётся в нём до завершения текущего тика. В результате эндогенный датум становиться не валидным. Основываясь на ранее введённой эвристике оценки условной валидности, выразим формально уменьшение значения коэффициента валидности $\tilde{\nu}(\tilde{t})$, обусловленное запаздыванием обновления эндогенного датума $\langle \dot{z}, \dot{t}, 0\rangle_{datum}^{\tilde{\nu}(\tilde{t})}$ с коэффициентом валидности последнего обновления $\tilde{\nu}(\dot{t})$, в последующие моменты времени $\tilde{t} \geq \dot{t}$ в следующем виде:

$$\tilde{\nu}(\tilde{t}) = \begin{cases} \tilde{\nu}(\dot{t}) = min\{\tilde{\nu}_1(\dot{t}), \tilde{\nu}_2(\dot{t}), \dots, \tilde{\nu}_N(\dot{t})\}, & \tilde{t} = \dot{t}; \\ \dfrac{\tilde{\nu}(\dot{t})}{\tilde{t} - \dot{t}}, (\tilde{t} > \dot{t}) \wedge \left(\dfrac{\tilde{\nu}(\dot{t})}{\tilde{t} - \dot{t}} \geq \tilde{\nu}_{lim}\right); & \\ 0, \quad (\tilde{t} > \dot{t}) \wedge \left(\dfrac{\tilde{\nu}(\dot{t})}{\tilde{t} - \dot{t}} < \tilde{\nu}_{lim}\right); & \end{cases} .$$

В представленной формуле $\dot{t}$ – это время последнего обновления датума, а $\tilde{t} \geq \dot{t}$. Из формулы следует, что $\tilde{v}(\dot{t}+1) = \tilde{v}(\dot{t})$. Это связано с тем, что часы ПРВ, по которым фиксируется время в ПРВ, являются дискретными, и это не позволяет в рамках предложенной эвристики с большей точностью вычислять коэффициент валидности эндогенного датума при запаздывании его обновления на один тик.

Заметим, что появление нечётких датумов обуславливается задержками обновления экзогенных датумов или спорадическим возникновением пиковых вычислительных нагрузок ядра ПРВ на вычислительную систему. При этом в режиме пиковых вычислительных нагрузок будет происходить "веерное отключение" активности некоторого количества акторов, так как их наборы входных данных будут терять нечёткую темпоральную целостность. В результате ресурсы вычислительной системы будут как-то во времени перераспределяться между акторами в течение деградации работы СРВ. Если всякого рода причины задержек будут носить временных характер, то валидность данных в БТД и штатный режим работы ПРВ будут автоматически восстанавливаться.

4.2.1.4. Деградация и использование нечёткого эндогенного импульса

По определению в наборе входных данных актора e_{pulse}, актуализирующего не валидный эндогенный импульс $td^0_{pulse} = \langle \dot{z}, \dot{t}, 0 \rangle^0_{pulse}$, может присутствовать не более одного входного импульса (или ни одного, если значение импульса генерируется внутри БТД). Поэтому формально транзакцию актуализации эндогенного импульса td^0_{pulse} с использованием в наборе входных данных нечёткого импульса с коэффициентом валидности не ниже предела $\tilde{v}_{lim}$ представим в виде:

$$e_{pulse}: \left\{ td^{\tilde{v}_1(\tilde{t}) \geq \tilde{v}_{lim}}_{pulse_1}, td^1_{class_2}, \dots, td^1_{class_N} \right\}_{in} \xrightarrow{\left(\Delta t^{system}_{e_{pulse}}\right)} td^0_{pulse}.$$

В отличие от эндогенного датума, интервал репрезентативности эндогенного импульса ограничивает время реакции СРВ на соответствующее импульсу спорадическое событие. Поэтому, при обновлении эндогенного импульса в текущем тике его период репрезентативности наследуется от входного импульса (экзогенного или эндогенного). При этом значение коэффициента валидности входного импульса меньшее единицы уже свидетельствует о запаздывании реакции ПРВ. Это учитывается при формировании коэффициента валидности значения зависящего от него эндогенного импульса в очередном тике запаздывания, а в итоге экстернального импульса, отправляемого в буфер выходного канала для отправки управляющего данного периферийной службе.

Так как значение экстернального импульса используется системой для управления объектом, то естественно полагать, что входящие в набор входные датумы или моды в момент перехода процедуры актуализации e_{pulse} в состояние "*Активен*", должны быть строго валидными (коэффициенты валидности равны единице). Это вытекает из эвристики, что использовать для управления объектом значение экстернального импульса, вычисленное на основе нечёткого датума или моды, практически не имеет смысла. Заметим также, что в очередном тике запаздывания теоретически не имеет смысла повторно обновлять нечёткий эндогенный импульс с коэффициентом валидности не достигшим нижнего предела - $\tilde{v}(\tilde{t}) \geq \tilde{v}_{lim}$, так как использование его значения в пределах допустимого запаздывания сохраняет практическую актуальность для управления объектом.

Актор актуализации обновляет не валидный эндогенный импульс td^{0}_{pulse} сформированным новым нечётким значением $\langle \tilde{z}, \dot{t}, \dot{\tau} \rangle^{\tilde{\nu}(\tilde{t}) \geq \tilde{\nu}_{lim}}_{pulse}$, где $\tilde{z}$ – значение вычисленное актором e_{pulse}, $\dot{t}$ и $\dot{\tau}$ – характеристики периода репрезентативности, унаследованные от входного экзогенного импульса $td^{\tilde{\nu}_1(\tilde{t}) \geq \tilde{\nu}_{lim}}_{pulse_1}$. Эти характеристики будут в дальнейшем использоваться для вычисления коэффициента валидности значения не валидного эндогенного импульса, по формуле:

$$\tilde{\nu}(\tilde{t}) = \begin{cases} 1, & \tilde{t} \in [\dot{t}, \dot{t} + \dot{\tau}]; \\ \dfrac{\dot{\tau}}{\tilde{t} - \dot{t}}, & (\tilde{t} > \dot{t} + \dot{\tau}) \wedge \left(\dfrac{\dot{\tau}}{\tilde{t} - \dot{t}} \geq \tilde{\nu}_{lim} \right); \\ 0, & \left(\tilde{t} > t \dot{+} \dot{\tau} \right) \wedge \left(\dfrac{\dot{\tau}}{\tilde{t} - \dot{t}} < \tilde{\nu}_{lim} \right); \end{cases} .$$

После обновления эндогенного импульса - $\langle \tilde{z}, \dot{t}, \dot{\tau} \rangle^{\tilde{\nu}(\tilde{t}) \geq \tilde{\nu}_{lim}}_{pulse}$, актор e_{pulse} переходит в состояние "*Блокирован*" до окончания текущего тика, и остаётся в нём в последующих тиках, пока $\tilde{\nu}(\tilde{t}) \geq \tilde{\nu}_{lim}$ (до обнуления коэффициента валидности в очередном тике). После чего актор e_{pulse} в каждом очередном тике вновь переходит в состояние "*Ожидает*" для проверки темпоральной целостности своего входного набора темпоральных данных.

Заключение

Рассмотренная онтология программирования основана на универсальном представлении любого приложения реального как системы управления в темпе собственных часов реального времени актуализацией базы темпоральных данных - ядро приложения, с которым связаны периферийные службы, обеспечивающие взаимодействие ПРВ со своим окружением. При этом ядро и состав периферийных служб ПРВ очевидным образом масштабируются в зависимости от сложности выполняемых функций и многообразия объектов внешнего окружения.

Рассмотренный состав акторов актуализации БТД и организация их выполнения как параллельных вычисленных процедур над базой темпоральных данных основаны на концепции оперативного контроля валидности темпоральных данных и допустимости их использования в вычислениях в режиме реального времени. Введённый для оценки актуальности темпоральных данных коэффициент валидности и допущение возможности использования акторами в качестве исходных данных значений своевременно не обновлённых темпоральных данных позволило формально определить критерий работы ПРВ в режиме жёсткого или мягкого реального времени. В режиме мягкого реального времени допускается использование акторами в вычислениях в качестве исходных данных условно актуальные (нечёткие) темпоральные данные БТД с заданными минимально допустимыми значениями коэффициентов валидности. Это позволяет явно управлять "пластичностью" ПРВ, устанавливая, в случае запаздывания обновления, каждому темпоральному данному БТД предел "деградации" их значений в виде минимального коэффициента валидности меньше единицы. Это допускает возможность функционирования ПРВ с нечёткими темпоральными данными, полагая, что их ограниченное во времени использование не приведёт к аварийному завершению его работы. При этом если полагать, что значения коэффициентов валидности для всех темпоральных данных БТД никогда не могут быть меньше единицы, то это автоматически определяет для ПРВ режим жёсткого реального времени.

Дальнейшие главы учебного пособия посвящены описанию программных средств системного API[7] современных POSIX[8]-ориентированных ОСРВ, на примере операционной системы QNX Neutrino, которые позволяют эффективно использовать описанную онтологию программирования приложений реального времени как для централизованных, так и распределённых систем реального времени, работающих на основе локальных сетей.

ЧАСТЬ 2. БАЗОВЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ РЕАЛЬНОГО ВРЕМЕНИ

В данном разделе в качестве базовых средств программирования приложений реального времени описываются средства API ОСРВ QNX Neutrino (или QNX 6), далее для ссылки на данную операционную систему будем использовать короткое название – QNX [9-17].

Разработка процессной структуры ПРВ начинается с создания программных модулей (исполняемых файлов), составляющих основу для запуска параллельных процессов. Свойства исполняемых файлов, сформированные программистом при их создании, наследуются запускаемыми процессами. Поэтому знание специфических особенностей создания и хранения программных файлов в файловой системе QNX, формирования свойств создаваемых файлов, их наследования процессами при запуске и влияния на взаимодействие параллельных процессов имеет важное значение для программирования ПРВ.

5. Файловое пространство QNX

В QNX учёт файлов организован в виде древовидной структуры (дерева), называемой файловым *пространством*. QNX позволяет объединять файлы файловых систем различных ОС, находящиеся на подключаемых устройствах внешней памяти, в единое файловое пространство. Единое дерево файлового пространства, такое, каким его видит пользователь системы, может строиться из отдельных файловых систем в общем случае разных типов, которые могут располагаться на различных устройствах внешней памяти и иметь различные внутренние организации. Поэтому файловое пространство QNX в общем случае не является однородным [11].

5.1. Организация файлового пространства QNX

Корнем дерева файлового пространства QNX является *корневой каталог*, имеющий имя </>. Поэтому полное (*абсолютное*) имя любого файла, на каком бы внешнем носителе он не находился, начинается с </>.

Для каждой запущенной программы (процесса) ОС формирует уникальный системный дескриптор управления, в котором, помимо прочего, ведёт два атрибута, связывающих процесс с файловым пространством – атрибут, указывающий на каталог, который устанавливается процессу в качестве корневого, и атрибут, указывающий на каталог, который устанавливается процессу в качестве текущего рабочего. В связи с этим процесс может адресоваться к файлу по имени либо в абсолютном, либо в относительном формате. Имя файла состоит из

[7] API (от англ. Application Programming Interface - программный интерфейс) - это множество системных вызовов ОСРВ, используя которые можно создавать приложения для этой операционной системы.

[8] POSIX (от англ. Portable Operating System Interface - портативный интерфейс операционной системы) - стандарт IEEE 1003.1, определяющий языковой интерфейс между прикладными программами (наряду с оболочками командной строки и служебными интерфейсами) и операционной системой.

последовательности компонентов – локальных имён, разделённых символами '/', принадлежащих вложенным каталогам или файлам. Каталог, содержащий в себе локальное имя считается для него родительским. Последовательность имён, предшествующая в имени файла последнему локальному имени, считается префиксом имени файла. Абсолютное имя файла начинается с символа '/', обозначающего корневой каталог. Например, /user/bin/sh является абсолютным именем файла sh, а /user/bin/ - его префикс. Если имя файла не начинается с символа '/', то оно обозначает путь в файловой системе от текущего каталога процесса до указанного файла. Такое имя считается относительным. Например, указав имя файла как mydir/test1.c, следует полагать, что в неком текущем для процесса каталоге имеется имя mydir каталога, родительского для имени исходного модуля test1.c.

Для удобства навигации каждый каталог всегда содержит две скрытых системных жёстких связей, ссылающихся на каталоги, в которых данный каталог находится. Первая связь имеет локальное имя ".". Она ссылается на свой родительский каталог. Вторая ссылка имеет локальное имя "..". Она ссылается на родительский каталог своего родительского каталога. Исключение составляет корневой каталог. Для него имя "." означает то же, что и имя "..". Они позволяют при локализации файла перемещаться по дереву из текущего каталога наверх. Например, относительное имя ./mydir/test1.c ссылается на файл, родительским каталогом которого является mydir, находящийся в каталоге, родительском для текущего каталога. А относительное имя ../mydir/test1.c ссылается на файл, родительским каталогом которого является `mydir`, находящийся в каталоге, содержащем каталог, родительский для текущего каталога, родительским каталогом является каталог, родительский для текущего каталога.

Поиск файла по имени состоит в последовательном просмотре каталогов, указанных в префиксе, и поиске очередного локального имени.

5.2. Базовая структура корневого каталога

Изначально файловое пространство QNX не является пустым (состоящим из одного пустого корневого каталога). Корневой каталог имеет исходную базовую структуру, состоящую из системных каталогов и файлов, содержащих информацию, обеспечивающую работу ОС и её администрирование. Нарушение этой структуры может привести к неработоспособности системы или отдельных её компонентов. Базовая структура корневого каталога QNX включает в себя следующие основные системные каталоги:

- /bin,
- /dev,
- /etc,
- /lib,
- /root,
- /fs,
- /home,
- /usr,
- /var,
- /tmp,
- /x86.

В каталоге /bin находятся наиболее часто употребляемые команды и утилиты системы, как правило, общего пользования.

Каталог /dev предназначен для хранения системных файлов, обеспечивающих взаимодействие с физическими или виртуальными устройствами. Каталог может содержать подкаталоги, группирующие файлы устройств одного типа.

В каталоге /etc находятся системные конфигурационные файлы, многие утилиты администрирования, а также скрипты инициализации системы.

В каталоге /lib находятся библиотечные файлы среды программирования на языке C/C++.

Каталог /root является каталогом системного администратора с именем root. При входе пользователя в систему под этим именем этот каталог становится его текущим каталогом.

Каталог /fs предназначен для автоматического монтирования файловых систем встроенных устройств внешней памяти (обычно жёстких дисков) к файловому пространству при загрузке ОС.

Каталог /home по умолчанию используется администратором для создания в нём домашних каталогов пользователей при их регистрации в системе.

В каталоге /usr находятся подкаталоги различных сервисных подсистем, исполняемые файлы утилит QNX, заголовочные файлы и т.д.

Каталог /var предназначен для хранения временных файлов различных сервисных подсистем.

Каталог /tmp предназначен для хранения временных файлов, необходимых для работы различных подсистем QNX, а также пользователей системы.

В каталоге /x86 содержатся средства, обеспечивающие разработку программ на языке C/C++ для исполнения на базе платформы intel. В нем, в частности, находится утилита-компилятор qcc.

5.3. Монтирование файловых систем

Как правило, приложения не работают с блочными устройствами напрямую (как с физическим носителем). Полагается, что в каждом физическом разделе блочного устройства (например, жёсткого диска) содержится файловая система некоторого типа. Для доступа к её содержимому она должна быть встроена в дерево корневой файловой системы QNX в виде вновь создаваемого каталога. Эта операция встраивания называется подключением или монтированием файловой системы устройства. После монтирования доступ к файловой системе устройства осуществляется в виде доступа к содержимому данного каталога.

При загрузке QNX автоматически подключает файловые системы, находящиеся в разделах НЖМД, помещая их в каталог /fs в виде подкаталогов с автоматически формируемыми именами. Вход в такой подкаталог приводит к попаданию в корень соответствующей файловой системы. Однако, прежде чем сможет состояться работа с файлами на устройствах с мобильными носителями (CD-ROM, НГМД, флэш-память и т.п.), файловая система соответствующего устройства с установленным на нём мобильным носителем должна быть "вручную" подключена к дереву файлового пространства QNX. Процедура подключения называется монтирование файловой системы устройства.

Монтирование производится командой <mount>, где указывается тип файловой системы (из списка типов файловых систем, известных QNX: dos, qnx4, cd и т.д.), полное имя файла устройства (зарегистрированного в QNX) и полное имя формируемого каталога,

ассоциируемого с монтируемой файловой системой. Например, для монтирования файловой системы типа dos на НГМД (ему соответствует в QNX файл устройства с именем - /dev/fd0) необходимо выполнить в окне терминала следующую команду shell:

mount –t dos /dev/fd0 <полное имя каталога>

В итоге в файловом пространстве появится каталог с заданным командой полным именем. Например, если монтируемой файловой системе дать полное имя /home/A:, то команда монтирования примет вид

mount –t dos /dev/fd0 /home/A:

При успешном выполнении в каталоге /home появляется подкаталог с именем <A:>, вход в который будет приводить к входу в файловую систему дискеты.

Если нужно монтировать дискету с файловой системой qnx4 (тип файловой системы QNX), то вместо dos нужно написать qnx4

mount –t qnx4 /dev/fd0 /home/B:

Файл устройства CD-ROM имеет имя - /dev/cd0. Команда для монтирования CD-ROM и создания каталога файловой системы устройства с именем cd0 в каталоге fs будет иметь вид:

mount –t cd /dev/cd0 /fs/cd0

Если необходимость в мобильном носителе отпала, его можно демонтировать и соответствующий каталог в файловом пространстве будет аннулирован. Например, для демонтирования НГМД нужно выполнить команду

umount /dev/fd0

или

umount /home/A:

5.4. Типы файлов

QNX более тонко определяет понятие файла и его имени. Имя файла рассматривается как атрибут файловой системы, косвенно связанный с некоторым набором данных на диске, который не имеет имени как такового. Каждый такой набор данных (файл) имеет связанные с ним метаданные (хранящиеся в индексных дескрипторах - inode), содержащие все характеристики файла и позволяющие операционной системе управлять выполнением операций, заказанных прикладной задачей: открыть файл, прочитать или записать данные, создать или удалить файл. В частности, метаданные содержат указатели на дисковые блоки хранения данных файла.

Имя файла в файловой системе рассматривается как ссылка на его метаданные, в то время как метаданные не содержат сведений об имени файла.

В QNX существуют 6 типов файлов, различающихся по функциональному назначению и действиям операционной системы при выполнении тех или иных операций над файлами:

- Обычный файл
- Связь
- Каталог
- Именованный канал
- Сокет
- Файл устройства

5.4.1. Обычный файл

Обычный файл представляется операционной системой как файл, содержащий просто последовательность байтов. Интерпретация содержимого такого файла производится исключительно прикладной программой (приложением). К таким файлам относятся и файлы исполняемых программ.

5.4.2. Связь

Кроме имени файла для ссылки на его метаданные могут использоваться также так называемые связи. Связь в QNX является аналогом ярлыка файловой системы ОС Windows. Она позволяет поставить в соответствие одному файлу несколько различных имён (псевдонимов), размещая их затем в различных местах файловой системы.

Связь может быть организована с помощью так называемых жёстких ссылок ("жёсткие связи") и символических или мягких ссылок ("символические связи").

При создании для некоторого файла дополнительного имени с помощью жёсткой связи в каталог помещается новый элемент, который ссылается на тот же файл. Для обеспечения механизма жёстких связей операционная система использует специальный системный файл /.inodeas, в котором, в частности, ведётся счётчик ссылок на файл. При создании очередной жёсткой связи счётчик ссылок увеличивается на единицу. Жёсткие связи могут быть помещены в различные каталоги, находящиеся на одном и том же физическом носителе (одном разделе жёсткого диска). Создание ещё одного имени файла (жёсткой связи) осуществляется с помощью команды <ln>. Нельзя создавать жёсткие связи для каталогов. При удалении одной из жёстких связей реально будет удалён только соответствующий элемент каталога, а счётчик ссылок на файл будет уменьшен на единицу. Как только счётчик достигнет нуля, то файл и соответствующие ему атрибуты управления будут физически уничтожены (при этом файл должен иметь статус "закрыт").

В отличие от жёсткой связи символическая связь реализуется в виде системного текстового файла, содержащего имя указываемого файла. Отличительным свойством символической связи является то, что с её помощью можно создавать дополнительные имена для любого файла (в том числе каталога), находящегося, в общем случае, на другом физическом носителе (например, в другом разделе диска или на другом узле сети). Возможность создания символических связей для каталогов создаёт опасность бесконечных циклов. Поэтому число переходов по символическим связям ограничено значением системной переменной SYMLOOP_MAX, определённой в файле <limits.h>.

5.4.3. Каталог

Каталог — это системный файл, который отличается от обычного тем, что интерпретируется операционной системой как набор записей определённой структуры, которые называют элементами каталога. Каждый элемент каталога связывает имя некоторого файла со служебной информацией о нем, включающей ссылку на место физического хранения данных. Любая задача, имеющая право на чтение каталога, может прочесть его содержимое, но только ОС имеет право на запись в каталог. Штатные средства просмотра содержимого файловой системы по умолчанию не показывают файлы (в том числе и каталоги), имена которых начинаются с точки ("."). Такие файлы называют "скрытыми", и обычно в них содержится системная информация.

5.4.4. Именованный канал

Именованный канал (FIFO) - этот тип файлов относят к средствам взаимодействия процессов, они используются для передачи данных между процессами в форме сообщений.

5.4.5. Сокеты

Сокеты — этот тип файлов относят к средствам доступа к сети TCP/IP.

5.4.6. Устройства

Файл устройства обеспечивает доступ к физическому или виртуальному устройству, зарегистрированному в QNX. Для взаимодействия с устройствами кроме драйверов им ставятся в соответствие файлы устройств. Программы обмениваются данными с устройствами через файлы устройств. Устройства разделяют на два типа:

- *Символьные* (*байт-ориентированные*) устройства читают и записывают данные в виде последовательного потока байтов. Сюда входят последовательные и параллельные порты, накопители на магнитной ленте, терминалы и звуковые карты.
- *Блочные* (*блок-ориентированные*) устройства читают и записывают данные блоками фиксированного размера. Блочные устройства предоставляют прямой доступ к своим данным. Примером блочного устройства является накопитель на жёстком или гибком диске.

Работа с файлами устройств осуществляется путём использования стандартных библиотечных функций ввода-вывода. Программы могут открывать файлы устройств, читать из них данные и осуществлять запись в них точно так же, как если бы это были обычные файлы.

Все файлы устройств, известных системе, содержатся в каталоге /dev. Имена этих файлов стандартизированы. Для доступа к символьному устройству достаточно открыть соответствующий файл устройства как обычный файл и осуществлять чтение/запись традиционным образом. Например, если к первому параллельному порту подключён принтер, то распечатать файл document.txt можно, направив его непосредственно на устройство /dev/lp0, используя команду копирования файлов <cat>:

cat document.txt > /dev/lp0

Чтобы эта команда завершилась успешно, необходимо иметь право записи в файл принтера.

Послать устройству данные из программы также не сложно. В приведённом ниже фрагменте программы с помощью низкоуровневых функций ввода-вывода содержимое буфера направляется в устройство /dev/lp0:

```
int fd=open("/dev/lp0", O_WRONLY);
write(fd, buffer, buffer_length);
close(fd);
```

5.4.7. Виртуальные устройства

В QNX есть ряд специальных символьных устройств, которым не соответствуют никакие аппаратные компоненты. Эти устройства являются виртуальными.

5.4.7.1. Устройство /dev/null

Устройство /dev/null служит двум целям. Поглощает любые данные, направляемые в устройство. В тех случаях, когда выводные данные программы не нужны, в качестве выходного файла назначают устройство /dev/null, например,

```
# verbose_command > /dev/null
```

При чтении из устройства /dev/null всегда возвращается признак конца строки (файла). Если открыть /dev/null с помощью функции open() и попытаться прочесть данные из него с помощью функции read(), то функция вернёт 0 байтов. При копировании содержимого файла /dev/null будет создан пустой файл нулевой длины:

```
# cp /dev/null empty_file
# ls -l empty_file
-rw-rw----   1 ivanov  ivanov     0 Mar 12  15:27 empty_file
```

5.4.7.2. Устройство /dev/zero

Устройство /dev/zero ведёт себя так, как если бы оно было файлом бесконечной длины, заполненным одними нулями. Сколько бы данных ни запрашивалось из этого файла, они всегда предоставляются в необходимом количестве.

Файл /dev/zero удобно использовать в функциях выделения памяти, которые отображают этот файл в память, чтобы инициализировать её нулями.

5.4.7.3. Устройство /dev/full

Устройство /dev/full ведёт себя так, как если бы оно было файлом, в котором нет свободного места. Операция записи в этот файл завершается ошибкой, и в переменную errno помещается код, свидетельствующий о том, что устройство переполнено.

Этот файл удобен для проверки того, как программа будет вести себя в случае, если при записи в файл возникает нехватка места.

5.4.7.4. Устройства генерирования случайных чисел

Специальные устройства /dev/random и /dev/urandom предоставляют доступ к средствам генерирования случайных чисел, входящим в QNX. Эти устройства для получения случайных чисел используют внешний "источник хаоса". Замеряя задержки между действиями пользователя, в частности нажатиями клавиш и перемещениями мыши, устройства способны генерировать непредсказуемый поток действительно случайных чисел. Получить доступ к этому потоку можно путём чтения из устройств /dev/random и /dev/urandom.

Разница между устройствами заключается в том, что если попытаться прочесть большое количество случайных чисел из устройства /dev/random и при этом нем выполнять никаких пользовательских действий (не нажимать клавиши, не перемещать мышь и т.п.), то случайные числа заканчиваются и операция чтения блокируется. Только когда пользователь проявит какую-то активность, система сгенерирует новые случайные числа и разблокирует операцию чтения. В противоположность этому операция чтения из устройства /dev/urandom никогда не блокируется. Если в системе заканчиваются случайные числа, используется криптографический алгоритм, чтобы сгенерировать псевдослучайные числа из последней цепочки случайных чисел.

6. Пользователи и группы

6.1. Идентификация пользователей

QNX располагает средствами учёта (идентификации) пользователей системы [11]. Идентификация пользователя заключается в присвоении ему системного имени и пароля. После установки QNX на компьютер система уже содержит единственного пользователя с именем root для входа в систему без пароля (пустая строка) и с правами администратора, которому предоставлены неограниченные полномочия по управлению и использованию ресурсов системы, например файлов. Вход в систему любого пользователя с именем root приводит к автоматической установке ему системного каталога с именем /root в качестве текущего.

Именование, установка пароля и спецификация полномочий вновь добавляемых пользователей системы производится системным администратором с помощью команды passwd. Любой пользователь может в дальнейшем изменить свой пароль, выполнив команду passwd. Утилита запросит прежний пароль и дважды попросит ввести новый пароль. В отличие от системного администратора с именем root пользователи с именами отличными от root имеют ограниченные права доступа к ресурсам системы, установленные им администратором при регистрации.

Для учёта пользователей система использует системные файлы /etc/passwd и /etc/shadow. Файл /etc/passwd состоит из строк следующего формата:

username:haspw:userid:group:comment:homedir:shell

username - имя пользователя, используемое для входа в систему;

haspw - если поле не пустое, то в файле /etc/shadow хранится пароль пользователя;

userid - идентификатор пользователя (у root - 0);

group - числовой идентификатор первичной группы (см. п.2.2.);

comment - любая строка, не содержащая символа ":";

homedir - домашний каталог пользователя, т.е. каталог, в котором пользователь может произвольно создавать и удалять файлы;

shell - командный интерпретатор, который запускает утилита login при успешном входе пользователя в систему.

Файл /etc/shadow состоит из строк следующего формата:

username:passwd:lastch:minch:maxch:warn:inact:expire:reserved

username - имя пользователя, используемое для входа в систему;

passwd - зашифрованный пароль пользователя;

lastch - время последней модификации;

minch - минимальное количество дней для модификации;

maxch - максимальное количество дней для модификации;

warn - количество дней для предупреждения;

inact - максимальное количество дней между входами в систему;

expire - дата истечения доступа;

reserved - зарезервировано для дальнейшего использования;

6.2. Создание и идентификация групп

Кроме идентификации и учёта отдельных пользователей QNX располагает средствами учёта групп пользователей (групп). Добавление и идентификация группы производится

системным администратором. Добавление новых групп и их идентификация выполняется путём простого редактирования файла /etc/group, строки которого имеют формат:

groupname:reserved:group:member

groupname - имя группы;

reserved - зарезервировано для дальнейшего использования;

group - числовой идентификатор группы;

member - список имён пользователей, принадлежащих данной группе.

В список имён пользователей можно добавить имя любого пользователя (username), идентифицированного в файле /etc/passwd. Пользователь системы может быть членом нескольких групп, одна из которых назначается первичной (primary), остальные - дополнительными (supplementary). Первичной группой становится та группа, числовой идентификатор которой прописывается в поле group строки учёта пользователя в файле /etc/passwd.

Как и пользователям, группам соответствуют определённые права доступа к ресурсам системы. Если пользователь является членом группы, то дополняет свои права доступа к ресурсам системы правами группы. Принцип формирования групп и включения в них пользователей определяется системным администратором.

6.3. Удаление пользователей и групп

Учётная информация о пользователях и группах хранится в файлах /etc/passwd, /etc/shadow, /etc/group. Для того чтобы удалить из системы пользователя, системный администратор должен отредактировать эти три файла, удалив или изменив строки, соответствующие удаляемому пользователю.

Для удаления группы достаточно отредактировать файл /etc/group, но необходимо обязательно убедиться, что нет пользователей, для которых эта группа является первичной (эта группа не должна упоминаться в файле /etc/passwd).

6.4. Сеанс работы пользователя в системе

Для доступа пользователя к системе QNX выполняет процедуру аутентификации пользователя. Она заключается в том, что после включения компьютера и загрузки системы запускается утилита login или phlogin, которая запрашивает у пользователя имя и пароль. Если пользователь зарегистрирован в системе и ввёл правильные имя и пароль, то login запускает командный интерпретатор, указанный в файле /etc/passwd, и пользователь входит в систему.

Вход пользователя в операционную систему учитывается системой так, что она создаёт так называемый сеанс работы пользователя. Сеанс логически объединяет все процессы, которые порождаются в результате входа и последующей работы пользователя в системе. Если пользователь входит в систему в режиме консоли командной строки, то на терминале появляется приглашение к вводу команд. Для системного администратора приглашение имеет вид "#", для обычного - "$". После этого пользователь может вводить командные строки.

По окончании работы пользователь завершает работу в системе. В режиме консоли командной строки работа завершается путём ввода команды "exit". При завершении работы пользователя в системе все незавершённые процессы, принадлежащие сеансу пользователя, автоматически аннулируются.

6.5. Разграничение доступа к файлам

QNX регулирует возможность различных процессов выполнять с файлом операции чтения/записи или запуск на выполнение. Для этого введено понятие владельца файла. В дескрипторе каждого файла при его создании, помимо прочих, формируются атрибуты UID и GID, специфицирующие соответственно владельца-пользователя (владелец процесса, создавшего файл) и владельца-группу (группа процессов с правами владения файлом), наследуемых от процесса, создавшего файл. При открытии файла любым процессом эти атрибуты файла сравниваются с соответствующими атрибутами в дескрипторе открывающего файл процесса для проверки его прав доступа к файлу. Процесс может быть либо владельцем файла, либо входить в группу, являющуюся владельцем файла, либо быть отнесённым по отношению к файлу к "прочим" процессам. Для владельцев и для прочих у файла установлены соответствующие разрешения на выполнение операций с файлом (чтение, запись или исполнение). Существуют системные функции, позволяющие управлять владельцами файла (значением атрибутов UID и GID).

Различают следующих пользователей файла. Зарегистрированный в системе пользователь, который некоторым способом инициировал создание файла, назначается *пользователем-владельцем* этого файла. Кроме пользователя-владельца при создании файла с ним ещё ассоциируется некоторая зарегистрированная в системе группа пользователей, которая специфицируется как *группа-владелец*. Остальные зарегистрированные в системе пользователи по отношению к созданному файлу рассматриваются как *прочие*. При создании файла его владельцам (пользователю, группе), а также прочим устанавливаются права доступа к файлу. Любой зарегистрированный в системе пользователь может получить права доступа к файлу, владельцем которого он не является, если он становится членом группы-владельца этого файла. Включение пользователя в группу автоматически предоставляет ему по отношению файлу, которым владеет группа, установленные для группы права доступа. Наоборот - для лишения пользователя прав доступа к файлу, ассоциированных с группой, достаточно исключить его из состава этой группы. Владельцы файла, а также их права могут при выполнении приложения меняться.

Каждому созданному в QNX файлу устанавливаются параметры - *идентификатор владельца* (UID) и *идентификатор группы* (GID), а также атрибуты *прав доступа* для трёх категорий пользователей: *владельца* (user owner), *группы* (group owner) и "*остальных*" (other). Отметим, что владелец может не являться членом группы, владеющей файлом.

Каждому запущенному в системе процессу в дескрипторе устанавливаются два параметра - так называемые эффективный идентификатор владельца (EUID) и эффективный идентификатор группы (EGID). Идентификаторы файла и процесса используются для проверки прав доступа процесса к файлу. Когда процесс пытается открыть какой-либо файл, QNX сначала проверяет, совпадает ли EUID процесса с UID файла. Если совпадает, то проверяется, имеет ли ассоциированный с процессом владелец право открыть файл с указанным режимом доступа. Если владелец имеет такие права, то файл открывается, если не имеет, то проверяется, совпадает ли EGID процесса с GID файла и право данной группы открывать файл с указанным режимом доступа. Если никакие идентификаторы не совпали, тогда данному процессу может быть разрешён режим доступа, установленный для "остальных". Для процессов с EUID=0 (т.е. для пользователя root) доступ к файлам предоставляется без процедуры проверки прав.

6.6. Права доступа к файлу

Операционная система QNX различает три базовых класса доступа к файлу:

User access (u) - для владельца-пользователя файла.

Group access (g) - для членов группы, являющейся владельцем файла.

Other access (o) - для остальных пользователей (кроме суперпользователя (администратор системы), у которого максимальные права).

Для каждого из указанных классов доступа QNX поддерживает три типа прав доступа к файлу: на чтение (r), на запись (w) и на выполнение (x).

Права доступа могут быть изменены только владельцем файла или пользователем root посредством команды chmod.

Значение (семантика) прав доступа зависит от типа файла. Для обычного файла смысл операций вытекает из названий прав доступа. Например, если исполняемый файл является скриптом командного интерпретатора shell, то для его запуска понадобится право на чтение (r), поскольку при выполнении скрипта командный интерпретатор должен иметь возможность считывать команды из файла, а также право на выполнение (x). Права доступа для каталога не столь очевидны. Система трактует операции чтения и записи для каталогов отлично от остальных файлов. Право чтения каталога позволяет получить только имена файлов, находящихся в данном каталоге. Чтобы получить дополнительную информацию о файлах каталога, требуются права на "выполнение" каталога (x). Это же право нужно иметь для доступа ко всем каталогам на пути к указанному. Особое значение для каталога имеет право на запись. Создание и удаление файлов в каталоге требуют права на запись в этот каталог. Но при этом, чтобы удалить некоторый файл из каталога, не обязательно иметь какие-либо права доступа к этому файлу, важно лишь иметь право на запись для каталога, в котором находится этот файл.

7. Программный интерфейс QNX

Одной из целей, которые изначально ставились перед разработчиками QNX, являлось создание удобной среды программирования и программного интерфейса - API. Разработка программ невозможна без знания интерфейса системных вызовов и без понимания внутренних структур и функций, предоставляемых операционной системой. Осмысленное администрирование системы также затруднительно без представления о том, как работает QNX. Программный интерфейс QNX позволяет наглядно показать внутренние механизмы операционной системы.

Программный интерфейс выражается в виде системных вызовов и функций стандартных библиотек. Далее будут рассмотрены важнейшие, функции стандартной библиотеки ввода/вывода, системные вызовы работы с файлами и управления файловой системой, системные вызовы создания процесса и управления процессами, входящие в состав API ОСРВ QNX или её аналога защищённой операционной системы реального времени (ЗОСРВ) «Нейтрино» [12].

7.1. Системные вызовы и функции стандартных библиотек

Прикладные задачи имеют возможность воспользоваться базовыми услугами, предоставляемыми QNX. Эти услуги получили название *системных вызовов*. Системный вызов инициирует функцию, выполняемую средствами операционной системы от имени процесса, выполнившего вызов, и является программным интерфейсом самого низкого уровня взаимодействия прикладных процессов с операционной системой. В среде программирования QNX они определяются как функции языка С. В QNX каждый системный вызов имеет соответствующую функцию (или семейство функций) с тем же именем, хранящуюся в стандартной библиотеке языка С (в дальнейшем эти функции будем для простоты называть системными вызовами). Фактически эти функции играют роль оболочки, которая выполняет необходимые преобразования аргументов и инициируют требуемый системный вызов ОС.

Помимо системных вызовов программный интерфейс предлагает большой набор функций общего назначения. Эти функции не являются системными вызовами, хотя в процессе выполнения многие из них выполняют системные вызовы. Эти функции также хранятся в стандартных библиотеках С и наряду с системными вызовами составляют основу среды программирования в QNX. К этим функциям относятся функции библиотеки ввода/вывода, функции распределения памяти, функции управления процессами и т.д.

7.2. Обработка ошибок

Разница между системными вызовами и библиотечными функциями проявляется ещё в способе передачи процессу информации об ошибке, произошедшей во время выполнения системного вызова или функции библиотеки.

Обычно в случае возникновения ошибки системные вызовы возвращают целое значение -1 и устанавливают значение глобальной системной переменной errno, указывающее причину возникновения ошибки. Системный заголовочный файл <errno.h> содержит коды ошибок, значения которых может принимать переменная errno, с краткими комментариями.

Библиотечные функции, как правило, не устанавливают значение переменной errno, а код возврата различен для разных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к описанию этих функций в справочной системе QNX.

Рассмотрим более подробно обработку ошибок, возникающих при выполнении системных вызовов с использованием переменной errno. Заметим, что значение errno не обнуляется следующим нормально завершившимся системным вызовом. Следовательно, значение errno имеет смысл только после вызова, который завершился с ошибкой.

Стандарт ANSI C определяет две функции, помогающие сообщить причину ошибочной ситуации:

```
#include <string.h>
char *strerror(int errnum);
```

и

```
#include <errno.h>
#include <stdio.h>
void perror(const char *s).
```

Функция strerror() принимает в качестве аргумента errnum номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации. Функция perror() выводит в стандартный поток сообщений об ошибках (обычно на экран) содержимое строки s и вслед за ним – системную информацию об ошибочной ситуации, основываясь на значении переменной errno.

8. Функции управления файловой системой

8.1. Смена корневого каталога

Процесс может изменить свой корневой каталог с помощью системного вызова:

#include<unistd.h>

int chroot(const char *path);

Функция chroot() делает каталог path корневым каталогом. После этого поиск файлов с абсолютными именами, начинающимися с '/', будет производиться, начиная с каталога, указанного аргументом path. Заметим, однако, что пользовательский текущий каталог сохраняется.

Для изменения корневого каталога значение эффективного пользовательского ID процесса (EUID) должно соответствовать системному администратору. Системная жёсткая ссылка "..", входящая в корневой каталог, указывает на него самого. В связи с этим жёсткая ссылка ".." не может быть использована для доступа к файлам за пределами поддерева, входящего в корневой каталог.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

8.2. Смена текущего каталога

Процесс может изменить текущий каталог с помощью системного вызова:

#include<unistd.h>

int chdir(const char *path);

Функция chdir() изменяет текущий рабочий каталог на path, который может быть относительным или абсолютным именем. Так как с процессом связывается только один текущий каталог, то в многонитевых приложениях любая нить, вызвавшая chdir(), изменит текущий каталог для всех нитей в этом процессе.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

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

int main( int argc, char* argv[] )
{
   if( argc != 2 ) {
      fprintf( stderr, "Use: cd <directory>\n" );
      return EXIT_FAILURE;
   }

   if( chdir( argv[1] ) == 0 ) {
      printf( "Directory changed to %s\n", argv[1] );
      return EXIT_SUCCESS;
```

```
  } else {
    perror( argv[1] );
    return EXIT_FAILURE;
  }
}
```

8.3. Создание каталога

Новый каталог можно создать с помощью вызова:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

Функция mkdir() создаёт новый пустой каталог, специфицированный в path с разрешениями доступа, заданными в mode в виде комбинации флагов разрешения, определённых в заголовочном файле <sys/stat.h>. ID владельца каталога устанавливается равным эффективному ID пользователя процесса. ID группы каталога устанавливается равным ID группы родительского каталога (если установлен флаг использования ID группы родительского каталога) или эффективный ID группы процесса.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример, создаётся новый каталог с именем /src in /hd:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
  mkdir( "/hd/src",
       S_IRWXU |
       S_IRGRP | S_IXGRP |
       S_IROTH | S_IXOTH );

  return EXIT_SUCCESS;
}
```

8.4. Удаление каталога

Для удаления каталога используется вызов:

```
#include <sys/types.h>
#include <unistd.h>
int rmdir(const char* path);
```

Функция rmdir() удаляет каталог, специфицированный в path, если его счётчик связей равен 0 и он не открыт ни каким процессом. Каталог должен быть пустым.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/*Удаляет каталог с именем /home/terry*/
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void ){
   rmdir( "/home/terry" );

   return EXIT_SUCCESS;
}
```

8.5. Создание жёсткой связи

Создать связь к существующему файлу можно с помощью вызова:

```
#include <unistd.h>
int link(const char* pname, const char* new);
```

Функция link() создаёт новый элемент каталога с именем new (путь доступа для новой связи), являющийся жёсткой ссылкой на существующий файл с именем pname (путь доступа к существующему файлу), и увеличивает счётчик связей для указанного файла на 1. При этом файл не может быть каталогом или находится на другом устройстве.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

8.6. Создание символической связи

Создать символическую связь с файлом можно с помощью вызова:

```
#include <unistd.h>
int symlink(const char* pname, const char* slink);
```

Функция symlink() создаёт символическую связь с именем slink, которая содержит абсолютное имя файла pname (slink является именем создаваемой символической связи, pname есть абсолютное имя, содержащееся в символической связи).

Контроль прав доступа к файлу pname не выполняется и отсутствует необходимость существования файла. Символическую связь можно создать к файлу и каталогу даже на другом устройстве.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/* Создание символической связи для "/usr/nto/include" в текущем каталоге */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main( void )
 {
  if( symlink( "/usr/nto/include", "slink" ) == -1) {
    perror( "slink -> /usr/nto/include" );
    exit( EXIT_FAILURE );
  }
  exit( EXIT_SUCCESS );
 }
```

8.7. Чтение символической связи

Содержимое символической связи можно прочитать и поместить в буфер с помощью вызова:

```
#include <unistd.h>
int readlink(const char* path, char* buf, size_t bufsiz);
```

Функция readlink() помещает содержание символической связи с именем path в буфер buf. Если readlink() завершается успешно, то bufsiz байтов содержания символической связи помещается в buf. Возвращаемый набор символов размером bufsiz не является строкой (не имеет в конце нуля).

При успешном завершении функция возвращает количество байтов, помещённых в buf. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/* В качестве аргумента программа принимает имя символической связи */
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char buf[PATH_MAX + 1];

int main( int argc, char** argv ){
 int nread, fd;
/* Чтение содержимого символической связи */

if(( nread = readlink( argv[1], buf, PATH_MAX )) == -1) {
    perror( argv[1] );
    exit(EXIT_FAILURE);
}
buf[nread] = '\0';
printf( "Символическая связь %s -> %s\n", argv[1], buf );
exit( EXIT_SUCCESS );
}
```

8.8. Переименование файла

Для переименования файла или связи используют вызов:

```
#include <stdio.h>
int rename(const char* old, const char* new);
```

Функция rename() меняет имя файла, специфицированного в old, на имя, специфицированное в new. Если файл (или пустой каталог) с именем new существует, он заменяется.

При успешном завершении функция возвращает 0. В случае ошибки возвращается значение отличное от 0 и устанавливается errno.

Пример:

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

int main( void ){
  if( rename( "old.dat", "new.dat" ) ) {
    puts( "Ошибка переименования old.dat в new.dat." );

    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

8.9. Удаление файла

Удалить файл (или связь) можно с помощью вызова:

```
#include <unistd.h>
int unlink( const char * path );
```

Функция unlink() удаляет файл (связь), чьё имя указано в path. Если указана жёсткая связь, то она удаляется, а счётчик связей соответствующего файла уменьшается на 1. Если указан файл или символическая связь, то реальное удаление объекта произойдёт в тот момент, когда счётчик связей окажется равным 0. До этого момента реальное удаление откладывается. Эта функция эквивалентна функции remove().

Если каталог, содержащий файл, перезаписываем и у каталога установлен бит S_ISVTX, то процесс может удалять или переименовывать файлы внутри такого каталога только, если выполняется одно или большее количество следующих условий:

- EUID процесса равен UID файла;
- EUID процесса равен UID каталога, содержащего файл;
- процесс имеет право на перезапись файла;
- пользователь является системным администратором (UID = 0).

При успешном завершении функция возвращает 0. В случае ошибки возвращается значение отличное от 0 и устанавливается errno.

Пример:

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

int main( void ){
   if( unlink( "vm.tmp" ) ) {
      puts( "Error removing vm.tmp!" );

      return EXIT_FAILURE;
   }

   return EXIT_SUCCESS;
}
```

8.10. Управление владельцами и правами доступа к файлам

Владение файлом определяет не только возможность доступа к файлу, но и тот набор операций (прав доступа), который пользователь может совершить с файлом: чтение, запись, запуск на выполнение (для исполняемых файлов). Изменение прав доступа может осуществлять только владелец-пользователь, а также пользователь root.

8.10.1. Управление владельцами

Изменение для файла пользователя-владельца производится командой chown, а изменение для файла группы-владельца выполняется командой - chgrp. Эти команды может выполнить только пользователь root.

Одного собственника можно заменить другим с помощью функций:

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char * path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group );
```

Функция chown() или fchown() заменяет у файла, специфицированного именем, указанным в path, или дескриптором fd, текущие значения владельца-пользователя – UID, и владельца-группы – GID, на значения, содержащиеся в owner и group соответственно. Если файл является символической связью, chown() изменяет владельцев непосредственно у файла или каталога, на который осуществляется ссылка.

Для изменения соответствующих атрибутов непосредственно символической ссылки следует использовать функцию:

```
int lchown(const char * path, uid_t owner, gid_t group);
```

Заметим, что только процесс с эффективным ID пользователя (EUID), равным пользовательскому ID файла (UID), или с привилегиями системного администратора (пользователь root с UID=0) может изменить непосредственно атрибуты файла.

Замечание. В QNX при формировании прав доступа создаваемого файла может быть установлен флаг _POSIX_CHOWN_RESTRICTED (его наличие проверяется путём тестирования флага _POSIX_CHOWN_RESTRICTED с помощью функции pathconf()). Наличие этого флага означает, что только системный администратор может изменить владельцев файла. Обычным владельцам это окажется не доступным.

Если аргумент path ссылается на обычный файл, то при успешном выполнении функции атрибуты файла S_ISUID и S_ISGID очищаются.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/*
 * Замена собственников файлов, заданных в списке, на
 * текущие значения собственников процесса - getuid(), getgid()
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <unistd.h>

int main( int argc, char** argv ) {
   int i;
   int ecode = 0;

   for( i = 1; i < argc; i++ ) {
    if( chown( argv[i], getuid(), getgid() ) == -1 ) {
      perror( argv[i] );
      ecode++;
    }
   }
   exit( ecode );
 }
```

8.10.2. Управление правами доступа

Изменить права доступа к файлу можно с помощью функций:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char * path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Функция chmod() или fchmod() изменяет для файла, специфицированного именем, указанным в path, или дескриптором файла в fd, флаги S_ISUID, S_ISGID, S_ISVTX и флаги разрешений доступа на соответствующие флаги, заданные в аргументе mode.

Эффективный ID пользователя у процесса должен соответствовать владельцу файла, или процесс, должен иметь необходимые привилегии, чтобы делать это.

Если каталог перезаписываем, и для каталога установлен бит S_ISVTX, то процесс может удалять или переименовывать файлы внутри такого каталога только, если выполняется одно или большее количество следующих условий:

- EUID процесса равен UID файла;
- EUID процесса равен UID каталога, содержащего файл;
- процесс имеет право на перезапись файла;
- пользователь является системным администратором (UID = 0).

Если процесс не имеет соответствующих привилегий и ID группы файла не соответствует эффективному ID группы процесса, а файл является обычным файлом, то бит S_ISGID в атрибутах файла очищается при успешном выполнении chmod().

Если эффективный ID пользователя процесса равен ID владельца файла, или процесс имеет соответствующие привилегии (его владельцем является системный администратор), то chmod() устанавливает для файла биты S_ISUID, S_ISGID.

Вызов chmod() не имеет никакого эффекта для уже открытых файлов. В этом случае требуется вызов fchmod().

При успешном завершении функции возвращают 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример 1:

```
/*
Изменяет права доступа к файлам,
разрешая чтение/запись только личному владельцу
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv ){
  int i;
  int ecode = 0;

    for( i = 1; i < argc; i++ ) {
       if( chmod( argv[i], S_IRUSR | S_IWUSR ) == -1 ) {
          perror( argv[i] );
          ecode++;
       }
    }
    return ecode;
}
```

Пример 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

void main( int argc, char **argv ){
  int fd;

/* Создание файла с правами r w x */
    fd = creat("my_file", S_IRUSR | S_IWUSR | S_IXUSR);

/* Добавим флаг SUID */
    fchmod(fd, S_IRWXU | S_ISUID );

/* Добавим флаг SGID */
    fchmod(fd, S_IRWXU | S_ISUID | S_ISGID );
```

```
/* Добавим флаг блокирования записей файла SGRP */
   fchmod(fd, S_IRWXU | S_ISUID | S_ISGID | S_ISGRP);
}
```

9. Функции базового ввода/вывода для работы с файлами

В среде программирования QNX существуют два основных интерфейса для работы с файлами:

Интерфейс системных вызовов, предлагающий системные функции низкого уровня, непосредственно взаимодействующие со средствами операционной системы.

Стандартная библиотека ввода/вывода, предлагающая функции буферизированного ввода/вывода.

Второй интерфейс является надстройкой над интерфейсом системных вызовов и предлагает более удобный (упрощённый) способ работы с файлами. Функции этого интерфейса определены стандартом ANSI языка С как стандартная библиотека ввода/вывода. Поэтому использование этих функций обеспечивает программе наибольшую мобильность. В то же время они не обеспечивают всех возможностей по управлению вводом/выводом, предоставляемых операционной системой QNX, которые могут потребоваться при создании приложений реального времени. Выбор между функциями интерфейса системных вызовов и стандартной библиотеки зависит от необходимой степени контроля ввода/вывода и требованием мобильности программы.

Функции стандартной библиотеки ввода/вывода, обеспечивая программе максимальную мобильность, в то же время не реализуют всех возможностей по управлению вводом/выводом в файлы, разделяемые параллельными процессами, предоставляемых операционной системой QNX, которые могут потребоваться приложению реального времени при работе с файлами. В этом случае необходимо воспользоваться системными функциями QNX для управления файлами.

9.1. Открытие файла

Для открытия или создания файла используется функция:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag [,mode_t mode]);
```

Функции open()позволяют процессу открыть файл (устройство), имя которого указано в path. Это имя может быть как абсолютным (полным, начинающимся с корневого каталога /), так и относительным (указанным относительно текущего каталога). Для открытого файла создаётся новый дескриптор, который не разделяется с каким-либо другим процессом в системе. Режим доступа к открытому файлу устанавливается согласно значению флагов, сформированных в разрядах аргумента oflag.

Значение oflag является целым значением и строится с использованием операции поразрядного "ИЛИ" над значениями флагов, символические константы для которых определены в заголовочном файле <fcntl.h>. Флаги позволяют определить режим открытия и доступа к существующему или вновь создаваемому файлу, а также уточнить реализацию выбранного режима доступа. Значения и семантика флагов следующая:

O_RDONLY - открыть существующий файл только для чтения;

O_RDWR - открыть для чтения и записи;

O_WRONLY - открыть только для записи.

Выбранный режим доступа к открываемому файлу уточняется флагами:

O_APPEND - если этот флаг установлен, то для режимов, допускающих запись в файл (O_RDWR и O_WRONLY), перед каждой операцией записи указатель файла будет устанавливаться в конец этого файла.

O_DSYNC - если этот флаг установлен, то каждый запрос write() будет ждать, пока все данные не будут успешно записаны. То есть все записи в файл, а также соответствующие им изменения в метаданных файла будут сохранены на внешнем носителе.

Ядро кэширует данные, считываемые или записываемые на внешний носитель, для ускорения этих операций. Обычно запись данных в файл ограничивается только записью в буферный кэш ядра операционной системы, данные из которого впоследствии записываются на внешний носитель. По умолчанию возврат из функции write() происходит после записи данных в буферный кэш, не дожидаясь записи на внешний носитель. Установка флага O_DSYNC гарантирует, что в результате завершения write() даже при фатальных нарушениях в системе (но исправности внешнего носителя) данные будут сохранены в файле и могут быть прочитаны при последующем открытии файла. Если носитель оборудован защитой от записи, то это интерпретируется, как поломка носителя, данные не будут записаны, даже если write() указывает, что все успешно.

O_RSYNC - если этот флаг установлен, то каждый запрос read() завершается только тогда, когда данные успешно прочитаны.

O_SYNC - если этот флаг установлен, то каждый запрос read() или write() завершается только тогда, когда данные успешно прочитаны или записаны. Все записи в файл, а также соответствующие им изменения в метаданных файла будут сохранены на внешнем носителе.

O_CLOEXEC - дескриптор файла не будет наследоваться вновь создаваемыми процессами (будет закрыт при запуске процесса).

O_CREAT - установка этого флага указывает, что если открываемый файл с указанным именем не существует, то необходимо создать новый файл и открыть его для записи. Если файл с указанным именем уже существует, то будет ли открыт этот файл, или создан новый файл для записи, или функция open() завершится с ошибкой определяется значением флага O_EXCL.

O_EXCL - флаг используется совместно с O_CREAT. При его установке новый файл будет создан только в том случае, если файл с указанным именем не существует и это же действие с тем же именем файла в данный момент не выполняется другими процессам. В противном случае возвращается ошибка создания файла. Если флаг не установлен, то при наличии файла с указанным именем флаг O_CREAT игнорируется и открывается существующий файл. Флаг O_EXCL без O_CREAT также игнорируется.

O_LARGEFILE - разрешает, чтобы указатель файла был длиной в 64 бита (при работе с огромными файлами).

O_NOCTTY - если этот флаг установлен и указанный файл является терминальным устройством, то функция open() не делает терминальное устройство управляющим терминалом для процесса.

O_NONBLOCK - если этот флаг установлен, то функция open() завершается без ожидания, когда устройство будет готовым или доступным. Последующее поведение устройства определяется его спецификой. Если флаг очищен, то функция open() ждёт, прежде чем завершиться, когда устройство будет готовым или доступным. Готовность устройства определяется его спецификой. В некоторых случаях реакция на флаг не определена.

O_REALIDS - требует использовать реальные UID и GID процесса для проверки разрешённых прав доступа к файлу.

O_TRUNC - если файл существует, является обычным файлом и он успешно открыт в режиме O_WRONLY или O_RDWR, то длина файла усекается до нуля, а права доступа и владелец оставлены неизменными. Флаг не имеет никакого эффекта для канала FIFO, файлов устройств или каталогов. Если файл открывается как O_RDONLY, флаг O_TRUNC игнорируется.

Аргумент mode имеет значение только когда создаётся новый файл и позволяет задать для файла права доступа пользователей.

Значение mode является целым значением и строится с использованием операции поразрядного "ИЛИ" над значениями флагов, специфицирующих права доступа к вновь создаваемому файлу.

Устанавливаемые права доступа и дополнительные атрибуты файла (SUID, SGID и Sticky bit) определяются следующими символическими значениями флагов (определены в заголовочном файле <sys/stat.h>):

Флаг	Значение
S_ISUID	Установить для файла бит атрибута SUID
S_ISGID	Установить для файла бит атрибута SGID или установить обязательное блокирование файла (определяется в зависимости от значений других флагов)
S_ISVTX	Установить Sticky bit
S_IRWXU	Установить право на чтение, запись и выполнение для владельца
S_IRUSR	Установить право на чтение для владельца
S_IWUSR	Установить право на запись для владельца
S_IXUSR	Установить право на выполнение для владельца
S_IRWXG	Установить право на чтение, запись и выполнение для группы
S_IRGRP	Установить право на чтение для группы
S_IWGRP	Установить право на запись для группы
S_IXGRP	Установить право на выполнение для группы
S_IRWXO	Установить право на чтение, запись и выполнение для остальных
S_IROTH	Установить право на чтение для остальных
S_IWOTH	Установить право на запись для остальных
S_IXOTH	Установить право на выполнение для остальных

Атрибуты SUID и SGID имеют смысл только при создании исполняемых файлов. Назначение атрибутов заключается в следующем. При запуске дочернего процесса (на основе исполняемого файла) его идентификаторы UID, GID, EUID, EGID устанавливаются равными соответствующим идентификаторам родительского процесса. Если же для исполняемого файла установлен атрибут SUID, то идентификаторы дочернего процесса будут установлены равными идентификатору владельца исполняемого файла. Атрибут SGID исполняемого файла делает то же с групповым идентификатором процесса. То есть, если существует, например, файл утилиты myprog, владельцем которого является пользователь с именем "user1", с правами доступа на выполнение файла для всех пользователей, то процесс, созданный при запуске утилиты myprog пользователем "user2", будет иметь UID и EUID унаследованные не от "user2", как следовало

бы ожидать, а равные значениям UID и EUID владельца файла утилиты myprog - "user1". Не трудно догадаться, что установка для исполняемого файла атрибутов SUID и SGID не безобидна с точки зрения безопасности информации. Но иногда без них нельзя обойтись, например – для файла утилиты passwd, позволяющей пользователю изменить свой пароль. Изменение пароля требует изменения содержимого системных файлов /etc/passwd и /etc/shadow. Понятно, что предоставление права на запись в эти файлы всем пользователям системы является отнюдь не лучшим решением. С другой стороны, необходимо, чтобы процесс, запущенный на основе исполняемого файла утилиты passwd, был согласован по правам доступа с файлами /etc/passwd и /etc/shadow. Установка атрибута SUID исполняемому файлу утилиты /usr/bin/passwd позволяет изящно разрешить это противоречие. Поскольку владельцем файла утилиты /usr/bin/passwd является пользователь root (системный администратор), то кто бы ни запустил утилиту passwd на выполнение, соответствующий процесс приобретает права системного администратора, т.е. может производить запись в системные файлы, защищенные от остальных пользователей. Понятно, что требование по безопасности для программ, подобных passwd, должны быть повышены за счёт ограничения их функциональных возможностей. Они не должны выполнять никаких операций, способствующих наследованию прав доступа другими процессами (например, вызов других программ).

Значение флага S_ISGID зависит от того, установлено или нет право на выполнение для группы - S_IXGRP. В первом случае он будет означать установку SGID, а во втором - обязательное блокирование файла.

Блокирование файлов позволяет устранить возможность конфликта, когда более одного процесса одновременно работают с одним и тем же файлом. Файловая система разрешает нескольким процессам одновременный доступ к файлу для чтения и записи. Хотя операции записи и чтения, осуществляемые с помощью системных вызовов read() и write(), являются атомарными, по умолчанию отсутствует синхронизация между отдельными вызовами. Другими словами, между двумя последовательными вызовами read() одного процесса другой процесс может модифицировать данные файла. Это, в частности, может привести к несогласованным операциям с файлом, и как следствие, к нарушению целостности его данных.

Если создаётся новый файл, то идентификатор владельца файла устанавливается равным эффективному идентификатору владельца процесса, выполняющего функцию open() (UID=EUID), соответственно идентификатор группы устанавливается равным эффективному идентификатору группы процесса (GID=EGID) или идентификатору группы родительского каталога (в котором создаётся файл), если для родительского каталога установлен флаг SGID.

В результате выполнения функция open() возвращает неотрицательное целое число (дескриптор файла), представляющее наименьшее значение неиспользуемого дескриптора файла. Для файла с прямым доступом указатель файла установлен в начало. В случае ошибки, возвращается -1 и устанавливается errno.

Замечание. При запуске программы для неё автоматически создаются три дескриптора: 0 - стандартный ввод, 1 - стандартный вывод, 2 - стандартный вывод сообщений об ошибках.

Частным случаем функции open() является функция:

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode]);
```

Эта функция создает и открывает новый файл с именем, указанным в pathname, и правами доступа, заданными в mode. Она эквивалентна вызову:
open("myfile.dat",O_WRONLY|O_CREAT|O_TRUNC,mode);

При успешном выполнении функция creat() возвращает дескриптор файла, в случае ошибки возвращается -1 и устанавливает errno.

Пример:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main( void ) {
int fd;

/* создать и/или открыть файл для записи*/
/* существующий файл использовать снова как пустой*/
/* создать с правами чтения/записи для владельца */

fd=open("myfile.dat",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
…
/* открыть существующий файл для чтения */

fd = open("myfile.dat", O_RDONLY);
…
/* открыть существующий файл для дозаписи или создать новый файл для записи, если такой файл не существует, с правами доступа на чтение/запись для всех*/

fd = open("myfile.dat",
O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
return EXIT_SUCCESS;
}
```

9.2. Доступ к файлу

Функции, реализующие системные вызовы ввода/вывода QNX, следующие:

```
#include<unistd.h>
ssize_t write(int fd, void *buf, size_t nbyte);
ssize_t read(int fd, void *buf, size_t nbyte);
off_t lseek(int fd, off_t offset, int whence);
off_t tell(int fd);
int close(int fd);
```

В представленных функциях аргумент fd является дескриптором открытого файла. Функция write() записывает в файл nbyte байт из буфера buf и возвращает количество

записанных байт. Функция read() читает из файла nbyte байт и записывает их в буфер buf, возвращает количество считанных. Функция lseek() смещает указатель позиции файла на величину offset относительно указанной базы whence, которая может принимать значения: SEEK_SET – от начала файла, SEEK_END – от конца файла, SEEK_CUR – от текущей позиции. Возвращает новое значение позиции. Функция tell() возвращает текущее значение позиции файла (указателя файла). Функция close() закрывает файл и возвращает нулевое значение. Все функции в случае ошибки возвращают -1 и устанавливают errno.

10. Структура и выполнение приложений реального времени

10.1. Программы, процессы, нити

В ОС QNX разработка приложений базируется на использовании таких конструктивных элементов как исполняемые *программные модули*, *процессы*, *нити* (Рис. 1).

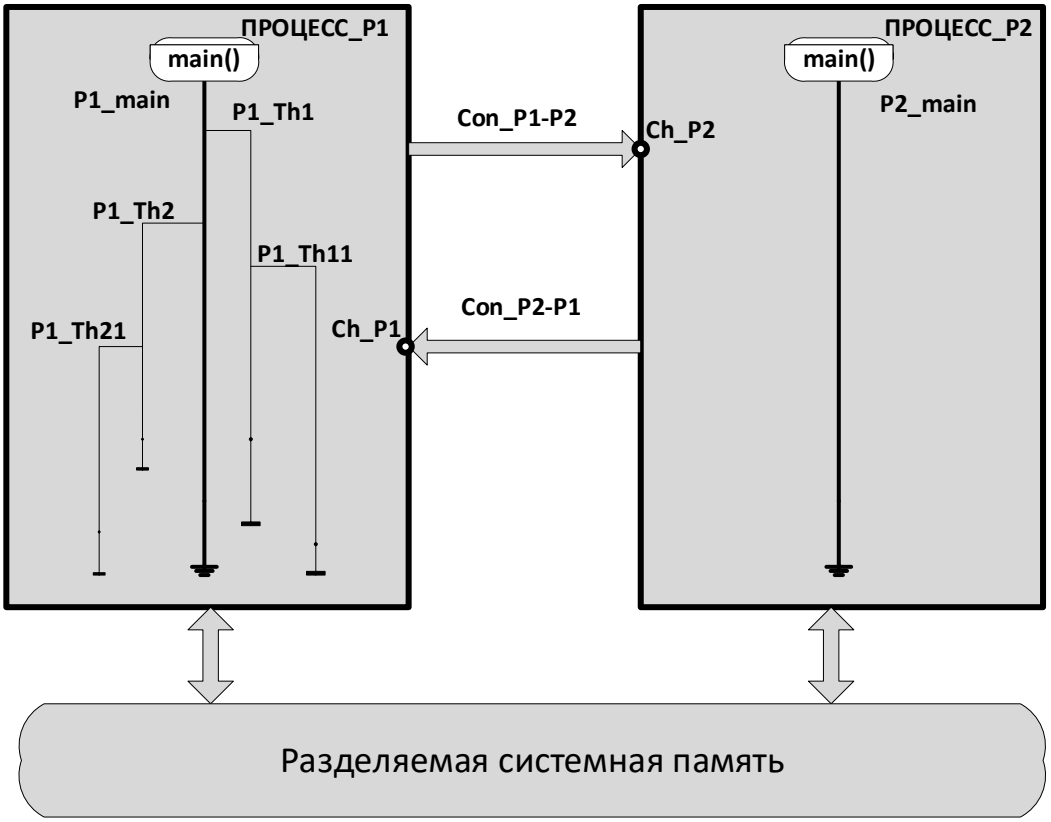


Рис. 1 Процессно-нитевая структура приложения

В общем случае приложение реального времени в операционной системе QNX является многопроцессным. Процессы запускаются на основе предварительно созданных исполняемых программных модулей (файлов). Каждая загрузка на выполнение программного модуля рассматривается и реализуется системой как запуск некоторого нового процесса. Одновременно могут выполняться любое количество процессов. Процессы выполняются либо независимо, либо взаимодействуя друг с другом посредством обмена сообщениями или разделяемой системной памяти. Очевидно, что для выполнения процесса ОС должна предоставлять каждому загруженному программному модулю долю системных ресурсов, таких как память, процессор, доступ к устройствам ввода/вывода и различным другим системным ресурсам, включая услуги ядра ОС. Выделяемые модулю системные ресурсы рассматриваются операционной системой как среда выполнения процесса. Создание нового процесса осуществляется каждый раз при очередной загрузке программного модуля на выполнение. При этом не важно разные запускаются программные модули или один и тот же.

Каждый процесс обеспечивается системой собственным изолированным и уникальным адресным пространством. У программного модуля текущего процесса нет возможности непосредственно обратиться в адресное пространство другого процесса. Программный модуль процесса может считывать и записывать информацию в собственный раздел данных и в стек,

но ему недоступны данные и стеки других процессов. В то же время процесс обеспечивает своему программному модулю возможность взаимодействовать с программными модулями, исполняющимися в других процессах, используя системные средства межпроцессного взаимодействия, например, посредством передачи или приёма сообщений или специально созданной процессами области общей разделяемой системной оперативной памяти. На Рис. 1 показаны два связанных процесса: ПРОЦЕСС_P1, ПРОЦЕСС_P2. Процессы имеют каналы для приёма сообщений (Ch_P1, Ch_P2) и соединения с каналами (ConP1-P2, ConP2-P1) для посылки сообщений, а также имеют доступ к разделяемой системной памяти.

Исполнение процесса начинается с особого запуска операционной системой функции main() соответствующего программного модуля. Такой способ запуска функции main() интерпретируется как создание в процессе потока управления, для обозначения которого далее будет использоваться термин "*нить*" (thread). Внутри процесса, при необходимости, нить main() может запускать в качестве нитей и другие функции программного модуля. Запуск функции в качестве нити осуществляется посредством специального запроса ОС и этим отличается от обычного вызова функции. Нить main() на Рис. 1 будет являться родителем запущенных ею нитей (дочерних): P1_Th1, P1_Th2. Нити очередного поколения могут в свою очередь порождать нити следующего поколения (P1_Th11, P1_Th21) и т.д.

Все нити, запущенные во всех процессах, выполняются параллельно. При этом каждая нить уже не имеет внутреннего программного параллелизма и рассматривается как программная процедура последовательно выполняемых инструкций. В итоге, исполнение программы в процессе в общем случае выглядит как запуск нити main() и последующее порождение и параллельное выполнение совокупности нитей, которые могут асинхронно создаваться и выполняться, разделяя общие ресурсы, а также асинхронно завершаться. Завершение нити main() является исключительным, так как приводит к автоматическому завершению выполнения всех созданных нитей и удалению процесса. Таким образом, процесс является, по сути, контейнером созданных и параллельно выполняющихся нитей, в котором, в момент старта, системой автоматически создаётся и запускается на выполнение нить - main().

10.2. Процессно-нитевая структура ПРВ

Исполняемая процессно-нитевая структура ПРВ формируется при загрузке проекта ПРВ в вычислительную систему и в общем случае модифицируется при выполнении. Механизм процессов обеспечивает при разработке логической структуры ПРВ *структурный* параллелизм (модульность) ПРВ, а механизм нитей - *функциональный* параллелизм ПРВ. В частности, ПРВ может представляться как один процесс, в котором исполняются множество нитей. Однако, представление ПРВ в виде совокупности взаимодействующих процессов придаёт ему следующие дополнительные свойства:

- возможность создания распределённого ПРВ – модульность на макроуровне;
- гибкость и конфигурируемость ПРВ;
- повышение надёжности ПРВ.

Модульность ПРВ на макроуровне обеспечивает относительную независимость исполняемых процессами программных модулей друг от друга, что обуславливает возможность их асинхронной разработки. Гибкость и конфигурируемость ПРВ вытекает из возможности динамически модифицировать его процессную структуру непосредственно при функционировании ПРВ ("на лету"). Единственная возможность установить зависимость

процессов друг от друга - наладить между ними информационную связь с помощью небольшого количества средств взаимодействия. Так как поведение процесса в рамках решаемой им задачи может быть чётко специфицировано, то упрощается процедура выявления и устранения ошибок в исполняемой программе. Кроме того, процессы выполняются в независимых адресных пространствах. Две нити, работающие в разных процессах, изолированы одна от другой и, следовательно, не могут изменять локальные данные в чужом процессе. Это способствует большей надёжности ПРВ.

10.3. Базовая архитектура QNX

Базовая архитектура QNX предельно компактна и основана на двух фундаментальных понятиях: микроядро и межпроцессное взаимодействие на основе сообщений. Микроядро (его и называют Neutrino) обеспечивает минимально необходимый набор системных функций:

- создание и уничтожение нитей;
- диспетчеризация нитей;
- поддержка механизмов синхронизации нитей;
- поддержка механизмов передачи сообщений;
- поддержка механизма обработки прерываний;
- поддержка часов и таймеров.

Кроме этого, Neutrino ничего больше не делает. Как видно из приведённого списка функций микроядро не управляет процессами. Формирование процессов, обеспечивающих работу входящих в них нитей в изолированных (локальных) адресных пространствах процессов, осуществляет специальная системная компонента QNX, называемая "*администратор (менеджер) процессов*". Администратор процессов выполняет следующие функции:

- управление процессами;
- управление механизмами защиты памяти;
- поддержка механизма разделяемой памяти и механизмов межпроцессного взаимодействия;
- управление пространством имён путей.

Поскольку управление процессами и памятью являются необходимыми функциями операционной системы, то администратор процессов скомпонован с микроядром Neutrino в единый программный модуль procnto, который ещё называют системным процессом. Заметим, что переключение контекста между нитями в одном процессе происходит без участия администратора процессов, а между нитями в разных процессах - с участием администратора процессов.

Вся остальная функциональность QNX обеспечивается специальными процессами, называемыми *администраторами (менеджерами) ресурсов* и системными прикладными процессами. С помощью администраторов ресурсов, например, реализуется доступ к устройствам внешней памяти (администраторы различных файловых систем) или сети (администратор сети qnet, администратор TCP/IP). В качестве системного прикладного процесса выступает, например, процесс, инициируемый запуском программного модуля командного интерпретатора shell.

10.4. Управление процессами

Процесс выступает в качестве обособленной вычислительной среды, обеспечивающей в общем случае параллельное и асинхронное выполнение запущенных в нём вычислительных процедур (функций) или так называемых – *нитей* [10][12].

10.4.1. Жизненный цикл процесса

Жизненный цикл процесса включает четыре этапа.

1. *Создание*. Процесс может быть создан только другим (родительским) процессом. При этом администратор процессов формирует для него необходимую среду выполнения нитей и создаёт у себя системную структуру атрибутов – *дескриптор*, характеризующих свойства процесса, используемых ядром QNX для управления созданным процессом.
2. *Загрузка.* Код и данные заданного программного модуля загружаются в выделенную процессу локальную память и запускается нить main().
3. *Выполнение.* В общем случае, в процессе, параллельно и асинхронно выполняются нить main() и создаваемые по мере необходимости другие нити.
4. *Завершение*. Завершение процесса проходит две стадии. На первой стадии происходит освобождение ресурсов, связанных с процессом (страницы ОЗУ, открытые файловые дескрипторы и т.п.). На второй стадии код возврата завершаемого процесса передаётся его процессу-родителю. При этом, в зависимости от режима создания дочернего процесса возможны следующие варианты поведения процесса-родителя:
   - процесс-родитель блокируется в ожидании получения кода завершения дочернего процесса. Код завершения доставляется процессу-родителю, он выходит из блокированного состояния, а дочерний процесс терминируется;
   - процесс-родитель не блокируется в ожидания и отказывается от получения кода завершения дочернего процесса. В этом случае дочерний процесс становится независимым от родительского процесса;
   - процесс-родитель не блокируется в ожидании, но не отказывается от получения кода завершения дочернего процесса. Дочерний процесс при завершении становится DEAD-блокированным процессом или "зомби". Для такого процесса администратор процессов сохраняет минимум управляющей информации, необходимой только для того, чтобы доставить код завершения родительскому процессу, когда он выполнит переход в ожидание завершения дочернего процесса.

10.4.2. Атрибуты и свойства процесса

Дескриптор созданного процесса имеет ряд атрибутов, определяющих свойства процесса, которые операционная система использует при управлении процессом. К первоочередным свойствам относятся:

- идентификатор процесса (Process ID - PID);
- идентификатор родительского процесса (Parent Process ID - PPID);
- реальные идентификаторы владельца и группы (UID и GID);
- эффективные идентификаторы владельца и группы (EUID и EGID);
- идентификаторы дополнительных групп;
- текущий каталог;

– корневой каталог;
– управляющий терминал (TTY);
– номер приоритета;
– дисциплина диспетчеризации;
– маска создания файлов (UMASK).

10.4.3. Идентификаторы процесса

Каждый процесс имеет уникальный целочисленный идентификатор PID, позволяющий ядру системы различать процессы. Когда создаётся новый процесс, ядро присваивает ему очередной свободный идентификатор. Присвоение идентификаторов происходит по возрастающей, т.е. идентификатор нового процесса больше, чем идентификатор процесса, созданного перед ним. Если идентификатор достиг максимального значения, следующий процесс получит минимальный свободный PID, и цикл повторяется. Когда процесс завершает свою работу, ядро освобождает его идентификатор.

Нити могут определить PID своего процесса с помощью функции:

```
#include <process.h>
pid_t getpid(void);
```

Кроме PID, каждый процесс в качестве атрибута содержит идентификатор породившего его родительского процесса - PPID. Он используется, в частности, в качестве адреса для доставки кода завершения дочернего процесса. Если дочерний процесс создан на том же узле локальной сети, что и родительский процесс, то нити дочернего процесса могут определить ID родительского процесса с помощью функции:

```
#include <sys/types.h>
#include <process.h>
pid_t getppid(void);
```

Однако, если дочерний процесс создан на другом узле локальной сети, то нити дочернего процесса не смогут определить ID родительского процесса с помощью этой функции. Для этого потребуется предусмотреть некоторый способ явной передачи значения ID родительского процесса дочернему.

Важную роль для процесса играют *реальный* и *эффективный* идентификаторы владельца, *реальный* и *эффективны*й идентификаторы группы, которые наследуются от файла программного модуля, использованного для создания процесса. Если в файле исполняемого модуля, который родительский процесс использовал для запуска дочернего процесса, не установлен флаг SUID, то эффективный идентификатор владельца в дескрипторе дочернего процесса устанавливается равным идентификатору владельца в дескрипторе родительского процесса (EUID=UID) и дочерний процесс получает (наследует) соответствующие права доступа к ресурсам системы родительского процесса. Если флаг SUID в файле исполняемого модуля был установлен, то эффективный идентификатор владельца дочернего процесса устанавливается равным идентификатору владельца из дескриптора файла исполняемого модуля, использованного родительским процессом для запуска дочернего процесса, и в итоге получает права доступа, владельца файла. Следовательно, полученное значение эффективного идентификатора владельца - EUID, в дескрипторе процесса позволяет ему наследовать права доступа к системным ресурсам (в первую очередь к ресурсам файловой системы) либо владельца родительского процесса (флаг SUID в дескрипторе файла программного модуля не

установлен), либо владельца, файла программного модуля (флаг SUID в дескрипторе файла программного модуля установлен).

Реальным идентификатором группы дочернего процесса, является идентификатор первичной или текущей группы, к которой принадлежал родительский процесс - GID. Эффективный идентификатор группы служит для определения прав доступа процесса к системным ресурсам, когда у него отсутствуют соответствующие права доступа как у владельца. Если в файле исполняемого модуля не установлен флаг SGID, то эффективный идентификатор группы дочернего процесса делается равным идентификатору группы (EGID=GID) родительского процесса. Если флаг SGID установлен, то эффективный идентификатор группы устанавливается равным идентификатору группы в дескрипторе файла исполняемого модуля, использованного для запуска процесса, и получает права доступа, соответствующие группе, владеющей файлом.

При регистрации пользователя в системе утилита login запускает системный командный интерпретатор shell. При этом идентификаторам UID (EUID) и GID (EGID) процесса shell присваиваются значения, полученные из записи, соответствующей пользователю в файле паролей /etc/passwd, полученной при регистрации. В результате командный интерпретатор приобретает права, определённые для данного пользователя и его первичной группы. Когда далее командный интерпретатор, выполняя команду, как родитель порождает соответствующий дочерний процесс на базе заданного программного модуля, то, по рассмотренным выше правилам, формируются все четыре идентификатора владельцев и, следовательно, запускаемый процесс получает соответствующие права, что и shell. Поскольку в текущем сеансе работы в системе конкретного пользователя прародителем всех процессов является shell, то в дескрипторах порождаемых процессов идентификаторы их реальных владельцев будут совпадать, а идентификаторы эффективных владельцев могут отличаться.

Для получения значений идентификаторов владельцев процесса используются следующие системные вызовы:

```
#include <sys/typs.h>
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

Процесс может изменить значения идентификаторов владельцев процесса с помощью системных вызовов:

```
#include <sys/typs.h>
#include <unistd.h>
int setuid(uid_t uid);
int seteuid(uid_t euid);
int setgid(gid_t gid);
int getegid(gid_t egid);
```

Системные вызовы setuid() и setgid() устанавливают сразу реальный и эффективный идентификаторы владельцев процесса, а системные вызовы seteuid() и setegid() - только эффективные. Чтобы можно было изменить идентификатор группы необходимо, чтобы имя пользователя было в списке членов этой группы в файле /etc/group.

Команды ps и sin командного интерпретатора shell позволяют вывести список процессов, выполняющихся в системе, и их атрибуты.

10.4.4. Текущий и корневой каталоги

Текущий и корневой каталоги процесс наследует от родительского процесса. Они могут быть изменены с помощью функций chdir() и chroot() соответственно. Определить текущий каталог процесса можно с помощью функции:

```
#include <unistd.h>
char* getcwd(char* buffer, size_t size);
```

Функция формирует строку, заканчивающуюся признаком конца строки \0, с именем текущего каталога, которая размещается в буфере памяти, указанном в buffer, размером, по крайней мере, size байт. Максимальный размер строки с именем каталога определяется значением PATH_MAX + 1 байт.

Функция возвращает адрес строки с именем текущего каталога, или NULL, в случае ошибки, код которой помещается в errno.

10.4.5. Приоритет и дисциплина диспетчеризации процесса

Приоритет и дисциплина диспетчеризации процесса используются ядром при определении очерёдности запуска нитей процесса при распределении процессорных ресурсов. Операционная система QNX Neutrino поддерживает до 256 уровней приоритета планирования. Непривилегированная нить (nonroot) может иметь приоритет в диапазоне от 1 до 63. Привилегированные нити (у которых EUID=0) могут иметь приоритет выше 63. Специальная системная нить idle ("пустая" нить) в администраторе процессов имеет нулевой приоритет и всегда готова к исполнению. Таким образом, диапазон приоритетов нитей, следующий:

- 0 - системный процесс idle;
- от 1 до 63 непривилегированная нить;
- от 1 до 255 привилегированная нить.

Системные сервисы (администраторы, менеджеры) запускаются с приоритетом 1, а драйверы устройств – с приоритетом 2. Прикладные приложения запускаются с приоритетом 3. Нити с одинаковым приоритетом используют процессорные ресурсы с учётом назначенных им дисциплин диспетчеризации.

Приоритет и дисциплина диспетчеризации процесса наследуются нитью main() от родительского процесса.

10.4.6. Управляющий терминал

Управляющий терминал (терминальная линия) — это терминал или псевдотерминал, ассоциированный с процессом. Этот терминал предназначен для связи пользователя со своими процессами. Все процессы группы имеют один и тот же управляющий терминал. С управляющим терминалом процесса связан специальный файл псевдоустройства с именем /dev/tty. Драйвер этого псевдоустройства, по существу, перенаправляет запросы на фактический терминальный драйвер, который может быть различным для различных процессов.

10.5. Типы процессов

10.5.1. Системные процессы

Системные процессы являются процессами, порождаемыми ядром или специальными системными программами ОС. Системные процессы, порождаемые ядром, не имеют исполняемых модулей и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, они могут вызывать функции и обращаться к данным, недоступным для остальных процессов.

10.5.2. Процессы демоны

Демоны — это не интерактивные процессы, которые запускаются обычным образом - путём загрузки в память соответствующих им программных модулей (исполняемых файлов), и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы и обеспечивают работу различных подсистем ОС: терминального доступа, печати, сетевого доступа и т.п. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают пока тот или иной процесс запросит у ядра определённую услугу, которую ядро перенаправит соответствующему демону, например, запрос на запуск процесса или открытие файла и т.п.

10.5.3. Прикладные процессы

К прикладным процессам относятся все остальные процессы, выполняющиеся в системе. К ним, как правило, относят процессы, порождённые в рамках пользовательского сеанса работы. Важнейшим из таких процессов является командный интерпретатор, который назначается пользователю при регистрации и запускается при входе пользователя в систему (login shell) и обеспечивает его работу в QNX. Завершение работы login shell приводит к отключению от системы. Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае при завершении сеанса работы пользователя и выходе из системы все его процессы терминируются.

10.6. Группы и сеансы

После создания процесса ему присваивается уникальный идентификатор. Дополнительно процессу назначается *идентификатор группы процессов* (process group ID). *Группа процессов* включает один или более процессов и существует, пока в системе присутствует хоты бы один процесс этой группы. Временно́й интервал, начинающийся с создания группы и заканчивающийся, когда последний процесс её покинет, называется *временем жизни группы*. Последний процесс может либо завершить своё выполнение, либо перейти в другую группу.

Нахождение в группе предоставляет процессам дополнительные свойства. Ряд системных вызовов могут быть применимы одновременно ко всем процессам группы. Например, системный вызов waitpid() позволяет родительскому процессу ожидать завершения конкретного процесса или любого процесса группы.

Каждый процесс при создании становится ещё и членом так называемого *сеанса* (session), объединяющего одну или нескольких групп процессов. Понятие сеанса введено для логического объединения групп процессов, порождённых в результате регистрации и последующей работы пользователя в системе. Сеанс создаётся, когда пользователь заходит в

систему, для него. Когда пользователь завершает работу в системе – сеанс аннулируется, завершая все текущие процессы, запущенные пользователем в рамках сеанса.

Процесс имеет возможность получить значение идентификатора собственной группы процессов или группы процесса, который является членом того же сеанса. Это делается с помощью системных вызовов getpgrp() и getpgid():

```
#include <unistd.h>
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

Аргумент pid адресует процесс, идентификатор группы которого требуется узнать. Если этот процесс не принадлежит тому же сеансу, что и процесс, сделавший системный вызов, функция возвращает ошибку.

Процесс, используя системный вызов setgid() может стать членом существующей группы или создать новую группу.

```
#include <unistd.h>
int setpgid(pid_t pid,pid_t pgid);
```

Функция устанавливает идентификатор группы для процесса pid равным pgid. Процесс имеет возможность установить идентификатор группы для себя и своих дочерних процессов. Однако процесс не может изменить идентификатор группы для дочернего процесса, который преобразовался в другой процесс, выполнив системный вызов exec(). Если значения обоих аргументов равны, то создаётся новая группа с идентификатором pgid, а процесс становится *лидером* этой группы (group leader). Группа не удаляется при завершении её лидера, пока в неё входит хотя бы один процесс.

Сеанс также имеет свой идентификатор. Идентификатор сеанса можно узнать с помощью функции getsid().

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Идентификатор pid должен адресовать процесс, являющийся членом того же сеанса, что и процесс, вызвавший getsid(). Заметим, однако, что эти ограничения не распространяются на процессы, имеющие привилегии администратора системы.

Процесс может создать и новый сеанс с помощью функции setsid().

```
#include <unistd.h>
pid_t setsid(void);
```

Новый сеанс создаётся лишь при условии, что процесс не является лидером какого-либо сеанса. В случае успеха процесс становится *лидером сеанса* и лидером новой группы.

Понятия группы и сеанса тесно связаны с терминалом или, точнее, с драйвером терминала. Каждый сеанс может иметь один ассоциированный терминал, который называется *управляющим терминалом* (controlling terminal), а группы, созданные в данном сеансе, наследуют этот управляющий терминал. Наличие управляющего терминала позволяет ядру контролировать стандартный ввод/вывод процессов, а также возможность направить сигнал всем процессам группы, ассоциированной с терминалом, например, при его отключении. При входе в систему терминал пользователя становится управляющим для лидера сеанса - интерпретатора shell, и всех процессов, порождённых лидером, которые запускает пользователь из командной строки интерпретатора. При выходе пользователя из системы shell завершает свою работу и отключается от управляющего терминала, что вызывает отправление

сигнала SIGHUP всем незавершённым процессам текущей группы. Это гарантирует, что после завершения пользователем работы в системе в ней не останется запущенных им процессов (кроме созданных им демонов и процессов, запущенных в фоновом режиме).

10.7. Запуск процессов

Для запуска процесса в файловой системе должен присутствовать файл с необходимым исполняемым программным модулем. Процесс можно запустить "вручную", с помощью командного интерпретатора shell, или из программы, используя специальные функции. При создании нового процесса происходит обращение к администратору процессов. В общем случае администратор процессов создаёт среду выполнения для нового процесса. После создания для процесса среды выполнения ядро выполняет запуск нити main() в этом процессе.

10.7.1. Запуск процесса из shell

Ручной запуск процесса осуществляется посредством команды командного интерпретатора shell, введённой с управляющего терминала. В качестве команды в командной строке интерпретатора shell достаточно указать имя файла исполняемого программного модуля. При этом процесс может запускаться с последующим ожиданием интерпретатора shell его завершения или без ожидания (в фоновом режиме). Например, если запускается процесс на основе программного модуля my_prog, находящегося в текущем каталоге, то команда

```
$my_prog
```

запустит процесс и shell будет ждать его завершения, после чего выдаст на экран приглашение для ввода следующей команды. Для запуска процесса в фоновом режиме команда должна включать опцию &:

```
$my_prog &
$
```

и приглашение к вводу следующей команды появляется немедленно. Если при запуске требуется понизить приоритет процесса, то используется команда

```
$nice my_prog
```

Порядок запуска процессов ПРВ можно полностью возложить на shell. Для этого создаются сценарии запуска (скрипты или командные файлы). Это обычные текстовые файлы, в которых в качестве строк задаются команды shell. Командный интерпретатор рассматривает введение в качестве команды имени любого текстового файла как попытку выполнить скрипт.

10.7.2. Программный запуск процессов

Любая нить текущего процесса может выполнить запуск нового процесса. В QNX имеются различные способы программного запуска процессов. Для этих способов существуют соответствующие им функции запуска процесса. Они следующие:

- функция system();
- семейство функций exec*();
- семейство функций spawn*();
- функция fork();
- функция vfork().

10.7.2.1. Функция system()

Можно запустить процесс посредством не ручного, а программного вызова командного интерпретатора shell для выполнения любой команды и, в частности, команды запуска процесса в виде имени исполняемого модуля, на базе которого процесс будет порождаться:

```
#include <stdlib.h>
int system(const char *command);
```

Поведение функции зависит от значения аргумента command. Если command равен NULL, то определяется, имеется ли в системе shell. Функция возвращает ноль, если shell отсутствует, или отличное от нуля значение, если присутствует. Если command не равен NULL, то запускается копия shell и ему, через указатель command, передаётся командная строка для обработки и возвращается результат выполнения shell. Если shell не смог загрузиться, то возвращается -1. При успешной загрузке shell возвращается статус завершения его выполнения, соответствующий выполненной команде.

Пример:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int main( void ){
   int status;

   status = system("ls");
   if( status == -1 ) {
     printf( "shell не может запуститься \n" );
   } else {
     printf( "Код завершения команды %d\n",
        WEXITSTATUS(status); /*системный макрос выделения из статуса возврата кода
                                    завершения*/
   }
   return EXIT_SUCCESS;
 }
```

10.7.2.2. Функции семейства exec*()

Существует набор функций семейства exec*(), которые реализуют однотипный способ запуска процессов и отличаются только некоторыми деталями. Особенность запуска процесса этими функциями заключается в следующем. Процесс, вызвавший функцию семейства exec*(), прекращает выполнять текущий программный код и начинает выполнение инструкций нового (указанного в вызове функции) программного модуля. Важно то, что идентификатор процесса - pid, при этом остаётся прежним. К функциям семейства exec*() относятся функции execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(). В качестве примера рассмотрим функции execl(), execle() и execv() семейства exec*().

Объявление функции execl() имеет вид:

```
#include <process.h>
```

int execl(const char *path,const char *arg0, const char *arg1,...,const char *argn, NULL);

Возвращает -1 в случае ошибки, иначе выход из функции неипроисходит.

Пример:

```
#include <stddef.h>
#include <process.h>
…
execl( "myprog", "myprog", "ARG1", "ARG2", NULL );
…
```

Объявление функции execle() имеет вид:

```
#include <process.h>
int execle(const char* path, const char* arg0, const char* arg1..., const char* argn, NULL,
        const char* envp[]);
```

Пример:

```
#include <stddef.h>
#include <process.h>

char* env_list[] = {"SOURCE=MYDATA", "TARGET=OUTPUT", "lines=65", NULL};
execle("myprog","myprog","ARG1", "ARG2", NULL, env_list);
…
```

В приведённом примере окружение для вызываемой программы состоит из трёх переменных окружения: SOURCE, TARGET и lines. Набор переменных окружения должен завершаться нулевым указателем – NULL.

Объявление функции execv() имеет вид:

```
#include <process.h>
int execv(const char *path, char *const argv[]);
```

Функция возвращает -1 в случае ошибки, иначе функция не возвращает управления, а лишь заменяет контекст текущего процесса новым модулем.

Пример:

```
#include <stddef.h>
#include <process.h>

char* arg_list[] = {"myprog","ARG1","ARG2",NULL};

execv("myprog",arg_list);
…
```

В рассмотренных примерах предполагается, что программный модуль myprog находится в текущем каталоге.

10.7.2.3. Функции семейства spawn*()

В отличие от функций семейства exec*() функции семейства spawn*() способны создавать новый (дочерний) процесс со своим pid, параллельно выполняющийся вместе с родительским

процессом. Если родительский процесс каким-то образом завершается, то это не означает, что ядро по этой причине должно сразу завершить и дочерний процесс. Имеется возможность с помощью флагов управлять поведением процесса-родителя после запуска дочернего процесса:

P_WAIT	Родительский процесс блокируется до тех пор, пока дочерний процесс не завершится.
P_NOWAIT	Родительский процесс не блокируется, выполняется параллельно с дочерним процессом, но в последствии должен ожидать завершение дочернего процесса, вызвав функцию wait(), waitpid().
P_NOWAITO	Родительский процесс не блокируется, выполняется параллельно с дочерним процессом и не должен ожидать завершения дочернего процесса (гарантирует от перехода дочернего процесса в DEAD-блокированное состояние - "зомби", при его завершении).
P_OVERLAY	Родительский процесс заменяется дочерним процессом, сохраняя PID родительского процесса (как при использовании функций семейства exec*())

В качестве примера рассмотрим функции spawnl() и spawnve() семейства spawn*(). Объявление функции spawnl() имеет вид:

```
#include <process.h>
int spawnl(int mode, const char * path, const char * arg0, const char * arg1..., const char * argn,
        NULL);
```

Аргумент mode предназначен для задания флага управления поведением родительского процесса. Предназначение остальных аргументов то же, что и у функции execl(). Возвращаемое функцией spawnl() значение зависит от значения флага в аргументе mode:

Значение mode	Возвращаемое значение
P_WAIT	Статус завершения дочернего процесса.
P_NOWAIT	PID дочернего процесса. Чтобы получить статус завершения дочернего процесса, родительский процесс должен выполнить функцию wait() или waitpid(), которой в качестве параметра передаётся PID дочернего процесса, завершение которого ожидается.
P_NOWAITO	PID дочернего процесса или 0, если дочерний процесс стартовал на удалённом узле. Статус завершения такого дочернего процесса получить невозможно.

В случае ошибки возвращается -1 (устанавливается errno).

Пример:

```
#include <stddef.h>
#include <process.h>

int exit_val;
...
exit_val = spawnl(P_WAIT,"myprog","myprog","ARG1","ARG2",NULL);
...
```

Объявление функции spawnve() имеет вид:

```
#include <process.h>
```

int spawnve(int mode, const char *path, char *const argv[], char *const envp[]);

Аргумент argv есть указатель на вектор аргументов. Значение argv[0] должно указывать на имя файла исполняемого модуля. Последний член argv должен быть NULL указатель. Значение argv и argv[0] не могут быть NULL указателем, даже если не требуется передавать какие-либо аргументы запускаемому процессу.

Аргумент envp есть указатель на массив указателей на строки, определяющие переменные среды. Массив должен завершаться NULL указателем. Определение переменной среды задается в форме:

"имя_переменной=значение_переменной"

Если значение envp равно NULL, то дочерний процесс наследует среду родительского процесса. Дочерний процесс при этом может осуществить доступ к переменным среды, используя глобальную системную переменную environ (определена в <unistd.h>).

Возвращаемое функцией spawnve() значение формируется по тем же правилам, что и у функции spawnl().

Пример:

```
#include <stddef.h>
#include <process.h>

char* arg_list[] = {"myprog", "ARG1", "ARG2", NULL};
char* env_list[] = {"SOURCE=MYDATA", "TARGET=OUTPUT", "lines=65", NULL};
int exit_val;
...
exit_val = spawnve( P_WAIT,"myprog",arg_list,env_list);
...
```

10.7.2.4. Функция fork()

Функция fork() создаёт новый (дочерний) процесс, являющийся точной копией процесса, его породившего (родительского). При этом дочерний процесс имеет уникальный PID. Идентичные дочерние процессы могут иметь разных родителей их породивших.

Отметим, что дочерний процесс имеет собственную копию дескрипторов файла родительского процесса, ссылающихся на те же самые открытые файлы родителя, а также собственные копии потоков к каталогам, открытых родительским процессом. Значения tms_utime, tms_stime, tms_cutime, и tms_cstime дочернего процесса установлены в 0. Блокировки файла, предварительно установленные родителем, дочерним процессом не наследуются. Задержанные звонки таймера очищаются для дочернего процесса. Набор задерживаемых сигналов для дочернего процесса инициализируется как пустой.

После выполнения функции fork() родительским процессом оба процесса продолжат выполняться с оператора, следующего за вызовом функции fork().

Объявление функции имеет вид:

```
#include <sys/types.h>
#include <process.h>
pid_t fork(void);
```

В дочернем процессе функция возвращает 0, а в родительском - PID дочернего процесса. В случае ошибки fork() возвращает -1 родительскому процессу и устанавливает errno.

Пример:

```
#include <stdio.h>
#include <sys/types.h>
#include <process.h>
int main(int argc,char *argv[]){
  int retval;
      printf("Это родительский процесс\n");
      fflush(stdout);
      retval = fork(void);
      printf("Кто это сказал?\n");
return EXIT_SUCCESS;
}
```

После вызова fork() оба процесса выполнят второй вызов printf(). Данное приложение выведет на экран следующее:

```
Это родительский процесс
Кто это сказал?
Кто это сказал?
```

Чтобы различить эти два процесса необходимо проанализировать возвращаемое функцией fork() значение в retval. В дочернем процессе retval будет иметь нулевое значение, а в родительском - содержать PID дочернего процесса. Для пояснения рассмотрим фрагмент программы:

```
printf("PID родителя равен %d\n",getpid());
fflush(stdout);
if(retval = fork(void)){
    printf("Это родитель, PID дочернего процесса %d\n", retval);
}
else{
    printf("Это дочерний процесс, PID %d\n",getpid());
}
```

Относительно функции fork() следует отметить, что в текущих версиях QNX она может быть успешно реализована только в процессах с одной нитью.

10.7.2.5. Функция vfork()

Объявление функции имеет вид:

```
#include <sys/types.h>
#include <process.h>
pid_t vfork(void);
```

Функция vfork() создаёт новый процесс, как и функция fork(), но в разделяемом адресном пространстве, и блокирует родительский процесс до тех пор, пока дочерний процесс не завершится или не вызовет функцию семейства exec*().

10.7.3. Ожидание завершения дочернего процесса

Если родительский процесс запускает дочерний процесс в режиме последующего ожидания его завершения, то он либо сразу (не явно) переходит в состояние ожидания статуса завершения (режим P_WAIT), либо (режим P_NOWAIT) продолжает выполнение, но будет в последствии явно ожидать статус завершения, выполнив функцию ожидания wait() или waitpid().

10.7.3.1. Функция wait()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait( int *stat_loc );
```

Аргумент stat_loc - NULL или указатель на место, где функция сохраняет статус завершения дочернего процесса.

Функция wait() приостанавливает поток родительского процесса, выполнивший функцию, до получения статуса завершения одного из дочерних процессов (если прежде родительский процесс не получил сигнал, который принудительно завершает его ожидание). Если информация о статусе доступна до вызова wait() - дочерний процесс находится в состоянии зомби, возврат из функции происходит немедленно. Если указатель stat_loc не NULL, то анализ статуса завершения целесообразно осуществлять, используя набор системных макросов (определённые в <sys/wait.h>), которые извлекают информацию из полученного статуса. Аргументом в макросах является целочисленное значение, на которое указывает stat_loc.

Макросы стандарта POSIX:

WEXITSTATUS(stat_val) - возвращает младшие 8 бит статуса завершения дочернего процесса, если значение макроса WIFEXITED(stat_val) не равно нулю.

WIFEXITED(stat_val) возвращает ненулевое значение, если статус получен от дочернего процесса, который нормально завершил работу.

WIFCONTINUED(stat_val) - возвращает ненулевое значение, если статус получен от дочернего процесса, который продолжился после остановки.

WIFSIGNALED(stat_val) - возвращает ненулевое значение, если статус получен от дочернего процесса, который завершил работу из-за приёма сигнала, который не был обработан.

WIFSTOPPED(stat_val) - возвращает ненулевое значение, если статус получен от дочернего процесса, который был остановлен.

WSTOPSIG(stat_val) - возвращает номер сигнала, который вызвал остановку дочернего процесса, если значение WIFSTOPPED(stat_val) не равно нулю.

WTERMSIG(stat_val) - возвращает номер сигнала, который вызвал завершение дочернего процесса, если значение WIFSTOPPED(stat_val) не равно нулю.

Функция возвращает идентификатор дочернего процесса. Если возникла ошибка функция возвращает -1, код ошибки записывается в errno.

10.7.3.2. Функция waitid()

Функция waitid() не является частью стандарта POSIX, но даёт больше информации о статусе, чем описанные выше системные макросы (см. описание в справочной системе ЗОСРВ «Нейтрино») [12].

10.8. Организация взаимодействия между процессами

Процессы в QNX обеспечивают базовый интерфейс взаимодействия между нитями [10][12]. Ключевым механизмом этого взаимодействия является *механизм обмена сообщениями*. При передаче сообщений между процессами один процесс (нити которого

принимают сообщения) считается *сервером*, а другой (нити которого посылают сообщения) - *клиентом*. Поэтому механизм обмена сообщениями в QNX называют моделью "*клиент/сервер*". Один и тот же процесс может одновременно обеспечивать, как приём, так и посылку сообщений, т.е. выполнять функции и клиента, и сервера. В частности, процесс может обеспечивать взаимодействие собственных нитей между собой посредством сообщений.

Для приёма сообщений некоторая нить сервера должна создать объект, называемый *каналом*. В сервере может быть создан один и более каналов. В свою очередь, чтобы нити клиента получили возможность посылать сообщения в канал сервера, некоторой нитью должен быть создан объект, называемый *соединением* (*связью*) с каналом (установить соединение с каналом). С одним каналом сервера может быть установлено произвольное количество соединений одним или несколькими клиентами. Нити процессов могут создавать и уничтожать каналы и соединения по мере необходимости.

10.8.1. Создание и удаление каналов

10.8.1.1. Создание канала

Создание канала может быть осуществлено любой нитью процесса (например, main()). Для этого используется функция:

```
#include <sys/neutrino.h>
int ChannelCreate(unsigned flags);
```

Аргумент flags представляет собой набор флагов, установка которых определяет свойства и поведение канала по отношению к процессу при возникновении особых ситуаций, контролируемых ядром QNX. Значения флагов будет рассматриваться далее по мере необходимости. Пока будем использовать значения флагов по умолчанию, для чего следует положить flags равным 0.

В случае успеха функция возвращает ID созданного канала (chid). Если возникает ошибка, то возвращается -1 и в errno помещается код ошибки.

10.8.1.2. Удаление канала

Удаление канала выполняется нитью с помощью функции

```
#include <sys/neutrino.h>
int ChannelDesstroy(int chid);
```

В качестве аргумента chid выступает ID ранее созданного канала. В случае ошибки функция возвращает -1, а в errno помещается код ошибки. Если выполнение успешное, то возвращается произвольное значение отличное от -1.

10.8.2. Установление и удаление соединений с каналом

10.8.2.1. Установление соединения

Установление соединения с каналом выполняется нитью с помощью функции:

```
#include <sys/neutrino.h>
int ConnectAttach(uint32_t nd, pid_t pid, int chid, unsigned index, int flags);
```

Аргументы функции:

nd - ID узла в сети (nd=ND_LOCAL_NODE, если узел местный);

pid - ID процесса-сервера;

chid - ID канала сервера;

index - минимально допустимое значение возвращаемого функцией ID соединения. Для формирования системного значения ID соединения без ограничения с низу, индекс устанавливается равным 0;

flags - набор флагов, установка которых определяет системные свойства соединения, в частности, поведение соединения по отношению к нитям процесса-клиента, пославших сообщение по данному соединению, при возникновении особых ситуаций, контролируемых ядром QNX (например, если в flags установлен флаг _NTO_COF_CLOEXEC (символическая системная константа), то соединение будет удаляться, когда клиент вызывает функцию семейства exec*(), чтобы заменить код процесса). Для выбора свойств соединения по умолчанию, flags устанавливается равным 0.

Функция ConnectAttach() создаёт соединение клиента с каналом chid, принадлежащим серверу pid на узле nd. Если узел местный, то nd присваивается значение системной константы ND_LOCAL_NODE (равна нулю). Если в качестве клиента и сервера выступает один и тот же процесс, то значение pid устанавливается равным 0.

Важно отметить, что значение аргумента index ограничивает с низу возвращаемое функцией значение системного ID соединения. Система будет возвращать первое не занятое значение ID соединения, начинающееся со значения, установленного аргументом index. Однако заметим, что при взаимодействии процесса с системным администратором ввода/вывода и также создаются соединения, ID которых трактуются уже как дескрипторы файлов, например: ID=0 - stdin, ID=1 – stdout, ID=2 - stderr. При index=0 возможно, что в качестве значения ID создаваемого соединения будет назначено значение ID, которое в системе ассоциируется с дескриптором ранее открытого файла и при выполнении процессом, например, функции printf() символьная строка будет послана не в открытый поток к файлу, а в канал, с которым процесс установил такое соединение. Кроме того, дочерний процесс может наследовать дескрипторы файлов родителя. При этом соединение, созданное родителем без использования _NTO_SIDE_CHANNEL в index и _NTO_COF_CLOEXEC в аргументе flags, наследуется дочерним процессом как дескриптор файла (путём дублирования дескрипторов файлов родителя). В процессе дублирования соединения как дескриптора файла серверу посылается в такой канал системное сообщение _IO_DUP (первые 2 байта этого сообщения есть 0x115), в то время как сервер такого сообщения не ожидает.

Таким образом, для гарантии создания процессом-клиентом уникального соединения именно с каналом процесса-сервера настоятельно рекомендуется в index задавать системное значение _NTO_SIDE_CHANNEL. Это обеспечивает получение для соединения значения ID большего, чем значение ID любого существующего дескриптора файла.

Соединения, принадлежащие клиенту, могут одновременно разделяться любой нитью клиента. Если клиент создаёт параллельные соединения к одному и тому же каналу сервера, то система ведёт каждое соединение отдельно.

10.8.2.2. Разрыв соединения

Соединение разрывается с помощью функции

```
#include <sys/neutrino.h>
int ConnectDetach(int coid);
```

В качестве аргумента coid выступает ID соединения. Если во время разрыва соединения какие-либо нити были блокированы в результате посылки сообщения по этому соединению, то нити разблокируются, а посылка сообщения завершается с ошибкой. В случае ошибки функция возвращает -1, а в errno помещается код ошибки. Если выполнение успешное, то возвращается произвольное значение отличное от -1.

Аннулирование соединений, когда необходимость в них отпадает, необходимо обязательно выполнять, так как ресурсы ядра, связанные с поддержанием соединений, не безграничны.

Рассмотрим фрагмент создания соединения с каналом (идентификатор которого равен 1, принадлежащим процессу-серверу с идентификатором равным 77 и находящемуся на одном узле сети с процессом-клиентом), и его последующего разрыва:

```
#define default 0 //
int chid = 1;     //ID канала
pid_t pid = 77;//ID процесса
int coid;         //ID соединения с каналом
int index = _NTO_SIDE_CHANNEL;    //гарантия уникальности ID соединения с каналом
int flags = default;      //установить системные свойства соединения по умолчанию
…
// установить соединение
coid=ConnectAttach(ND_LOCAL_NODE, pid , chid, _NTO_SIDE_CHANNEL, flags);
…
//соединение разорвать
ConnectDetach(coid);
…
```

10.9. Передача сообщений

10.9.1. Посылка сообщения

Посылку сообщения в канал сервера выполняет клиент. Предварительно клиент вызовом ConnectAttach() создаёт соединение coid с каналом сервера (предполагается, что необходимые для этого значения nd, pid и chid сервера ему известны). Посылка в канал сообщения осуществляется с помощью функции:

```
#include <sys/neutrino.h>
int MsgSend(int coid, const void* smsg, int sbytes, void* rmsg, int rbytes);
```

Посылаемые данные берутся из буфера, указанного smsg. Предполагается, что сервер, приняв сообщение, выполнит соответствующее действие, и отправит ожидаемое клиентом ответное сообщение в буфер, указанный в rmsg. Число байтов в посылаемом сообщении задаётся в sbytes, а число байтов в ожидаемом ответе задаётся в rbytes.

Количество переданных байтов из буфера сообщения клиента в буфер сообщения сервера определяется минимальным размером буферов сообщения, используемых клиентом и сервером. Это гарантирует от переполнения буфера сервера для приёма сообщения от клиента. Такая же гарантия имеет место для клиента при получении ответа от сервера.

Если процесс-сервер имел нить, которая ожидала прихода сообщения (была RECEIVE-Блокирован на этом канале), то перенос сообщения в адресное пространство сервера осуществляется, немедленно, а принимающая сообщение нить сервера становится готовой для выполнения. Посылающая сообщение нить клиента при этом становится REPLY-блокированной. Если нити, ожидающей приёма сообщения из данного канала, в сервере нет, то пославшая сообщение нить клиента становится SEND-блокированной и ставится в очередь к каналу в порядке приоритета вместе с другими нитями, так же пославшими сообщение в этот канал. Фактический перенос данных из адресного пространства клиента в адресное пространство сервера не происходит до тех пор, пока принимающая нить сервера не выполнит функцию получения данных из канала. После этого нить клиента, пославшая данные, становится REPLY-блокированной (ждёт ответного сообщения).

В случае успешного выполнения функция MsgSend() возвращает значение статуса, заданного в аргументе status функции MsgReply(), которую выполняет нить сервера, посылающая ответ клиенту. Если возникает ошибка выполнения вызова, то функция возвращает -1 и в errno ядро устанавливает код ошибки вызова. Если вызов ядром выполнен успешно, но нить сервера вместо MsgReply() использовала функцию MsgError(), то функция возвращает -1, но errno получает значение ошибки, заданное в MsgError().

Функция MsgSend() принадлежит семейству функций MsgSend*() и семантически связана с функциями ConnectAttach(), TimerTimeout() и функциями семейства MsgReceiv*().

Рассмотрим пример передачи сообщения процессу с ID процесса равным pid в канал с ID канала равным chid.

Пример.

```
#include <sys/neutrino.h>
#include <strino.h>
#define WIDTH 80

char *smsg="Это содержимое буфера сообщения";
char rmsg[200]= "\0"; //Это пустой буфер ответа
int coid;
...
/* pid – ID процесса, chid – ID канала */
...
//Установить соединение
coid=ConnectAttach(ND_LOCAL_NODE, pid, chid, _NTO_SIDE_CHANNEL, 0);
if(coid==-1){
    fprintf(stderr,"Ошибка соединения\n")
    exit(EXIT_FAILURE);
}
//Послать сообщение
if(MsgSend(coid, smsg, strlen(smsg)+1, rmsg, sizeof(rmsg)) == -1){
    fprintf(stderr,"Ошибка MsgSend\n")
    exit(EXIT_FAILURE);
```

```
}
if(strlen(rmsg)>0) printf("Сервер ответил \n%s\n",rmsg);
…
```

10.9.2. Приём сообщения

Процесс-сервер должен принять сообщение и послать ответ клиенту. Для приёма сообщения сервером используется функция:

```
#include <sys/neutrino.h>
int MsgReceive(int chid, void *msg, int bytes, struct _msg_info *info);
```

Если канал пуст, то эта функция переводит сервер в RECEIVE-блокированное состояние ожидания поступления сообщения в канал chid и его помещения в буфер, адрес которого указан в msg. Размер буфера msg задаётся в bytes в байтах. Число принятых в буфер msg байтов не может превысить значения bytes (контролируется ядром).

Если сообщение уже поступило в канал перед выполнением сервером вызова MsgReceive(), то оно немедленно копируется ядром в адресное пространство сервера, и сервер не блокируется в состоянии ожидания. Если сообщения в канале нет, то выполнившая вызов MsgReceive() нить сервера переходит в RECEIVE-блокированное состояние, ожидая пока сообщение от клиента не поступит в канал. При получении сообщения нить переходит в состояние READY - готова к выполнению.

Если значение указателя info отлично от NULL, то он ссылается на структуру, в которой сохраняется дополнительная информация относительно сообщения и нити клиента, которая выполнила функцию MsgSend(). Если значение info равно NULL, то эта информация, при необходимости, может быть получена, с помощью функции MsgInfo() (справочное описание этой функции содержит и определение структуры _msg_info).

Если при выполнении функции возникает ошибка, то возвращается -1 и errno присваивается код ошибки. В случае успеха возвращается идентификатор rcvid, специфицирующий нить клиента, пославшую сообщение.

10.9.3. Посылка ответа

Получив сообщение от клиента сервер должен послать ему ответное сообщение с помощью функции:

```
#include <sys/neutrino.h>
int MsgReply(int rcvid, int status, const void* msg, int size);
```

rcvid - ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive().
status - статус завершения, который возвращается нити клиента, пославшей сообщение, при успешном завершении функции MsgSend().
msg - указатель буфера, содержащего ответное сообщение.
size - размер сообщения в байтах.

Функция посылает ответ с сообщением нити клиента, идентифицированной rcvid. При этом нить должна находится в REPLY-блокированном состоянии. Ответ может послать любая нить сервера. Важно только, чтобы на каждое принятое сервером сообщение следовал бы ответ и только один. Выполнение функции MsgSend(), вызванной нитью клиента, которой

соответствует rcvid, завершается разблокированием нити и возвратом функцией MsgSend() значения статуса, заданного сервером в аргументе status при выполнении функции MsgReply().

Ядро контролирует, чтобы число байтов ответного сообщения, принимаемого клиентом, не превышало объёма буфера клиента, предназначенного для приёма ответа.

Функция MsgReply() не блокирует нить сервера, передача ответа выполняется немедленно. Нет никаких особых требований к своевременности ответа, но в конечном итоге серверу необходимо ответить на каждое принятое сообщение, чтобы нить клиента, пославшая сообщение, вышла из REPLY-блокированного состоянии.

Заметим, что в функциях MsgSend() и MsgReceive() указывается количество посылаемых и принимаемых байт. Если они не совпадают, то выбирается минимальное из указанных значений с целью согласования размеров буферов передачи и приёма. Аналогичная ситуация и с функцией MsgReply(). То есть, при необходимости сообщение будет "урезано" и лишние байты "отброшены". Если возникает ошибка, то возвращается -1 и errno присваивается код ошибки.

Пример.

```
#include <sys/neutrino.h>
…
void server(void){
    int rcvid;
    int chid;
    char message[512];
//Создать канал
      chid=ChannelCreate(0);
      //Бесконечный цикл
      while(1){
      //Получить и вывести сообщение
      rcvid=MsgReceive(chid,message,sizeof(message),NULL);
      printf("Получил сообщение, rcvid = %X\n",rcvid);
      printf("Сообщение такое: %s\n", message);

/*Подготовить и отправить ответ - используем тот же буфер, что и для приёма сообщения*/
      strcpy(message,"Это ответ");
      MsgReply(rcvid,EOK,message,sizeof(message));
      }
}
```

10.9.4. Сценарии ответов

Использование MsgReply() достаточно прозрачно. Но за внешней простотой скрывается принципиальная возможность управления активностью клиента со стороны сервера. Во-первых, сервер совершенно не обязан посылать ответ клиенту как можно быстрее, и вообще никак не ограничен во времени с ответом клиенту. Поэтому сервер, при необходимости, может целенаправленно управлять временем нахождения клиента в REPLY-блокированном состоянии. При этом сервер может принимать и обрабатывать сообщения по тому же каналу от

других клиентов (возможно, от других нитей того же процесса-клиента) и как-то отсылать им ответы. Во-вторых, ответ сервера может быть пустым и преследовать цель только разблокировать клиента, например, MsgReply(rcvid, EOK, NULL, 0).

Если же серверу необходимо проинформировать клиента о проблемах, возникших при обслуживании полученного от клиента сообщения, то в этом случае для посылки клиенту ответа удобно воспользоваться специальной функцией:

```
#include <sys/neutrino.h>
int MsgError(int rcvid, int error);
```

rcvid - ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive(), когда сообщение получено сервером.

error - код ошибки, отсылаемый клиенту.

Функции MsgError() выполняет разблокирование нити клиента, вызвавшей MsgSend(), и устанавливает errno клиента в значение error. Если error имеет значение системного кода EOK, то функция MsgSend() возвращает EOK. Если error имеет любое другое значение, то функция MsgSend() возвращает (-1). Важно учесть, что значение error, равное системному коду ERESTART, заставляет клиента немедленно повторить вызов MsgSend(). Однако заметим, что этот код нельзя использовать серверу после вызова MsgWrite() при отправке ответа частями (см. ниже).

10.9.5. Сообщения типа "импульс"

Импульс — это специальное сообщение системного типа struct _pulse, посылка которого осуществляется с помощью функции MsgSendPulse():

```
#include <sys/neutrino.h>
int MsgSendPulse ( int coid, int priority, int code, int value );
```

Функция посылает в канал сервера, по соединению coid короткое *неблокирующее* системное сообщение. Импульс имеет тип следующую системной структуры:

```
struct _pulse {
_uint16         type;
_uint16         subtype;
_int8           code;
_uint8          zero[3];
union sigval    value;
_int32          scoid;
};
```

Поля структуры type и subtype равны нулю (признак импульса). Содержимое полей code и value задаются отправителем. Обычно code указывает причину, по которой был отправлен импульс, а value содержит 32 бита данных, посылаемых с импульсом (т.е. всего 40 бит). Код может быть 8-битным значением меньшим ноля, чтобы избежать конфликта с ядром или менеджерами QNX, генерирующими системные импульсы. Ядро предоставляет прикладным процессам для использования по своему усмотрению 127 отрицательных значений code, которые заключены в диапазоне от _PULSE_CODE_MINAVAIL до

_PULSE_CODE_MAXAVAIL
(-127÷-1). Элемент value имеет тип объединения вида:

```
union sigval{
        int sival_int;
        void *sival_ptr;
        };
```

Это означает, что доставленное импульсом значение типа union sigval серверу необходимо явно типизировать как целое или указатель неопределённого типа.

Импульс посылается с указанием приоритета. Параметр priority должен быть в пределах диапазона правильных приоритетов в диапазоне от sched_get_priority_min() до sched_get_priority_max().

Посылка процессом-клиентом импульса и его приём процессом-сервером имеет существенные особенности. Посылка импульса не блокирует нить процесса-клиента. Приём импульса сервером выполняется как приём обычного сообщения. Отличие только в том, что функция MsgReceive() возвращает ноль (признак прихода импульса) и не требуется посылать ответ, используя функцию MsgReply(). С импульсом можно передать только 40 бит полезной информации (8-битный код и 32 бита данных).

Если серверу требуется принимать только импульсы, оставляя без внимания все другие сообщения, то в этом случае необходимо использовать функцию MsgReceivePulse():

```
int MsgReceivePulse(int chid, void *rmsg, int rbytes, struct msg_info *info);
```

Заметим, что параметр info *всегда* равен NULL. Если по некоторому каналу принимаются и обычные сообщения, и импульсы, и при этом на нем блокированы нити, выполнившие функцию MsgReceivePulse(), и нет ни одной нити, заблокированной при выполнении функции MsgReceive(), то импульсы будут обслуживаться, а обычные сообщения обслуживаться не будут. Клиенты, пославшие обычные сообщения, будут SEND-блокированными до тех пор, пока какая-либо нить сервера не выполнит функцию MsgReceive(). Но при этом она может принять как обычное сообщение, так и импульс! Поэтому в этом случае необходимо обязательно контролировать возврат функцией MsgReceive() нулевого значения как признака приёма импульса.

Типичным программным фрагментом процесса-сервера, обрабатывающего импульсы, является:

```
#include <sys/neutrino.h>
#define MY_PULSE_TIMER …
struct _pulse *pulse;
char msg[…];
…
rcvid=MsgReceive(chid, msg, …);
if(rcvid==0){//Пришёл импульс
        pulse = (struct _pulse *) msg;
        //Определить тип импульса
        switch(pulse->code){
        case MY_PULSE_TIMER://Импульс – уведомление от таймера
```

```
            …
            break;
            //и так далее
}
else {// Обработать обычное сообщение
…
}
```

10.10. Управление сообщениями неопределённой длины

Функции MsgSend(), MsgReceive(), MsgReply() должны указывать буфер фиксированной длины для приёма/передачи сообщения. Однако не всегда имеется возможность заранее знать предельную длину сообщений. Сервер, например, может не знать, какие по длине сообщения ему придётся принимать от клиентов, а также какие по длине ответы будут формироваться в результате обработки принятых сообщений. В таких случаях серверу может потребоваться осуществлять приём/передачу сообщений и ответов по частям, используя при приёме сообщения наряду с функцией MsgReceive() ещё и вызовы функции MsgRead(), а при посылке ответа - вызовы функции MsgWrite(), прежде чем будет выполнена функция MsgReply().

10.10.1. Управление приёмом сообщений

Если сервер не располагает буфером, способным всегда целиком разместить поступающие сообщения, то он должен удостовериться, что принял сообщение целиком, а именно - выяснить длину посланного клиентом сообщения. Такую информацию функция MsgReceive() предоставляет серверу посредством аргумента info, если при создании канала был установлен флаг _NTO_CHF_SENDER_LEN.

Аргумент info является структурой типа _msg_info:

```
struct _msg_info{
    int nd;//ID принимающего узла
    int srcnd; //ID передающего узла
    pid_t pid; //ID клиента
    int32_t chid; //ID канала
    int32_t scoid; //внутренний системный ID, используемый ядром
    int32_t coid; //ID соединения
    int32_t msglen; //количество принятых сервером байтов сообщения
    int32_t tid; //ID нити, пославшей сообщение
    int16_t priority; //приоритет нити, пославшей сообщение
    int16_t flags; //дополнительные информационные флаги
    int32_t srcmsglen; /*длина посланного сообщения в байтах (это поле актуально, если при
                выполнении функции ChennelCreat() был установлен флаг
                _NTO_CHF_SENDER_LEN)*/
}
```

Чтобы выяснить, целиком ли принято посланное клиентом сообщение, серверу достаточно сравнить значения полей msglen и srcmsglen аргумента info. Если msglen < srcmsglen, то сервер принял только часть посланного сообщения, которая

уместилась в буфере сервера. Остальная часть осталась в буфере клиента. Используя функцию MsgRead()сервер имеет возможность по частям дополучить сообщение.

Определение функции MsgRead() следующее:

```
#include <sys/neutrino.h>
int MsgRead(int rcvid, void* msg, int bytes, int offset);
```

Аргументы функции:
rcvid - ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive(),
msg - буфер сервера для приёма части сообщения,
bytes - длина принимаемой сервером части сообщения,
offset - смещение относительно начала сообщения в буфере клиента.

Выполнение сервером функции MsgRead() оставляет соответствующую нить клиента в REPLY-блокированном состоянии. Поэтому сервер может многократно выполнять эту функцию до получения всего посланного нитью сообщения по частям, выделяя (например, динамически) буферы для каждой части или по ходу обрабатывая принятую часть сообщения в одном и том же буфере. Когда приём и обработка сообщения полностью завершается, сервер, как и прежде, должен выполнить функцию MsgReply() для вывода нити клиента из REPLY-блокированного состояния.

10.10.2. Управление передачей ответа

Кроме приёма по частям сообщения от клиента сервер может по частям передавать и ответ клиенту. Однако при этом важно, чтобы клиент располагал буфером по величине достаточным для приёма целиком ответа, отправляемого по частям. В противном случае ответ будет принят только частично в объёме буфера, выделенного клиентом для приёма ответа,.

Для передачи клиенту ответа по частям сервер должен перед выполнением функции MsgReply() использовать функцию MsgWrite(), которая имеет прототип:

```
#include <sys/neutrino.h>
int MsgWrite(int rcvid, void* msg, int bytes, int offset);
```

Эта функция пишет данные в буфер ответа нити, ID которой указан в rcvid - возвращается функцией MsgReceive() при получении сообщения сервером. Нить, по отношению к которой выполняется запись, должна быть в REPLY-блокированном состоянии. Выполнение MsgWrite() не выводит нить из REPLY-блокированного состояния.

Данные в количестве bytes байтов берутся из буфера сервера, указанного в msg, и записываются в буфер ответа клиента, начиная с места, отстоящего от начала буфера на величину offset байт (смещение от начала буфера).

Если размер ответа bytes превышает размер буфера ответа клиента, то его переполнения не произойдёт, а превышающая размер буфера часть ответа просто не будет клиентом получена.

Чтобы закончить передачу ответа и вывести клиента из REPLY-блокированного состояния, серия вызовов MsgWrite() должна быть завершена вызовом функции MsgReply(). При этом ответ, отправляемый функцией MsgReply() не должен обязательно содержать какие-либо данные. Если он все-таки содержит данные, то они будут всегда записываться с нулевым смещением в буфере ответа нити назначения. Это - удобный способ записи заголовка ответа, когда он полностью передан.

Функция MsgWrite() возвращает число реально переданных байтов ответа. В случае ошибки возвращается -1 и устанавливается errno.

10.10.3. Передача сообщений с использованием векторов ввода/вывода

Если сообщение состоит из несвязных частей, то его передача может осуществляться с использованием так называемых векторов ввода/вывода. Под вектором ввода/вывода (IOV) понимается структура, которая содержит два поля - адрес и длину части сообщения. Для определения IOV используется системный тип iov_t, имеющий определение вида:

```
typedef struct iovec{
void *iov_base;//адрес сообщения
size_t iov_len; //длина сообщения
} iov_t;
```

Для инициализации значения IOV удобно использовать системную макрокоманду:

```
SETIOV(_iov, base, _len);
```

где:
_iov - имя вектора ввода/вывода;
base - адрес части сообщения;
_len - длина части сообщения в байтах.

При передаче несвязного сообщения для каждой части такого сообщения формируется свой вектор ввода/вывода. Затем из них формируется массив векторов ввода/вывода, в котором вектора располагают в нужном порядке следования частей сообщения при передаче.

Для посылки клиентом сообщения, части которого находятся в несвязной области памяти, представленной массивом векторов IOV, применяется функция:

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

int MsgSendv(int       coid,
             const iov_t*    siov, //Массив векторов сообщения
             int       sparts, //Количество векторов в сообщении
             const iov_t*    riov, //Массив векторов ответа
             int       rbytes); //Количество векторов в ответе
```

Для приёма сервером сообщения в несвязную область памяти (буфер) с использованием IOV применяется функция:

```
#include <sys/neutrino.h>
int MsgReceivev(int chid,
                const iov_t * riov,  //Массив IOV буфера приёма
                int rparts,    //Количество IOV в массиве
                struct _msg_info *info );   //Доп. информация
```

С помощью векторов можно организовать и передачу сервером ответа клиенту. Для этого используется функция:

```
#include <sys/neutrino.h>
int MsgReplyv(int rcvid,
              int status,      //Статус ответа
```

```
const iov_t* riov,     //Массив IOV буфера ответа
int rparts);    //Количество IOV в массиве
```

Заметим, что клиент и сервер не обязаны согласовывать способ передачи/приёма сообщения, т.е. структура буфера сообщения и ответа клиента, а также буфера приёма и ответа сервера не обязаны совпадать. Например, клиент может использовать для посылки сообщения функцию MsgSend(), а сервер – MsgReceivev(), и наоборот.

11. Организация взаимодействия процессов в сети

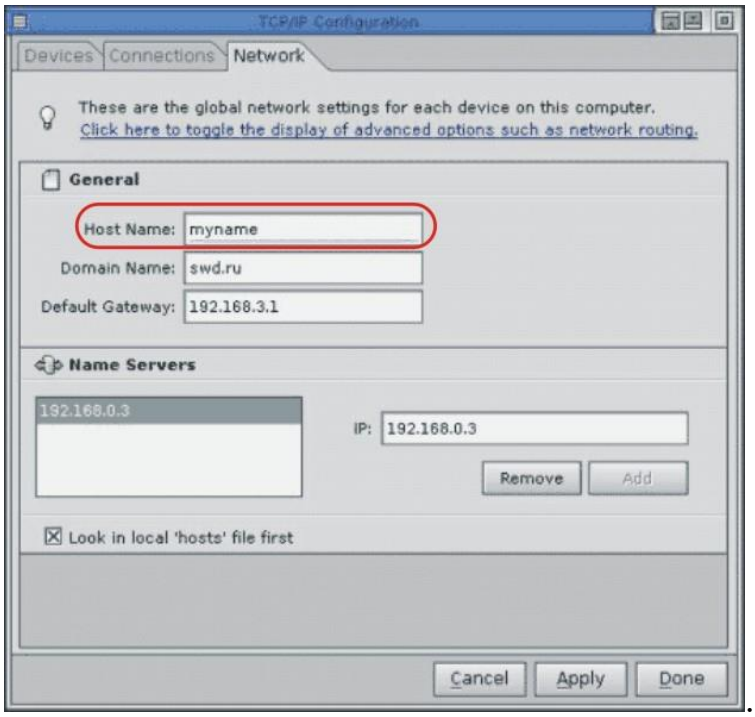
11.1. Сетевая концепция QNX

QNX изначально разрабатывалась как сетевая операционная система. Обычно локальная вычислительная сеть реализует механизм разделения файлов и внешних устройств между несколькими взаимосвязанными компьютерами. В QNX эта концепция получила дальнейшее развитие, в результате чего вся сеть стала представлять собой единый разделяемый набор ресурсов [11].

Любой процесс на любой машине сети может использовать любой ресурс любой другой машины. Для приложения нет никакой разницы между своим или удалённым ресурсом. Приложению не требуется иметь никаких специальных средств для обеспечения доступа к удалённому ресурсу. Пользователи имеют доступ ко всем файлам сети, могут использовать любое внешнее устройство и запускать приложения на любой машине сети (при условии, что они имеют на это соответствующее полномочие). Соответственно, все процессы могут взаимодействовать между собой по всей сети. Таким образом, механизм передачи сообщений в QNX, обеспечивает гибкую и прозрачную сетевую обработку.

11.2. Сетевая настройка QNX

Для работы в сети QNX располагает собственным сетевым протоколом - Qnet. Для настройки компьютера (узла) для работы в сети ему необходимо присвоить символическое имя (идентификатор). В среде PHOTON для этого нужно выполнить последовательно команды Launch -> Configure -> Network. В результате откроется окно с именем TCP/IP Configuration, в котором следует выбрать закладку Network:



На этой закладке в поле HostName вводим символическое имя компьютера. Введённое имя узла (в данном случае - myname), будет использоваться протоколом Qnet. Никаких других настроек для Qnet не требуется.

Для присоединения компьютера в качестве узла к локальной сети (монтирование сети) и запуска *администратора сети* (системного процесса npm-qnet, реализующего протокол Qnet для работы в сети) необходимо в окне терминала выполнить команду:

```
#mount –T io-net /lib/dll/npm-qnet.so
```

Для автоматического монтирования сети при запуске ОС необходимо эту команду включить в командный файл /etc/rc.d/rc.local. У этого файла должен быть установлен атрибут выполняемого файла (executable). При наличии этого файла он выполняется при загрузке ОС. Содержимое файла /etc/rc.d/rc.local может выглядеть следующим образом:

```
#!/bin/sh
mount –T io-net /lib/dll/npm-qnet.so
```

Если монтирование прошло успешно, на этом настройка сети закончена. В файловой системе локального узла появляется каталог /net, который содержит в себе каталоги с именами, соответствующими примонтированным к сети узлов. После этого доступ к ресурсам узла сети можно получить по пути /net/<имя_узла>/<имя_ресурса>.

Если узлу необходимо отсоединиться от сети, то следует выполнить команду:

```
umount /dev/io-net/en0
```

11.3. Организация взаимодействия процессов в сети

11.3.1. Особенности обмена сообщениями в сети

Существенным отличием организации обмена сообщениями между процессом-клиентом и процессом-сервером, находящихся на разных узлах локальной сети является то, что в этом принимают участие одновременно два ядра ОСРВ с администраторами npm-qnet на узле клиента и узле сервера. На узле клиента администратор npm-qnet участвует в обслуживании перенаправляемых ядром ОС запросов клиента, как на установление соединения, так и на посылку сообщений, а на узле сервера – участвует в обслуживании запросов сервера на приём сообщения и посылку ответа. Это приводит к необходимости учитывать следующие особенности запуска удалённых дочерних процессов и обмена сообщениями в сети:

– Запущенный на удалённом узле дочерний процесс не может получить pid своего удалённого родительского процесса с помощью функции getppid(), так как опосредованно им становится администратор сети npm-qnet, получивший и перенаправивший запрос на запуск дочернего процесса ядру своего узла, при передаче клиентом сообщения по сети, удалённый сервер реально получает сообщение от местного администратора сети npm-qnet.

– Для установления соединения с удалённым процессом функция ConnectAttach() требует указать отличное от нуля значение дескриптора удалённого узла (0 – дескриптор локального узла).

– Запрос процессом дескриптора удалённого узла с заданным именем компьютера терпит неудачу, если связь нарушена, или компьютер не в сети.

– Полученное процессом на местном узле каким-либо способом значение дескриптора удалённого узла в общем случае может, при определённых условиях, потерять актуальность

(например, при нарушении соединения с каналом), поэтому если на местном узле необходимо гарантированно "запомнить" удалённый узел, то следует запоминать имя компьютера удалённого узла в сети, по которому, при необходимости, можно получать актуальный дескриптор узла.

– Возвращённый функцией ConnectAttach() признак успешного соединения не является надёжной гарантией актуальности соединения в дальнейшем, отсутствие соединения (нарушен кабель, отключилась машина и т.д.) может выявиться при любой попытке передать сообщение, функция передачи сообщения возвращает сообщение об ошибке, а соединение аннулируется, повторное установление соединения в направлении этого узла вновь потребует получения актуального значения дескриптора этого узла, так как ранее полученное значение не является постоянно закреплённым за этим узлом.

– Функции MsgReply(), MsgRead(), MsgWrite() и им подобные при взаимодействии удалённых клиента и сервера в сети становятся блокирующими вызовами, т.к. при их реализации используются услуги администратора npm-qnet, которому ядро перенаправляет вызов как процессу-серверу. Выход из блокирующего состояния происходит, когда администратор npm-qnet либо корректно, либо с ошибкой завершает доставку сообщения.

– Когда завершается запрос MsgReceive(), сервер может и не получить от клиента сообщение в полном объёме даже при наличии у сервера достаточного объёма буфера приёма сообщения. Это опять связано с тем, что перенаправленное ядром сообщение клиента предварительно принимает в собственный буфер администратор сети npm-qnet на узле клиента, который затем передаёт принятое сообщение администратору сети npm-qnet на удалённом узле сервера. Так как размер сообщения заранее администратору npm-qnet не известен, то он за один раз получает объем данных, не превышающий фиксированного максимального размера его буфера (в настоящее время 8KB).

Из последнего замечания следует, что, если клиент посылает, например, 1 Мбайт данных и сервер использует вызов MsgReceive() с буфером приёма в 1 Мбайт, то администраторы npm-qnet в узлах сети за один раз смогут передать/принять только часть сообщения, поместившуюся в ограниченный буфер. При этом число байтов, фактически переданных серверу, фиксируется в аргументе info функции MsgReceive() в поле msglen структуры struct _msg_info, или его можно получить с помощью функции MsgInfo(). Если выясняется, что сообщение получено не полностью, его остаток необходимо получать по частям, последовательно принимая части сообщения, размер которых не превышает возможностей администратора npm-qnet, пока не будет получено все сообщение. Для приёма последующих частей сообщения после выполнения функции MsgReceive() необходимо нужное число раз использовать функцию MsgRead().

Например, можно в цикле воспользоваться следующим кодом, гарантирующим получение от клиента целиком всего сообщения.

```
…
chid=ChannelCreate(_NTO_CHF_SENDER_LEN);/*информировать о длине переданного
                                        сообщения*/
…
rcvid=MsgReceive(chid, msg, nbytes ,&info);/*в поле info.msglen содержится длина принятого
                                        сообщения*/
```

```
/* Проверяется, всё ли сообщение было принято*/
if (rcvid>0 && info.srcmsglen>info.msglen && info.msglen<nbytes){ /*в поле info.srcmsglen длина
                                                                  передаваемого
                                                                  сообщения*/
        /* Цикл приёма остатка сообщения. */
int n;
        if((n=MsgRead(rcvid,(char*)msg+info.msglen, nbytes-info.msglen,info.msglen))<0){
                MsgError(rcvid,-n);
                continue; //повторить передачу ошибочно принятого блока сообщения
        }
info.msglen+=n;
}
```

11.3.2. Определение дескрипторов удалённых узлов сети

Для указания дескриптора местного узла достаточно использовать системную константу ND_LOCAL_NODE (её значение есть 0). Она определена в заголовочном файле <sys/netmgr.h>. При взаимодействии через сеть требуется указывать отличное от нуля значение дескриптора узла сети - nd, используемого функцией ConnectAttach(uint32_t nd,...) в качестве первого параметра. В сети QNX 6 узел глобально представлен только своим именем (символьное имя, присвоенное компьютеру в сети). Значение дескриптора узла, соответствующего компьютеру с указанным именем, является относительным и может быть получено процессом в локальном узле с помощью функции:

```
#include <sys/netmgr.h>
int netmgr_strtond(const char *nodename, char **endstr);
```

Функция определяет местное значение дескриптора удалённого узла, соответствующее указанному имени узла. Если аргумент endstr не NULL, то в качестве его значения необходимо использовать адрес указателя, ссылающегося на значение байта, отличного от '\0', следующего за именем узла в строке nodename.

Функция возвращает значение дескриптора узла, или -1, если произошла ошибка. В случае ошибки устанавливается значение системной переменной errno.

Полученный в результате дескриптор узла nd не является уникальным для всей сети, а лишь для текущего узла, на котором он был получен. Значение дескриптора, например, nd=5, на одной машине может соответствовать узлу с именем "A", а такое же значение дескриптора на другой машине - узлу с именем "B". Более того, значения дескрипторов узлов, в общем случае, носят временный характер. То есть, при отсоединении узла от сети и повторном соединении актуальность, полученного ранее дескриптора удалённого узла, теряется. Кроме того, полученный дескриптор удалённого узла, но не используемый ни в одном соединении, администратор сети может при необходимости переназначить другому удалённому узлу. Следовательно, получать дескриптор удалённого узла "заранее" не имеет смысла, его необходимо немедленно использовать для установления соединения с каналом процесса на этом удалённом узле. Пока есть актуальные соединения дескриптор удалённого узла на локальном узле не изменяется.

Имя удалённого узла может быть получено различными путями. Если, например, известен актуальный для локального узла дескриптор некоторого удалённого узла, то имя этого удалённого узла может быть получено с помощью функции

int netmgr_ndtostr(unsigned flags, int nd, char buf, size_t buflen);

По умолчанию (flags равен 0) функция помещает в буфер buf размером buflen байт строку, представляющую собой абсолютное (полное) имя узла, соответствующего указанному дескриптору узла nd. Эту строку можно послать любому другому узлу для использования её в функции netmgr_strtond() для получения локального значения дескриптора удалённого узла с этим именем.

Для получения короткого имени локального узла можно воспользоваться функцией

#include <unistd.h>
size_t confstr(int namvar, char buf, size_t buflen);

Функция confstr() позволяет получить строку размером buflen байт, помещаемую в буфер buf, со значением параметра системной конфигурации, определённого в namvar. Для получения короткого имени узла в сети в качестве параметра namvar следует указать системную константу _CS_HOSTNAME.

Существует возможность получить дескриптор нужного узла опосредованно, не зная имени узла. Если на некотором узле C возникла необходимость получить значение дескриптора удалённого узла A, и ему известно, какое локальное значение имеет дескриптор узла A на удалённом узле B, а также собственное локальное значение дескриптора узла B, тогда можно получить собственное локальное значение дескриптора узла A, используя функцию:

int netmgr_remote_nd(int local_nd_B_in_C, int local_nd_A_in_B)

Функция возвращает собственное локальное значение дескриптора узла A на узле C.

Если необходимо убедиться, что два дескриптора являются идентичными, можно воспользоваться макрокомандой

ND_NODE_CMP(nd1,nd2);

Если возвращаемое значение является нулевым, дескрипторы nd1 и nd2 относятся к одному и тому же узлу. Если значение меньше или больше нуля, то дескрипторы разные. При необходимости это можно использовать для сортировки дескрипторов узлов.

11.3.3. Запуск процесса на удалённом узле

Рассмотренные ранее функции стандартной библиотеки C не предназначены для запуска процессов на удалённом узле. Для этого в QNX необходимо использовать системную функцию spawn() [9][10][12]. Эта функция предоставляет максимальные средства управления запуском дочернего процесса как локально, так и в сети. Функция имеет вид:

```
#include <spawn.h>
pid_t spawn( const char *path,
             int fd_count,
             const int fd_map[],
             const struct inheritance *inherit,
             char *const argv[],
             char *const envp[] );
```

Аргументы:

path - полное имя исполняемого модуля, используемого родителем для запуска дочернего процесса.

fd_count - число элементов в массиве fd_map.

fd_map - массив ограниченного набора дескрипторов файлов, открытых в родительском процессе, которые разделяет с родителем дочерний процесс. Если fd_count не равен 0, то размер заданного массива fd_map должен позволять предоставить дочернему процессу для использования, по крайней мере, fd_count дескрипторов открытых родителем файлов, но не может быть более величины OPEN_MAX – максимальное количество дескрипторов открытых родителем файлов, предоставляемых дочернему процессу для совместного использования. Если значение аргумента fd_count равно 0, то массив fd_map игнорируется, но при этом все дескрипторы файлов, кроме тех, для которых в дескрипторе установлен флаг FD_CLOEXEC (установлен непосредственно при создании файла или модифицирован функцией fcntl()), наследуются дочерним процессом.

inherit - структура системного типа для настройки свойств дочернего процесса:

```
struct inheritance {unsigned long flags;
                    pid_t pgroup;
                    sigset_t sigmask;
                    sigset_t sigdefault;
                    uint32_t nd}//дескриптор узла
```

которая используется для управления запуском и формирования свойств дочернего процесса, наследуемых от родительского процесса. Значения полей, следующие:

unsigned long flags - аргумент для установки флагов управления запуском и наследованием дочерним процессом свойств родительского процесса. Используются следующие флаги:

SPAWN_SEARCH_PATH - если не указано полное имя программного модуля (файла) в файловом пространстве, то установка флага предписывает осуществлять поиск программного модуля в каталогах, заданных в переменной среды PATH.

SPAWN_SETGROUP - установка флага предписывает включить дочерний процесс в группу с GID, заданным в поле pgroup. Если этот флаг не установлен, дочерний процесс становится членом текущей группы процессов.

SPAWN_SETND - установка флага предписывает запустить дочерний процесс на узле с дескриптором, заданным в поле nd.

SPAWN_SETSIGDEF - установка флага предписывает использовать поле sigdefault, чтобы определить для дочернего процесса набор сигналов с действиями по умолчанию. Если этот флаг не установлен, дочерний процесс наследует сигнальные действия родительского процесса.

SPAWN_SETSIGMASK - установка флага предписывает использовать поле sigmask для задания маски сигналов дочернего процесса.

pid_t pgroup - если в inherit.flags установлен флаг SPAWN_SETGROUP, то задаёт значение GID дочернего процесса. Если же pgroup присвоить значение SPAWN_NEWPGROUP, то дочерний процесс начинает новую группу с GID группы, равным ID дочернего процесса.

sigset_t sigmask – предназначен для задания маски сигналов дочернего процесса, если в inherit.flags установлен флаг SPAWN_SETSIGMASK.

sigset_t sigdefault - если в inherit.flags установлен флаг SPAWN_SETSIGDEF, определяет набор сигналов дочернего процесса с действиями по умолчанию.

uint32_t nd - если в inherit.flags установлен флаг SPAWN_SETND, то аргумент задаёт дескриптор удалённого узла, где запускается дочерний процесс.

argv -вектор аргументов. Значение argv не может быть равным NULL. Если аргументов нет, то argv[0] равен NULL. Если argv[0] не равен NULL, а количество передаваемых дочернему процессу аргументов равно argc, то argv[0] должно указывать на абсолютное имя файла, содержащего программный модуль. Последний элемент - argv[argc+1], должен быть NULL.

envp - вектор указателей строк с определением переменных среды. Вектор заканчивается указателем NULL. Каждый указатель указывает на строку вида:

<имя переменной среды> = <значение - строка символов>,

которая определяет переменную среды. Если значение envp равно NULL, то дочерний процесс наследует окружающую среду от родителя.

Функция spawn() порождает и запускает новый дочерний процесс, на основе исполняемого модуля, который содержится в файле с именем path.

Дочерний процесс наследует следующие атрибуты родительского процесса:

- ID группы процесса, если SPAWN_SETGROUP не установлен в inherit.flags (для дочернего процесса на локальном узле).
- Принадлежность сеансу (для дочернего процесса на локальном узле).
- Реальный ID пользователя и реальный ID группы.
- ID дополнительной группы.
- Приоритет и дисциплину диспетчеризации.
- *Текущий корневой и рабочий каталог.*
- Маску создания файла.
- Маску сигналов (если SPAWN_SETSIGMASK не установлен в inherit.flags).
- Сигнальные действия, специфицированные как SIG_DFL.
- Сигнальные действия, специфицированные как SIG_IGN (за исключением изменённых inherit.sigdefault, когда SPAWN_SETSIGDEF установлен в inherit.flags).

Дочерний процесс имеет некоторые отличия от родительского процесса при его запуске как на локальном, так и на удалённом узле:

- Набор сигналов, которые обрабатываются родительским процессом, установлены по умолчанию (SIG_DFL).
- Значения tms_utime, tms_stime, tms_cutime, и tms_cstime для дочернего процесса устанавливаются в нуль.
- Число секунд, оставшихся до момента, когда сигнал SIGALRM будет сгенерирован, установлен для дочернего процесса в нуль.
- Набор отложенных сигналов для дочернего процесса пуст.
- Набор блокировок файлов, установленных родителем, не наследуется.
- Таймеры процесса, созданные родителем, не наследуются.
- Блокировки и распределение памяти родителем не наследуются.

– Если дочерний процесс порождён на удалённом узле, то ID группы родительского процесса, и его принадлежность сеансу не наследуются; для дочернего процесса формируется новый сеанс и новая группа процессов.

Дочерний процесс имеет доступ к окружению родительского процесса, используя глобальную переменную окружения (находится в <unistd.h>).

Если path содержит путь, принадлежащий файловой системе, смонтированной с установленным флагом ST_NOSUID, то эффективный ID пользователя и эффективный ID группы будут соответствовать ID пользователя и ID группы родительского процесса. Иначе, если есть соответствующие установки, эффективный ID пользователя дочернего процесса, устанавливается равным ID владельца пути. Точно так же, если есть соответствующие установки, эффективный ID группы дочернего процесса, устанавливается равным ID группы, являющейся владельцем пути.

Реальный ID пользователя, реальный ID группы и ID дополнительной группы дочернего процесса остаются такими, как у родительского процесса. Эффективный ID пользователя и эффективный ID группы дочернего процесса сохраняются такими, как ID пользователя и ID группы, заданные функцией setuid().

Важно отметить, что дочерний процесс, запущенный на удалённом узле, не наследует в сети PID родительского процесса. Следовательно функция getppid() для него не актуальна.

Связь между родительским процессом и дочерним не подразумевает, что дочерний процесс умирает, когда умирает родительский процесс.

Функция spawn() возвращает ID дочернего процесса, или -1 в случае ошибки. При этом в системной глобальной переменной errno, устанавливается код ошибки.

11.4. Локализация сервера

Для передачи сообщений клиентская нить должна создать соединение с каналом сервера, используя одну и ту же функцию ConnectAttach() как для сервера на местном, так и удалённом узле сети. Для этого ей необходимо знать pid процесса (сервера), nd узла компьютера в локальной сети (на котором запущен сервер), и идентификатор созданного сервером канала - chid. Когда клиентская и серверная нити находятся в одном процессе, получение этих данных для них не представляет труда: достаточно присвоить дескриптор созданного сервером канала глобальной переменной процесса, и он будет доступен в процессе всем нитям. При этом идентификатор локального узла nd=0, а идентификатор процесса - pid=getpid().

Сравнительно не сложно связать между собой родительский и дочерний процесс, находящиеся на одном узле локальной сети. В этом случае родительский процесс можно в начале рассматривать в качестве сервера по отношению к своему дочернему процессу – клиенту. Клиентская нить, при этом, сможет определить идентификатор сервера с помощью функции getppid(), идентификатор локального узла nd=0, а идентификатор созданного сервером канала (chid), предварительно преобразовав его целое значение в строковый вид с помощью функции itoa(), можно при запуске дочернего процесса передать как аргумент его нити main(). Полученное нитью main() в дочернем процессе значение дескриптора канала chid необходимо преобразовать из строкового представления в целое значение типа int с помощью функции atoi(). В итоге связи между дочерними процессами при формировании многопроцессного

приложения формируются посредством родительского процесса, запущенного как корневой процесс приложения.

Процедура установления соединения процессов-клиентов с каналами процессов-серверов усложняется, если процессы запускаются независимо друг от друга - *асинхронно*. В этом случае процедура локализации сервера клиентом и установление с ним связи становится для клиента отдельной не тривиальной задачей.

Одним из простых вариантов решения задачи локализации серверов для распределённых процессов является использование файла с таблицей параметров запущенных в сети процессов, который создаётся в файловой системе локальной сети на некотором известном всем процессам распределённого приложения узле и который используется всеми процессами для доступа к параметрам запущенных процессов и установления связей. Каждый запускаемый процесс получает уникальное в рамках приложения символическое имя, создаёт канал и регистрирует себя в таблице вместе с необходимыми для установления с ним соединения параметрами: символическое имя процесса, имя узла процесса в сети, дескриптор процесса, дескриптор канала. Это позволяет процессам, открыв файл с таблицей, находить друг друга в таблице по символическому имени и получать требуемые для установления соединений параметры. В локальном случае процессы могут использовать таблицу, созданную в именованной памяти.

Наиболее радикальное решение этой задачи заключается в использовании технологии программирования в ОС QNX процесса-сервера как «администратора ресурсов». Особенностью процесса, написанного как администратор ресурсов, является то, что при запуске он автоматически регистрируется в ОС QNX как объект файлового пространства, которому присваивается символическое имя (уникальное имя файла в пространстве имён путей ОС QNX). В этом случае нити процесса-клиента для создания соединения с каналом такого сервера-администратора ресурсов достаточно знать имя системного файла, зарегистрированного сервером в файловом пространстве, и использовать соответствующую стандартную библиотечную функцию open(), указав имя этого файла. В то же время следует заметить, что разработка администратора ресурсов в полном объёме является трудоёмкой задачей. Однако для решения задачи локализации есть простая альтернатива, основанная на частичной реализации технологии написания процесса-сервера как администратора ресурсов (так называемый «неполный администратор»).

На практике из рассмотренных выше механизмов локализации сервера и установления с ним соединения обычно используют "механизм родительского процесса" или "механизм неполного администратора".

11.4.1. Механизм родительского процесса

Суть механизма в том, что некий стартовый процесс отвечает за установление связей и выступает по отношению дочернему процессу как родительский процесс-сервер. Запускаемый дочерний процесс при организации связи с родителем выступает в роли дочернего процесса-клиента. Если сервер запускает клиента на удалённом узле, то для его запуска может быть использована только функция spawn().

Суть механизма заключается в том, что родительский процесс при запуске дочернего процесса на некотором узле может передать ему в качестве аргументов необходимые сведения для установления соединения с родительским процессом-сервером. Обмен информацией между родительским и дочерним процессом по установленному соединению является основой

для формирования структуры связей между запускаемыми процессами распределённого приложения. Ниже проиллюстрируем использование механизма родительского процесса для установления связи удалённого дочернего процесса с каналом родительского процесса.

Для определённости будем полагать, что имя узла, на котором стартует родительский процесс-сервер будет "CompSer", а дочерний процесс-клиент запускается на узле с именем "CompCle". Исполняемые модули обоих процессов находятся в файловой системе узла "CompSer" в каталоге /root и имеют имена соответственно "server" и "client". На узле "CompSer" на базе модуля "server" запускается стартовый процесс, который в свою очередь должен запустить модуль "client" на узле "CompCle". После этого процесс модуля "client" (обозначим его client) должен установить соединение с каналом процесса модуля "server" (обозначим его server).

Если решать эту задачу "как обычно", то возникает ряд проблем, связанных со спецификой запуска и наследования дочерним процессом client параметров сервера server на удалённом узле. После выполнения функции spawn() процесс client будет запущен на узле "CompCle". Но при попытке установить соединение с каналом родительского процесса, используя полученные параметры, возникает ошибка. Это вызвано уже рассмотренными выше особенностями работы процессов в сети:

1) Так как дочерний процесс client наследует от родительского процесса сетевой корень его файловой системы - /net/CompSer/, то будучи запущенным на узле "CompCle", тем не менее, «местным узлом» для дочернего процесса client остаётся узел "CompSer", и использованная клиентом для определения дескриптора узла сервера по имени "CompSer" функция netmgr_strtond() будет возвращать дочернему процессу client значение nd=0;

2) Функция getppid() при определении дочерним процессом client pid родительского процесса будет возвращать pid администратора сети на узле "CompCle", который опосредованно участвовал в запуске процесса client;

3) Функция spawn() возвращает родительскому процессу pid дочернего процесса client, а он уже будет принадлежать списку идентификаторов процессов на узле "CompCle".

Так как процесс client при запуске наследует корень файловой системы родительского процесса server, то "местным" узлом для процесса client будет считаться узел "CompSer". Поэтому при выполнении процессом client функции netmgr_strtond() для определения на своём узле "CompCle" дескриптора родительского узла по имени "CompSer" будет возвращаться значение nd=0, и очевидно, что попытка использовать этот дескриптор для установления дочерним процессом client соединения с каналом удалённого процесса server завершится ошибкой (или выполнится не корректно, случайно на узле "CompCle" окажется процесс с такими же pid и chid, что и у процесса server на узле "CompSer"). Таким образом, необходимо «всё расставить по своим местам».

Во-первых, так как на удалённом узле процесс client не может получить pid удалённого родительского процесса с помощью функции getppid(), то родительскому процессу придётся, например, при запуске дочернего процесса client передавать свой pid в качестве аргумента функции main().

Во-вторых, для того чтобы процесс client "прописался" в файловой системе узла "CompCle", необходимо изменить унаследованный от удалённого родительского процесса

server корень его файловой системы на корень файловой системы местного узла "CompCle". Для этого в дескрипторе процесса client требуется поменять значение сетевого корня файловой системы на /net/CompCle/. Это можно сделать, используя функцию chroot(). Например, перед выполнением функции spawn(), запускающей удалённый дочерний процесс client на узле "CompCle", можно было бы поменять предварительно для родительского процесса server корень сетевой файловой системы на /net/CompCle/, который и будет унаследован дочерним процессом client. Однако возникает другая проблема, заключающаяся в том, что вернуть обратно прежний корень процессу server невозможно. Процедура смены корня в сетевой файловой системе является необратимой! Это, в общем случае, нарушает в дальнейшем работу процесса server на узле "CompSer". Следовательно, необходимо действовать как-то иначе. Например, процесс server может запустить удалённый дочерний процесс client "руками" процесса-посредника - loader, передав ему необходимые параметры для запуска дочернего процесса функцией spawn(), и необходимые параметров для связи с каналом процесса server на узле "CompSer". В задачу loader будет входить и предварительная смена его корня файловой системы на узел /net/CompCle/ перед запуском дочернего процесса client на удалённом узле "CompCle". После этого процесса-посредника – loader, просто терминируется.

Процесс server может запустить процесс loader на локальном узле CompSer любой функцией семейства spawn*(). При этом процесс loader определяет pid своего родителя функцией getppid(), но должен получить от родительского процесса server следующие аргументы необходимые для запуска дочернего процесса client:

- путь к модулю client запускаемого дочернего процесса;
- имя узла "CompCle";
- chid созданного процессом server канала.

В итоге родительский процесс server сохраняет свой корень в сетевой файловой системе, а процесс client при запуске наследует корень на узле "CompCle" и в качестве параметра функции main() получает от процесса loader pid процесса serveri и chid его канала.

Исходный текст процесса server может выглядеть следующим образом:

```
/* Родительский процесс server */
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <unistd.h>
#include <spawn.h>

#define MSG_LEN 80
#define REPLY_LEN 80
#define NODE_NAME "CompCle"
#define PATH_LOADER "/root/loader"
#define PATH_CLIENT "/root/client"

int      chid;

void main(){
```

```
char    msg_to_receive[MSG_LEN];
char    replybuf[REPLY_LEN];
char    chid_ch[15];
int     rcvid;
…
chid=ChannelCreate(0);//создание канала
…
itoa(chid,chid_ch,10);//Преобразование chid в строку
spawnl(P_NOWAITO,PATH_LOADER, PATH_LOADER, PATH_CLIENT, NODE_NAME,
        chid_ch,NULL);//запуск процесса loader и передача ему параметров

//ждать сообщения от удалённого дочернего процесса
rcvid=MsgReceive(chid,&msg_to_receive, sizeof(msg_to_receive),NULL);
if (rcvid!=-1){
MsgReply(rcvid,EOK,&replybuf,sizeof(replybuf));
//связь установлена
…
}
```

При запуске процесса client процессом loader он передаёт ему в качестве аргументов:

- имя узла CompSer (его loader может получить с помощью функции confstr());
- pid процесса server (его loader может получить с помощью функции getppid());
- chid созданного сервером канала (его loader получает от server как аргумент).

Следует, однако, заметить, что, процесс loader, привязавшись, перед выполнением функции spawn(), к корню файловой системы удалённого узла, доступ к объектам файловой системы местного узла необходимо осуществлять, уже используя сетевые имена, явно указывая в пути в качестве префикса имя местного узла:

/net/<имя узла>/<полное_имя_файла>.

Исходный текст модуля loader может выглядеть следующим образом:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <unistd.h>
#include <spawn.h>

#define BUFF_SIZE 80
…
int main(int argc,char **argv){

spawn_inheritance_type      inherit;
char    *args[5]={NULL,NULL,NULL,NULL,NULL};//массив параметров
char    *envp[1]={NULL};//массив переменных среды
```

```
char    buff[BUFF_SIZE];//имя узла сервера
char    path_client[80];//путь к модулю дочернего процесса
char    pid_str[15];//строка с PID сервера
char    path_root[80];//путь к корню


confstr(_CS_HOSTNAME,buff,BUFF_SIZE);/*получить в buff имя узла сервера*/

/*Формирование пути к модулю client на узле "CompSer" как на удалённом узле*/
strcpy(path_client,"/net/");//корень в сети
strcat(path_client,buff);//добавить имя узла сервера
strcat(path_client,"/"); //добавить слеш
strcat(path_client,argv[1]);/*добавить путь к модулю запуска дочернего процесса*/
args[0]=path_client;//стандарт для значения args[0]

args[1]=buff;// сетевое имя узла родительского процесса

itoa(getppid(),pid_str,10);/*преобразовать в строку PID процесса server*/
args[2]=pid_str;//строка с PID процесса server

args[3]=argv[3];//chid канала родительского процесса

/*Формирование пути для изменения корня ФС на узел клиента*/
strcpy(path_root,"/net/");
strcat(path_root,argv[2]); //добавить имя узла клиента
strcat(path_root,"/");

inherit.nd=netmgr_strtond(argv[2],NULL);/*получить nd узла клиента*/
inherit.flags=SPAWN_SETND;//флаг запуска удалённого процесса

/*!!!!!!!!!!! Изменение корня ФС и запуск клиента !!!!!!!!!!!!!!!*/
chroot(path_root);
if(spawn(path_client,0,NULL,&inherit,args,envp)!=-1) return(EXIT_SUCCESS);
else return(EXIT_FAILURE);
}
```

Модуль client должен принять аргументы и выполнить соединение с каналом сервера:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>
…
int     coid;
```

```
…
int       main(int argc, char **argv){//приём параметров сервера
…
coid=ConnectAttach(netmgr_strtond(argv[1],NULL),atoi(argv[2]), atoi(argv[3]),0,0);
…
}
```

Замечание. Переключение корня файловой системы на удалённый узел может выполнить и процесс client, используя функцию chroot(), сразу после его загрузки процессом server на удалённый узел. В этом случае процесс loader не нужен, но при запуске процесса client процесс server в качестве аргумента должен передать клиенту ещё и сетевое имя узла сервера для получения дескриптора его узла.

11.4.2. Механизм именованных каналов

Механизм именованных каналов основан на идеи регистрации создаваемого процессом-сервером именованного канала в файловой системе QNX в качестве особого типа файла с заданным символическим именем. При успешном создании сервером именованного канала соответствующий ему файл появляется в ФС, доступ к нему автоматически открывается, а дескриптор возвращается серверу. Появление в файловой системе файлов именованных каналов даёт возможность процессу-клиенту не локализовать процесс-сервер, а устанавливать соединение с именованным каналом сервера, указывая только имя канала, и не заботясь об указании дескрипторов сервера и даже узла локальной сети.

Для реализации механизма именованных каналов процесс-сервер должен разрабатываться как специальный системный процесс - *системный администратор*, с которым операционная система связывает *системные ресурсы*, придающие ему специальные свойства, позволяющие процессу выполнять или обслуживать следующие системные вызовов [10][12][15]:

name_attach() - создание именованного канала
name_detach() - удаление именованного канала
name_open()   - открыть доступ (соединение) к именованному каналу
name_close()   - закрыть доступ (соединение) к именованному каналу

11.4.2.1. Создание именованного канала

Создание именованного канала сервер осуществляет с помощью вызова name_attach(), при выполнении которого именованный канал создаётся и регистрируется как объект файловой системы, а сервер приобретает свойства системного администратора:

```
name_attach_t* name_attach ( dispatch_t *dpp, // dispatch-драйвер
                             const char *path, // имя в ФС
                             unsigned flags ); // флаги
```

Для системного администратора необходимо указать так называемый dispatch-драйвер. Для создания именованного канала достаточно использовать системный dispatch-драйвер по умолчанию. Для этого аргументу dpp следует установить значение NULL. В этом случае системные ресурсы, необходимые серверу как процессу-администратору, строятся ядром ОС автоматически. Аргумент path задаёт путь регистрации именованного канала в пространстве

имён файловой системы (ФС) QNX. Путь регистрации имени канала не должна начинаться с символа точки или символа слеш. Имя именованного канала - path, регистрируется в одном из каталогов ФС - /dev/name/local или /dev/name/global. Соответственно, полное имя именованного канала в ФС будет - /dev/name/[local|global]/path. Внутри строки path символ слеш может использоваться, если процесс-сервер создаёт в каталоге ФС /dev/name/[local|global] иерархию именованных каналов, принадлежащих процессу.

Аргумент flags устанавливает свойство видимости имени канала в локальной сети. Если flags равен 0, имя канала регистрируется в ФС как локальное, видимое в рамках текущего узла локальной сети. Если flags присвоить значение системной константы NAME_FLAG_ATTACH_GLOBAL, то имя канала регистрируется в ФС как глобальное сетевое имя, видимое в пределах всей локальной сети. В итоге имена каналов, созданные процессом как локальные, помещаются в ФС узла в каталог /dev/name/local, а глобальные - /dev/name/global.

Вызов создания сервером именованного канала возвращает указатель на структуру системного типа name_attach_t, которая включает в себя следующие поля:

```
typedef struct _name_attach{dispatch_t* dpp; // dispatch-драйвер
                            int chid; // идентификатор канала
                            int mntid; // идентификатор монтирования
                            int zero[2]; // нулевые значения
                            }name_attach_t;
```

В результате канал неявно создаётся посредством внутреннего вызова ядром ОС ChannelCreate(), а полученное ID канала заносится ядром в поле chid структуры _name_attach, откуда сервер и получает нужный ему ID канала. Остальные поля для сервера не представляют интереса.

Важно отметить, что канал создаётся ядром ОС вызовом ChannelCreate() с установленными флагами посылки ядром по этому каналу серверу уведомлений при возникновении следующих связанных с каналом событий:

_NTO_CHF_UNBLOCK - преждевременный выход клиента из блокированного состояния;

_NTO_CHF_COID_DISCONNECT - некоторым из клиентов, связанных с каналом, выполнен разрыв соединения с каналом;

_NTO_CHF_DISCONNECT - все ранее установленные клиентами соединения с каналом разорваны.

Указанный набор флагов предписывает ядру при преждевременной деблокировке клиентской нити, не получившей от сервера ответа, или при разрыве установленных с каналом соединений уведомлять процесс-сервер об этих событиях, отправляя ему в этот канал соответствующий уведомляющий импульс (системное сообщение типа _pulse):

```
struct _pulse {uint16_t      type;
               uint16_t      subtype;
               int8_t        code;
               uint8_t       zero[3];
               union sigval  value;
               int32_t       scoid;
};
```

Следовательно, при программировании сервера, ожидающего сообщения по именованному каналу, необходимо контролировать возможность возникновения таких событий и поступления в именованный канал посылаемых ядром соответствующих уведомлений.

Значения полей code и value в структуре пришедшего импульса однозначно соответствуют причине уведомления - типу события:

Преждевременный выход клиентской нити из блокированного состояния (например, по таймауту или сигналу) поле code будет равно системной константе _PULSE_CODE_UNBLOCK, а поле value – значению, возвращённому серверу функцией MsgReceive() ссылки на клиента – rcvid, от которого было принято сообщение.

Разрыв клиентом некоторого из ранее установленных с именованным каналом соединений. Импульс, уведомляющий об этом событии, в поле code будет иметь значение системной константы _PULSE_CODE_COIDDEATH, а в поле value – ID соединения (coid). Если клиент терминируется, предварительно не разорвав соединения с каналом, соединение разорвёт ядро и пошлёт уведомление.

Все соединения с каналом разорваны. Импульс, посылаемый ядром уведомляющий об этом событии, в поле code будет иметь значение системной константы _PULSE_CODE_DISCONNECT, а в поле value – None. В этом случае ядро возлагает на сервер обязанность аннулировать созданное ядром соединение с именованным каналом сервера для посылки уведомляющих импульсов, выполнив вызов ConnectDetach(scoid), где scoid – системный ID соединения ядра с каналом сервера, взятый из поля scoid принятого импульса (обычно, когда флажок _NTO_CHF_DISCONNECT не установлен, ядро по умолчанию автоматически удаляет системное соединение с каналом сервера).

Замечание. Ядро поддерживает взаимно-однозначное соответствие между именем именованного канала, зарегистрированного в ФС, и его chid в процессе-сервере. Поэтому нельзя создавать копии процессов, создающих одноимённые каналы с одинаковыми именами, так как возникает конфликт имён в файловой системе.

11.4.2.2. Соединение с именованным каналом

Локализация клиентом сервера, создавшего именованные каналы, значительно упрощается. Клиенту достаточно только знать имя файла (путь в ФС), ассоциированного с именованным каналом. Для установления клиентом соединения с именованным каналом сервера используется функция:

```
int name_open( const char* path, // относительное имя канала
               int flag ); // флаг области видимости имени
```

Аргумент name задаёт имя канала сервера в каталоге, с которым клиентская нить устанавливает соединение. Аргумент flags определяет указанное имя канала как *локальное* или *глобальное*. Если flags равен 0, то имя канала процесс-клиент считает локальным - *уникальное* имя в пределах ФС узла. Если flags присваивается значение системной константы NAME_FLAG_ATTACH_GLOBAL, то имя канала процесс-клиент считает глобальным - *уникальное* имя канала в пределах локальной сети.

При удачном выполнении функции name_open() возвращается идентификатор соединения - coid, а при неудачном - значение –1. Важно отметить следующее. При удачном

выполнении клиентом функции name_open() ядро уведомляет об этом сервера, посылая ему не импульс, а так называемое служебное IO-сообщение (с помощью вызова MsgSend()) системного типа _IO_CONNECT. Сообщение типа _IO_CONNECT имеет следующую структуру:

```
struct _io_connect {
        uint16_t        type;
        uint16_t        subtype;
        uint32_t        file_type;
        uint16_t        reply_max;
        uint16_t        entry_max;
        uint32_t        key;
        uint32_t        handle;
        uint32_t        ioflag;
        uint32_t        mode;
        uint16_t        sflag;
        uint16_t        access;
        uint16_t        zero;
        uint16_t        path_len;
        uint8_t         eflag;
        uint8_t         extra_type;
        uint16_t        extra_len;
        char            path[1];
};
```

IO-сообщение типа _IO_CONNECT используется ядром в различных целях с различным содержанием его полей. Важно учесть, что в случае с именованным каналом поле type будет содержать значение именованной системной константы _IO_CONNECT, а поле subtype будет содержать значение именованной системной константы _IO_CONNECT_OPEN. Ядро тем самым оповещает сервер, что с каналом установлено соединение. Сервер должен быть готовым принять по именованному каналу такое сообщение и обязательно вызовом MsgReply() послать ядру ответ EOK, чтобы соединение завершилось успешно.

Ядро, в общем случае, кроме уже рассмотренных системных сообщений, может посылать в именованный канал сервера–администратора и другие системные IO-сообщения, используемые ядром при взаимодействии с любыми системными администраторами в различных целях. Серверу, работающему с именованным каналом, необходимости в их анализе нет, но возможность их прихода необходимо контролировать при создании сервера-администратора именованных каналов. В заголовке таких `IO`-сообщений значение поля type будет находиться в диапазоне _IO_BASE < type ≤ _IO_MAX. Получение таких IO-сообщений по именованному каналу сервер именованного канала должен рассматривать их как случайные - «ошибочно отправленные ядром», и послать ядру ответ, используя вызов MsgError(), содержащий системное символическое значение ENOSYS.

Из сказанного выше следует, что сервер по созданному именованному каналу должен ожидать прихода:

– ожидаемых сообщений от клиентов,

- системных импульсов-уведомлений от ядра,
- системное IO-сообщение типа _IO_CONNECT,
- «случайные» IO-сообщения других типов.

В связи с этим сервер обязан контролировать тип принимаемых по именованному каналу сообщений и корректно на них реагировать:

- При приёме уведомления-импульса серверу нет необходимости отправлять ответ.
- При приёме сообщения типа _IO_CONNECT сервер должен ответить EOK
- При приёме прочих IO-сообщений - должен ответить ENOSYS, используя вызов MsgError().

Так как обычные сообщения от клиентов будут поступать в именованный канал вместе с системными сообщениями типа _pulse и _IO_CONNECT, то для удобства распознавания сервером клиентских и системных сообщений процессам-клиентам целесообразно наделять свои сообщения, посылаемые в именованный канал, четырёхбайтным заголовком, аналогичным заголовку системных сообщений типа _pulse и _IO_CONNECT:

```
uint16_t    type;
uint16_t    subtype;
```

где полям type и subtype следует присвоить *нулевые* значения.

Приведём пример программного модуля, иллюстрирующий создание сервером именованного канала и установление клиентом с ним соединения. Программный модуль может запускаться либо в режиме сервера, либо - клиента. В режиме сервера процесс запускается с аргументом <–**s**>, в режиме клиента - с аргументом <–**c**>.

Пример:

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>

#define NAME_CHAN"name_chan"

typedef struct _pulse msg_header_t; //абстрактный тип для заголовка сообщения как у
импульса
typedef struct _my_data {
msg_header_t hdr;
int data;
} my_data_t; // абстрактный тип для сообщений клиента

/*** Функция регистрации сервером именованного канала ***/
int server(){
```

```
name_attach_t *attach;
my_data_t msg;
int rcvid;

/* Создание именованного канала с именем "name_chan" */

if ((attach = name_attach(NULL, NAME_CHAN, 0)) == NULL) {
return EXIT_FAILURE;
}

  /* Цикл ожидания поступления сообщений в именованный канал */
while (1){
        rcvid = MsgReceive(attach->chid,&msg,sizeof(msg),NULL);
        if (rcvid == -1) {/* Ошибка, завершение */
                break;
        }

if (rcvid == 0) {/* Получен импульс */
switch (msg.hdr.code) {
case _PULSE_CODE_DISCONNECT:
/* Клиент разорвал все ранее созданные связи (вызвал name_close() для каждого вызова
name_open() с именем канала) или терминировался */
ConnectDetach(msg.hdr.scoid);
/* Уничтожить соединение ядра с каналом сервера*/
name_detach(attach, 0);
/* Удалить имя процесса */
printf("Server receive _PULSE_CODE_DISCONNECT and terminated");
return EXIT_SUCCESS;
case _PULSE_CODE_UNBLOCK:
/* Клиент хочет деблокироваться (получен сигнал или истёк таймаут). Серверу необходимо
принять решение о посылке ответа */
break;
default:
   /* Пришёл импульс от какого-то процесса или от ядра
   (PULSE_CODE_COIDDEATH или _PULSE_CODE_THREADDEATH) */
   break;
}
continue;// Завершить текущую итерацию цикла
}
   /* Полученное сообщение не импульс */
```

```
    if (msg.hdr.type == _IO_CONNECT ) {/* Получено сообщение типа _IO_CONNECT - клиент
    выполнил name_open(),нужен ответ EOK */
    MsgReply( rcvid, EOK, NULL, 0 );
    continue; // Завершить текущую итерацию цикла
}


if (msg.hdr.type > _IO_BASE && msg.hdr.type <= _IO_MAX ){ /*Получено IO-сообщение от ядра,
аннулировать его */
MsgError( rcvid, ENOSYS );
continue; // Завершить текущую итерацию цикла
}

/* Получено сообщение от клиента */
printf("Server receive %d \n", msg.data);
MsgReply(rcvid, EOK, 0, 0);
}
}

/*** Функция клиента ***/
int client() {
   my_data_t msg;
   int server_coid;

   if ((server_coid = name_open(NAME_CHAN, 0)) == -1) {
      return EXIT_FAILURE;
   }

   /* Заголовок сообщения клиента */
   msg.hdr.type = 0x00;
   msg.hdr.subtype = 0x00;

   /* Послать 5-ть сообщений серверу */
   for (msg.data=0; msg.data < 5; msg.data++) {
      printf("Client sending %d \n", msg.data);
      if (MsgSend(server_coid, &msg, sizeof(msg), NULL, 0) == -1) {
         break;
      }
   }

   /* Закрыть соединение с сервером */
```

```
    name_close(server_coid);
    return EXIT_SUCCESS;
}

int main(int argc, char **argv) {
    int ret;

    if (argc < 2) {
        printf("Usage %s -s | -c \n", argv[0]);
        ret = EXIT_FAILURE;
    }
    else
                if (strcmp(argv[1], "-c") == 0) {
                        printf("Running Client ... \n");
                        ret = client();   /* Запуск как клиента */
                }
                else
                        if (strcmp(argv[1], "-s") == 0) {
                                printf("Running Server ... \n");
                                ret = server();   /* Запуск как сервера */
                        }
                        else {
                                printf("Usage %s -s | -c \n", argv[0]);
                                ret = EXIT_FAILURE;
                        }
    return ret;
}
```

Для удаления клиентом ранее созданного соединения с именованным каналом используется функция:

int name_close(int coid);

Здесь аргумент coid - ID соединения с именованным каналом сервера (значение coid возвращает функция name_open()).

11.4.3. Использование именованных каналов в сети

Сервис глобальных имён в локальной QNET-сети стал возможным, начиная с QNX версии 6.3. Этот сервис обеспечивается так называемым GNS-менеджером (утилита gns), которая должна быть загружена в узлах локальной сети, в которых процессы будут создавать или открывать глобальные именованные каналы. Используя этот сервис, процесс-сервер может создавать глобальные именованные каналы, а процессы-клиенты, используя глобальное имя канала, могут присоединяться к ним как локально, так и через QNET-сеть. Процессы могут

воспользоваться сервисом глобальных имён, только если они имеют привилегии администратора системы (root).

Чтобы развернуть в QNET-сети сервис глобальных имён, необходимо на некотором узле (хосте) загрузить GNS-менеджер в режиме сервера (gns-сервер):

```
# gns -s [nodename ...]
```

На остальных узлах GNS-менеджер должен быть загружен в режиме клиента (gns-клиент):

```
# gns -c [nodename ...]
```

Менеджер в режиме сервера осуществляет в сети управление централизованной базой данных, в которой хранится информация о созданных процессами глобальных именованных каналах. Он обрабатывает запросы от локальных и удалённых процессов на создание или открытие именованных каналов.

При запуске gns-сервера параметр nodename задаёт имена хостов (ведущих узлов), на которых запускается gns-сервер. При запуске gns-клиента параметр nodename задаёт имена хостов, с которыми gns-клиент будет взаимодействовать. Если параметр nodename явно не задан, то выбор gns-клиентом хоста будет осуществляться автоматически.

GNS-менеджер в режиме клиента играет роль посредника между удалённым процессом и gns-сервером, обеспечивая удалённым процессам выполнение запросов создания или открытия глобальных именованных каналов. На хосте запросы от локальных процессов обслуживаются gns-сервером напрямую.

На разных узлах сети процессы могут создавать глобальные каналы с одинаковыми именами. При этом содержание каталога /dev/name/global на всех узлах, на которых запущен gns-клиент или gns-сервер, будет одинаковым. Это позволяет дублировать именованные каналы в сети.

Для открытия процессами именованного канала используется запрос name_open(). Если одно имя зарегистрировано в сети на нескольких узлах, то для поиска и подключения к конкретному каналу применяются следующие правила.

Если существует канал, зарегистрированный на том же узле, что и выполняющий запрос процесс (локальный канал), gns-менеджер сначала выполняет попытку подключения к локальному каналу, при успешном подключении процесс открывает и использует локальный именованный канал.

При отсутствии локально зарегистрированного именованного канала, либо если по некоторым причинам получен отказ от службы именованных каналов локально обеспечить запрос, gns-менеджер выполняет попытку поиска и подключения к одноимённому глобальному каналу, зарегистрированному в сети на удалённом узле. При наличии в сети нескольких узлов с зарегистрированными одноимёнными глобальными каналами порядок попыток подключения к ним (т.е. последовательность подключения) не определён.

Работа с несколькими серверами GNS. В сети на разных узлах можно запустить несколько gns-менеджеров службы глобальных имён в режиме сервера.

Работа с несколькими доменами служб. На разных узлах сети можно установить несколько gns-клиентов для взаимодействия с разными хостами. Например, для взаимодействия клиента с хостом server1 он запускается на узле командой:

```
# gns -c server1
```

А на другом узле для взаимодействия с хостом `server2` клиент запускается командой:

```
# gns -c server2
```

При этом создаются отдельные "домены служб". Клиенты, подключённые к хосту server1 (в "домене служб 1") не могут использовать службы, зарегистрированные на сервере server2 (в "домене служб 2").

Резервные серверы GNS. Поскольку сервер GNS является централизованной базой данных, отсутствие резервирования при выходе сервера GNS из строя означает прерывание работы службы именованных каналов в сети. Для повышения надёжности можно запустить несколько серверов GNS и направить клиенты на все эти серверы.

Например, на хостах с именами server1 и server2 с целью резервирования службы именованных каналов можно загрузить серверы GNS, выполнив команду:

```
# gns -s
```

На остальных узлах можно запустить клиентов GNS, выполнив команду с перечислением имён хостов с запущенными серверами GNS:

```
# gns -c server1 server2
```

В этом случае при каждой попытке регистрации неким процессом глобального именованного канала запрос регистрации отправляется одновременно на хост server1 и server2. При каждой попытке неким процессом открыть глобальный именованный канал запрос направляется одновременно и на хост server1 и на хост server2.

Примечание. GNS на хосте server1 не взаимодействует с GNS на хосте server2. Это означает, что GNS на хосте server1 не может перенаправить запрос клиента на хост server2, поскольку менеджер GNS не функционирует как клиент и сервер одновременно. Поэтому запросы направляются одновременно на оба хоста.

Автоматический поиск (сканирование) клиентами хостов. Каждый менеджер GNS зарегистрирован по пути /dev/name/gns_server или /dev/name/gns_client. При запуске клиента GNS на узле без указания целевого хоста выполняется автоматический поиск сервера(-ов) GNS в локальной сети. Автоматический поиск выполняется по каталогам локальной сети (обычно /net) для поиска путевого имени /net/компьютер/proc/mount/dev/name/gns_server.

Примечание. Автоматический поиск не является гарантией обнаружения хоста, т.к. в сети Qnet в каталоге /net могут находиться не все локальные компьютеры.

Автоматический поиск клиентов целесообразен при потере соединения с хостом. В этом случае для поиска хоста клиентом выполняется повторный поиск. Это упрощает запуск другого сервера GNS и его синхронизацию с первым сервером (синхронизация рассматривается ниже), а также последующее уничтожение первого сервера GNS.

Если для клиента в командной строке указан конкретный сервер (конкретные серверы) GNS, то автоматическое сканирование не выполняется. При потере соединения с сервером он пытается восстановить соединение при каждом запросе на регистрацию, поиск или подключение.

Режим резервного сервера. Менеджер GNS можно запустить в режиме "резервный сервер". Для этого следует запустить менеджер GNS в режиме сервера и ввести в командной

строке имя конкретного резервируемого хоста. Например, на хосте node1 сервер запускается стандартно:

```
# gns -s
```

а на хосте node2 – как сервер резервный для хоста node1:

```
# gns -s node1
```

Менеджер GNS на хосте node2 выполняет синхронизацию с менеджером GNS на хосте node1, получает всю информацию о глобальных именованных каналов из хоста node1 и сохраняет её локально.

Все клиенты GNS, уже подключённые к серверу GNS в хосте node1, получают информацию о новом сервере на хосте node2 и подключаются к нему. При этом для клиентов GNS теперь существует несколько хостов. Это аналогично выполнению запуска клиента GNS следующим образом:

```
# gns -c node1 node2
```

Замечание. Более детальную информацию об использовании именованных каналов в сети можно получить в справочной системе ОСРВ QNX [12].

ЧАСТЬ 3. ПРОГРАММИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

12. Параллельно выполняемые вычислительные процедуры

Организация параллельного выполнения вычислительных процедур внутри процессов осуществляется посредством запуска параллельно выполняемых функций – *нитей*. При создании процесса в нем автоматически в качестве особой главной нити запускается функция main(). При необходимости, нить main() может создавать другие нити для параллельного с ней выполнения. Любая вновь созданная нить может в том же самом процессе создать новую нить. Нить, создавшая новую нить, считается родительской нитью, а созданная ею нить – *дочерней нитью*. Никаких ограничений на количество параллельно выполняемых в процессе нитей (кроме вычислительных ресурсов среды исполнения и системных ресурсов ядра ОСРВ) не накладывается.

Запуск функции в качестве нити отличается от обычного вызова функции тем, что требует предварительной спецификации её свойств как параллельно выполняемой вычислительной процедуры, поэтому осуществляется посредством специального системного запроса. Программный интерфейс ОСРВ предоставляет необходимые средства подготовки и запуска нитей в процессах. Так как всем нитям одновременно доступны все внутренние ресурсы процесса, то программный интерфейс ОСРВ предоставляет системные средства синхронизации выполнения параллельных вычислений (блокирование/деблокирование выполнения) для управления их доступом к разделяемым ресурсам [10][12][15].

12.1. Формирование свойств и запуск нити

12.1.1. Прототип функции и атрибуты нити

Функция, используемая для запуска нитей, должна иметь следующий универсальный прототип:

```
void* thread_routine(void*);
```

Заметим, что в качестве аргумента функция получает указатель неопределённого типа, что позволяет передавать в такую функцию любой предварительно подготовленный набор параметров любого типа (например, в виде полей структуры). Результат выполнения функции так же возвращается как указатель неопределённого типа, который при получении следует семантически привести к требуемому определённому в программе типу.

При создании нити она наделяется (в частности, по умолчанию) требуемыми системными свойствами (атрибутами). В качестве таких атрибутов выступают:

- свойство нити быть обособленной или присоединяемой;
- параметры стека нити;
- параметры диспетчеризации нити для разделения процессора.

Свойства запускаемой нити формируются посредством так называемой атрибутной запись нити, которая определяется как переменная системного типа pthread_attr_t. Перед формированием атрибутной записи она должна быть проинициализирована с помощью функции:

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

где attr - инициализируемая атрибутная запись, используемая затем в функциях, предназначенных для задания атрибутов, определяющих соответствующие свойства нити. Функция формирует атрибутную запись для запуска нити со свойствами по умолчанию.

12.1.2. Присоединяемая или обособленная нить

Если нить создаётся как *присоединяемая*, то у других нитей появляется возможность "присоединяться" к ней, а в результате блокироваться и ждать её завершения. Если нить создаётся как *обособленная*, то такой возможности по отношению к ней у других нитей не будет. Для задания этого свойства в атрибутной записи используется функция:

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Задание свойства нити быть присоединяемой или нет осуществляется с помощью аргумента detachstate. Если нить должна быть *присоединяемой* (свойство по умолчанию), то аргументу присваивается значение символической системной константы PTHREAD_CREATE_JOINABLE. Если нить должна быть *обособленной*, то аргументу присваивается значение PTHREAD_CREATE_DETACHED. При успешном завершении функция возвращает значение символической системной константы EOK.

12.1.3. Параметры стека нити

По умолчанию ядро QNX выделяет запускаемой нити стек минимального размера - PTHREAD_STACK_MIN (4 KB), достаточный для функции нити, при выполнении которой стек практически не используется, например:

```
void *nothing_thread(void*){
   return;
}
```

В остальных случаях с учётом программных особенностей функции необходимо явно устанавливать достаточный размер стека. При явном определении характеристик стека нити должны быть заданы следующие параметры:

- адрес стека;
- размер стека;

Для задания адреса стека используется функция:

```
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t* attr, void* stackaddr);
```

Аргумент stackaddr задаёт адрес области памяти процесса, которая будет использоваться в качестве стека нити. Если в качестве stackaddr задать NULL, то стек формируется по умолчанию.

Чтобы задать размер явно формируемого стека используется функция:

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t* attr, size_t stacksize);
```

Аргумент stacksize задаёт размер стека в байтах. Реальный размер памяти, выделенный стеку, будет кратным размеру страницы памяти, величину которой можно узнать с помощью вызова:

sysconf(_SC_PAGESIZE).

При успешном завершении обе функции возвращают EOK.

Если нити выделяется системный стек по умолчанию (атрибут stackaddr равен NULL), то можно установить так называемую "*области защиты*" стека.

Область защиты – это область памяти, предназначенная для системного контроля переполнения стека, выделенного по умолчанию (атрибут stackaddr равен NULL). Она создаётся ядром непосредственно за стеком. Запись в эту область приводит к посылке ядром данной нити сигнала SIGEVG. Для этого в атрибутной записи необходимо задать параметр - размер "*области защиты*" выделенного системой стека. Для задания размера области защиты стека используется функция:

#include <pthread.h>
int pthread_attr_setguardsize(pthread_attr_t* attr, size_t guardsize);

При успешном завершении функция возвращает EOK. В результате нити будет установлена непосредственно за стеком область памяти защиты стека размером не менее guardsize байт. Если в качестве guardsize задать 0, то область защиты стека будет отсутствовать. Штатный размер области защиты стека можно узнать с помощью вызова:

sysconf(_SC_PAGESIZE).

Внимание! Если адрес стека *явно задан*, то вызов функции для установления области защиты игнорируется и область защиты *не выделяется*.

12.1.4. Приоритет и дисциплина диспетчеризации нити

Каждая созданная нить может находиться в системе в одном из следующих *состояний*:

- *Блокирована* – нить блокирована ядром QNX до возникновения контролируемых ядром условий её готовности к выполнению.
- *Готова* – нить готова к выполнению, и ждёт предоставления ей ядром QNX процессорного времени.
- *Активна* – ядром QNX нити предоставлено процессорное время и она в данный момент выполняется.

Нить в состоянии готовности постоянно конкурирует с другими готовыми и с активной нитью за переход в активное состояние. Конкуренция нитей осуществляется в соответствии с приоритетом, полученным при их создании. Приоритет для "непривилегированных" нитей является положительным целым числом в диапазоне от 1 до 63. Правило конкуренции заключается в том, что среди готовых нитей в состоянии активности может находиться нить, имеющая наибольший приоритет. Как только нить с более высоким, чем у текущей активной нити, становится готовой, активная нить *вытесняется* (принудительно переводится в состояние готовности), а более приоритетная нить становится активной. Если среди готовых к

выполнению нитей одинаково высокий приоритет имеют несколько нитей, то активной станет нить, которая раньше других оказалась в состоянии готовности.

Доля процессорного времени, выделяемого нити, ставшей активной, для выполнения, зависит от дисциплины диспетчеризации, назначенной ей при создании. В ОС QNX в качестве базовых дисциплин используются следующие дисциплины диспетчеризации:

FIFO (*First In First Out*) – нить, ставшая активной, выполняется до тех пор, пока не завершит свою работу, не будет вытеснена более приоритетной нитью или не перейдёт в блокированное состояние.

RR (*Round Robin*) – *карусельная* диспетчеризация, при которой продолжительность нахождения нити в активном состоянии ограничивается так называемым *квантом времени* выполнения (*time slice*), после истечения которого нить вытесняется и вновь поступает в очередь готовых к выполнению нитей, а первая готовая нить становится активной и выполняется в соответствии с её дисциплиной диспетчеризации.

Квант времени дисциплины RR является задаваемым системным параметром. Величину кванта времени - interval, выделяемого ядром процессу с дескриптором pid, можно узнать, воспользовавшись функцией:

```
#include <sched.h>
int sched_rr_get_interval(pid_t pid, struct timespec* interval);
```

Особенностью дисциплины FIFO является то, что теоретически активная нить может занимать процессор "вечно", если нет готовых нитей с большим приоритетом. При этом "деликатная нить" может сама периодически пытаться добровольно уступить процессор, используя функцию:

```
#include <sched.h>
int sched_yield(void);
```

Если окажется готовой другая нить с таким же приоритетом, то она станет активной. Если такой нити не окажется, то данная нить вновь станет активной и продолжит работу.

Созданная "дочерняя" нить по умолчанию наследует от родительской нити атрибуты диспетчеризации и приоритет. Однако, при необходимости эти параметры можно явно задать в атрибутной записи, выбрав значения отличные от наследуемых. Для этого следует отказаться от наследования, установив в атрибутной записи атрибут отмены наследования с помощью функции:

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t* attr, int inheritsched);
```

По умолчанию атрибут наследования в атрибутной записи имеет значение равное PTHREAD_INHERIT_SCHED - режим наследования. Чтобы изменить режим наследования по умолчанию и задать приоритет или дисциплину диспетчеризации явно, необходимо предварительно выполнить функцию со значения inheritsched, равным системной символической константе PTHREAD_EXPLICIT_SCHED – отказ от наследования.

12.1.4.1. Задание дисциплины диспетчеризации нити

Чтобы явно задать дисциплину диспетчеризации следует воспользоваться функцией:

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setschedpolicy(pthread_attr_t* attr, int policy);
```

Аргумент policy может принимать следующие значения:
SCHED_FIFO – для FIFO;
SCHED_RR – для RR;
SCHED_OTHER – зависит от версии ОС и может быть SCHED_RR.
SCHED_NOCHANGE – не изменять дисциплину диспетчеризации.

12.1.4.2. Задание приоритета нити

Для задания приоритета необходимо предварительно определить переменную системного типа struct sched_param, в которой значение приоритета присваивается полю с именем sched_priority. Например:

```
struct sched_param param;
…
param.sched_priority = 15;//Значение приоритета
…
```

Затем следует выполнить функцию:

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
```

Заметим, что для установки приоритета можно обойтись и без функции pthread_attr_setschedparam(). Для этого достаточно выполнить присвоение приоритета, непосредственно соответствующему полю в атрибутной записи, например:

```
attr.param.sched_priority = 15;//Значение приоритета
```

При успешном выполнении все рассмотренные выше функции возвращают EOK.

12.1.5. Создание и запуск нити

Для создания и запуска нити с подготовленными атрибутами используется функция:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

Функция создаёт и запускает как нить функцию, указанную в аргументе start_routine, с атрибутами, заданными в атрибутной записи, указанной в аргументе attr. Если предполагается использовать атрибуты нити по умолчанию, то attr можно положить равным NULL. При запуске нити ей в качестве аргумента функции start_routine передаётся указатель arg. Если функция без аргументов, то аргумент arg делают равным NULL. Аргумент thread указывает переменную, в которой сохраняется дескриптор (ID) созданной нити. Если необходимости в дескрипторе нити нет, можно задать NULL. При необходимости запущенная нить может получить свой дескриптор с помощью функции:

```
#include <pthread.h>
pthread_t pthread_self( void );
```

Пример:

```
/* Создание обособленной нити с дисциплиной RR и приоритетом 15*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <iostream>
#include <unistd.h>
void* func_thread (void* arg){ //Определение функции для запуска нити
        printf("Запущена нить с ID: %d\n", pthread_self());
        pthread_t tid=pthread_self();
                sleep(5);
        std::cout << "Завершение нити с ID: " << pthread_self() << std::endl;
        pthread_exit(EXIT_SUCCESS);//return( 0 );//возврат статуса успешного завершения нити
}

int main(void){
        pthread_attr_t attr;//Атрибутная запись нити
        struct sched_param prio;//Приоритет нити
        int rval;//Возвращаемый статус завершения нити

        pthread_attr_init(&attr);//Инициализация атрибутной записи
        /*Задать свойство обособленности*/
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
        /*Отказаться от наследования свойств диспетчеризации*/
        pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);
        /*Задать карусельную дисциплину диспетчеризации */
        pthread_attr_setschedpolicy(&attr,SCHED_RR);

        /*Задать приоритет 15*/
        prio.sched_priority = 15;
        pthread_attr_setschedparam(&attr,&prio);

/*Запустить нить*/
        pthread_create(NULL, &attr, &func_thread, NULL);
        rval = pthread_join (thread_id, NULL);
        if (rval == EXIT_SUCCESS){
```

```
            std::cout << "Успешное завершение нити: ID = " << pthread_self() << std::endl;
        }
        else{
            std::cout << "Не удачное завершения нити: ID = " << pthread_self() << std::endl;
        }
return rval;
}
```

12.2. Проблема инверсии приоритетов

Инверсия приоритетов – это фундаментальная проблема приложений реального времени [9][10]. Инверсия приоритетов выражается в том, что вре́менное преимущество выполнения получает нить с низким приоритетом, в то время как готовые к выполнению нити с более высоким приоритетом оказываются заблокированными. Причины, способствующие инверсии приоритетов, разнообразны, рассмотрим одну из них. Пусть в некотором приложении среди прочих имеются три процесса P1, P2, P3, в которых запущены и выполняются соответственно нити *l*(5), *h*(12), *s*(20). В скобках указаны приоритеты нитей. Положим, что процесс P3 играет роль сервера, принимающего на обработку сообщения, а процессы P1 и P2 являются его клиентами. Пусть нить *s* в некоторый момент времени переходит в RECEIVE-блокированное состояние, для ожидания сообщений от P1 или P2. Пусть нити *l*(5) и *h*(12) оказались готовыми к выполнению и нить *h*(12) стала активна, так как имеет больший приоритет. Допустим, что нить *h*(12) в некоторый момент времени переходит в блокированное состояние для ожидания некоторого события (например, освобождения мутекса или уведомления о наступлении запланированного ею момента времени). Так как нить *s*(20) находится в RECEIVE-блокированном состоянии, то нить *l*(5) становится активной, посылает сообщение серверу P3 и сразу переходит в REPLY-блокированное состояние. Одновременно нить *s*(20) выходит из RECEIVE-блокированного состояния и становится активной. Она принимает сообщение от нити *l*(5) и начинает его обслуживать. Положим, что обслуживание сообщения потребовало значительного времени выполнения нити *s*(20), в течение которого возникло ожидаемое нитью *h*(12) событие. Нить *h*(12) переходит в состояние готовности к выполнению, но не может стать активной, так как в данный момент активной является более приоритетная нить *s*(20), занятая обслуживанием сообщения от нити *l*(5), приоритет которой ниже, чем у нити *h*(12). Это означает, что нить *l*(5) получила преимущество и задерживает готовую к выполнению нить *h*(12) с более высоким приоритетом, временно воспользовавшись приоритетом нити *s*(20), обсуживающей её сообщение. Сложившееся состояние и называется инверсией приоритетов.

Попробуем предвосхитить возможность возникновения подобного состояния и предложить механизм восстановления приоритетного использования процессора. Так как причиной инверсии приоритетов стал высокий приоритет сервера, то таким механизмом может быть механизм *наследования сервером приоритета клиентской нити*. Если применить этот механизм к рассмотренному выше примеру, то очевидно, что в этом случае при обработке сообщения от нити *l*(5) приоритет нити *s*(20) станет равным 5 и окажется ниже приоритета нити *h*(12), когда та станет готовой к выполнению, нить *h*(12) вытеснит нить *s*(5) и станет активной. Однако, как только самой нити *h*(12) потребуется услуга сервера P3, она пошлёт ему сообщение и перейдёт в SEND-блокированное состояние, нить *s*(5) вновь станет активной и продолжит

обработку сообщения от нити *l*(5). Более того, готовые нити других процессов рассматриваемого приложения, имеющие приоритеты выше, чем у нити *l*(5), но меньше, чем у нити *h*(12), будут вытеснять нить *s*(5) и тем самым способствовать задержке нити *h*(12). Это ещё одна форма проявления инверсии приоритетов.

Чтобы компенсировать нити *h*(12) издержки инверсии приоритетов можно предложить динамически изменять приоритет нити *s*(*), делая его равным наибольшему из приоритетов всех заблокированных клиентов на сервере Р3. Это по крайней мере ускорит завершение обработки нитью *s*(*) сообщения нити *l*(5) (не будут "мешать" другие нити) и, следовательно, ускорит переход нити *s*(*) к обработке сообщения нити *h*(12).

Механизм наследования серверной нитью приоритета клиентской нити реализован в QNX по умолчанию как наиболее благоприятный. Однако заметим, что этот механизм наследования не предполагает дальнейшего последовательного наследования полученного серверной нитью приоритета. Это значит, что если сервер будет причиной блокировки некоторого высокоприоритетного клиента, а сам в свою очередь в процессе обслуживания сообщения очередного клиента сам окажется клиентом и будет блокирован другим сервером, то этот другой сервер будет опять наследовать, но его собственный приоритет, а не приоритет, временно наследованный от очередного клиента. Тем самым полученная ранее компенсация инверсии приоритетов будет утрачена.

Важно отметить, что наследование сервером приоритета обслуживаемого клиента не запрещает выполняющейся серверной нити при необходимости изменить свой текущий приоритет, например, когда она завершила обработку всех полученных сообщений (послала клиентам ответы), и далее выполняет собственную работу (если сервер сразу переходит в RECEIVE-блокированное состояние, то его приоритет не имеет значения). Для изменения приоритета используется функция:

```
#include <pthread.h>
#include <sched.h>
int pthread_setschedparam(pthread_t* tid, int policy, const struct sched_param* param);
```

Эта функция устанавливает нити, на которую указывает tid, дисциплину диспетчеризации, заданную в policy, и приоритет, заданный в param.

При необходимости можно отказаться от наследования серверной нитью по умолчанию приоритета обслуживаемого клиента, установив соответствующее свойство каналу сервера, к которому присоединяются клиенты. Для этого следует при создании сервером канала в аргументе flags функции ChannelCreate() установить флаг _NTO_CHF_FIXED_PRIORITY, отменяющий наследование. Сервер будет всегда выполняться с собственным приоритетом при обработке сообщения, поступившего по такому каналу.

13. Методы и функции синхронизации нитей

Под синхронизацией нитей понимается согласование хода или порядка выполнения нитей друг с другом или с реальным временем. Соответственно этому QNX предлагает ряд механизмов синхронизации, учитывающих специфику различных потребностей в согласовании поведения нитей [10][12][15].

13.1. Присоединение

Одним из важных аспектов синхронизации нитей является согласование момента начала продолжения работы одной нити с моментом завершения выполнения другой нити. Этот вид синхронизации нитей обеспечивается механизмом, который называется *присоединением*.

Присоединение позволяет одной нити блокироваться в состоянии ожидания завершения выполнения другой нити. Для этого нить, к которой осуществляется присоединение, должна допускать возможность присоединения (т.е. должна быть запущена с атрибутом присоединяющей нити).

Для присоединения к нити с целью ожидания её завершения присоединяющаяся нить должна выполнить функцию:

```
int pthread_join(pthread_t thread,//ID-нити присоединения
                 void **value_ptr /*возвращаемое нитью значение*/
                 );
```

Если значение, возвращаемое нитью при завершении, не представляет интереса, то аргумент value_ptr следует положить равным NULL.

Пример:

```
int *thread(int);//Объявление функции, запускаемой как нить
void main(void){
   int x=5;
   void *value_ret;
   pthread_t thread_id;
   …
      pthread_create(&thread_id, NULL, (void*(*)(void*))thread, (void*)x);
   …
   /*Нить main ждет завершения нити thread*/
      pthread_join(thread_id, &value_ret);
      printf("Нить thread возвратила значение: %d\n", *(int*)value_ret);
   …
}
```

13.2. Барьеры

Барьер - метод синхронизации, позволяющий нити перейти в состояние ожидания у барьера, как у закрытого "шлагбаума", сбора заданного числа нитей. Если нить достигает барьера, а количество нитей, ранее достигших барьера, меньше заданного для барьера значения, то выполнение нити приостанавливается (блокируется). После того, как заданное число нитей соберётся у барьера, все эти нити деблокируются и становятся готовыми для выполнения.

Такой метод синхронизации нитей удобен, когда, например, нитям необходимо "отчитаться друг перед другом" о завершении выполнения ими необходимых действий. Этот факт выражается в их "встрече у барьера".

Создание в программе барьера заключается в определении программного объекта (переменной) системного типа pthread_barrier_t и инициализации его атрибутов с помощью функции:

int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, int count);

При создании барьера указывается ожидаемое количество нитей - count, и атрибутная переменная - attr, устанавливающая его свойства.

Если принимаются свойства барьера по умолчанию, то в качестве значения attr задаётся NULL. Барьер, созданный по умолчанию (в локальной памяти процесса), является *локальным* барьером, доступным нитям только этого процесса. Барьер, явно созданный процессом вне адресного пространства процессов (в разделяемой памяти), можно определить как *глобальный*, доступный нитям разных процессов. Для этого необходимо явно определить в процессе атрибутную переменную attr системного типа pthread_barrierattr_t и проинициализировать её с помощью функции:

int pthread_barrierattr_init(pthread_barrierattr_t *attr);

Для создания глобального барьера необходимо сформировать в атрибутной записи attr соответствующее значение, используя функцию:

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);

Чтобы определить барьер как глобальный, аргумент pshared должен иметь значение PTHREAD_PROCESS_SHARED (по умолчанию - PTHREAD_PROCESS_PRIVAIT).

После того, как барьер по умолчанию или явно определён он используется нитями для синхронизации. Для синхронизации нити барьером она должна выполнить функцию "ожидания у барьера":

int pthread_barrier_wait(pthread_barrier_t *barrier);

Если количество нитей, выполнивших эту функцию, меньше заданного для барьера ожидаемого им количества нитей, то их выполнение приостанавливается (они задерживаются барьером – переходят в блокированное состояние). Как только их количество становится равным заданному, то все они деблокируются и становятся готовыми продолжить выполнение.

Когда необходимость в барьере отпадает, то с целью освобождения системных ресурсов ядра его можно аннулировать с помощью функции:

int pthread_barrier_destroy(pthread_barrier_t * barrier);

Пример:

```
#include <pthread.h>
#include <sync.h>
#include <sys/neutrino.h>
pthread_barrier_t barrier;//Объект типа "барьер"
/**********************/
void *thread1(void* x){
```

```
…
    pthread_barrier_wait(&barrier);//Нить thread1 ждет у барьера
    /*Нити собрались у барьера. Продолжение работы*/
    …
}
/**********************/
void *thread2(void* x){
    …
    pthread_barrier_wait(&barrier);//Нить thread2 ждет у барьера
    /*Нити собрались у барьера. Продолжение работы*/
    …
}
/**********************/
void main(void){
/*Создать барьер со значением счетчика равным 3*/
    pthread_barrier_init(&barrier, NULL, 3);
/*Создать нити thread1 и thread2*/
    pthread_create(NULL, NULL, thread1, NULL);
    pthread_create(NULL, NULL, thread2, NULL);
    pthread_barrier_wait(&barrier);//Нить main ждёт у барьера
/*Нити thread1, thread2 и main собрались у барьера. Продолжение работы всех нитей*/
    …
}
```

13.3. Мутексы

Мутекс - метод синхронизации, обеспечивающий нитям взаимное исключение по отношению к некоторому общему разделяемому нитями ресурсу, не допускающему его одновременное использование более одной нитью. Иными словами, в каждый момент времени ресурсом может владеть не более чем одна нить.

Метод основан на создании в программе объекта системного типа - *мутекс* (MUTual EXclusion - взаимное исключение) и его использовании нитями, при необходимости доступа к разделяемому ресурсу. Логика управления мутексом выражается в его захвате и освобождении нитями. Если некоторая нить выполняет вызов захвата мутекса, уже ранее захваченного другой нитью, то она блокируется ядром до момента освобождения мутекса нитью его захватившей. Когда мутекс освобождается, то ядро завершает вызов захвата мутекса одной из ранее заблокированных нитей и делает её готовой для выполнения.

Мутекс определяется как программный объект системного типа *pthread_mutex_t* с заданными свойствами. Мутекс может быть создан *локальным* или *глобальным*. Локальный мутекс создаётся в адресном пространстве процесса и доступен нитям только этого процесса. Глобальный мутекс необходимо создать вне адресного пространства процесса в разделяемом системном адресном пространстве. Это позволяет сделать его доступным нитям, находящимся в разных процессах. Кроме того, можно выбирать свойства мутекса для управления

последствиями инверсии приоритетов нитей, когда некоторая нить, захватившая мутекс, является причиной блокировки более приоритетных нитей, выполняющих захват этого же мутекса.

13.3.1. Создание мутекса

Для создания мутекса в программе определяется объект (переменная) системного типа *pthread_mutex_t*, который затем инициализируется атрибутной записью со сформированным набором свойств мутекса. Мутекс может быть создан со свойствами по умолчанию или с явно заданными свойствами. Для явного задания свойств создаваемого мутекса необходимо их сформировать в предварительно созданной в программе атрибутной переменной системного типа *pthread_mutexattr_t*, и затем использовать её при явной инициализации свойств мутекса с помощью функции:

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

Если аргумент attr положить равным NULL, то свойства мутекса устанавливаются по умолчанию. По умолчанию мутекс является локальным и доступен нитям только текущего процесса.

13.3.2. Формирование свойств мутекса

Для явного формирования свойств создаваемого мутекса в программе создаётся атрибутная переменная системного типа pthread_mutexattr_t, которую необходимо проинициализировать с помощью функции:

int pthread_mutexattr_init(const pthread_mutexattr_t *attr);

После инициализации атрибутная переменная содержит запись, характеризующая свойства мутекса по умолчанию, которые можно явно изменить.

Свойство глобальности для мутекса (созданного в разделяемой процессами системной памяти) устанавливается в атрибутной переменной с помощью функции:

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);

где аргумент pshared должен иметь значение символической системной константы PTHREAD_PROCESS_SHARED – разделяемый мутекс. Значение PTHREAD_PROCESS_PRIVATE – не разделяемый мутекс, является значением по умолчанию.

В качестве свойства мутексу может быть назначен приоритет. Приоритет мутекса используется для временного изменения приоритета заблокированных мутексом нитей, что позволяет компенсировать инверсию приоритетов нитей. При этом для мутекса можно выбрать один из двух протоколов временного изменения приоритета захватившей его нити. При выборе протокола по умолчанию ядро динамически повышает приоритет захватившей мутекс низкоприоритетной нити до уровня максимального приоритета среди нитей, заблокированных в данный момент любыми мутексами с таким же протоколом. Это повышает способность нити конкурировать за использование процессорного времени с другими незаблокированными нитями. После освобождения нитью мутекса ей возвращается собственный приоритет.

В качестве альтернативы протоколу динамического изменения приоритета нити (по умолчанию) мутексу можно *явно* задать значение приоритета, которым захватившая его нить будет временно наделена до момента освобождения мутекса. Это свойство мутекса

устанавливаются в атрибутной записи с помощью последовательного выполнения двух функций.

Функция:

int pthread_mutexattr_setprotocol(pthread_mutexattr *attr, int protocol);

устанавливает протокол, где аргументу protocol следует явно установить значение PTHREAD_PRIO_PROTECT (по умолчанию - PTHREAD_PRIO_INHERIT).

Функция:

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);

где аргумент prioceiling устанавливает значение приоритета, связываемого с мутексом, в диапазоне возможных приоритетов для непривилегированной нити (от 1 до 63). В итоге нить, захватившая мутекс, будет выполняться с приоритетом, равным наивысшему приоритету из всех приоритетов, явно установленных для мутексов с протоколом PTHREAD_PRIO_PROTECT, захваченных этой нитью, независимо от величины приоритетов, блокированных этими же мутексами других нитей.

Если нить удерживает несколько мутексов с протоколами PTHREAD_PRIO_INHERIT и PTHREAD_PRIO_PROTECT, её приоритет будет равен наивысшему из приоритетов, обеспечиваемых этими мутексами.

13.3.3. Захват мутекса

Для синхронизации нити мутексом она должна выполнить функцию захвата мутекса:

int pthread_mutex_lock (pthread_mutex_t *mutex);

Функция либо блокирует выполнение нити, если мутекс ранее уже был захвачен другой нитью, либо осуществляется захват мутекса нитью. Если в текущий момент мутекс захвачен, то к мутексу образуется очередь из других нитей, выполнивших вызов захвата мутекса. При освобождении мутекса он будет захвачен очередной нитью в соответствии с её приоритетом и временем поступления вызова.

13.3.4. Осторожный захват мутекса

Если блокирование нити при попытке захвата мутекса не допустимо, то следует воспользоваться функцией:

int pthread_mutex_trylock(pthread_mutex_t *mutex);

Если мутекс свободен, то он будет захвачен. В противном случае - нить продолжит своё выполнение. Для анализа результата попытки захвата мутекса нитью необходимо контролировать возвращаемое функцией значение:

EOK – успешный захват мутекса;

EBUSY – мутекс уже захвачен другой нитью;

EINVAL – недопустимый мутекс;

EAGAIN – недостаточно ресурсов системы для реализации запроса на захват мутекса.

13.3.5. Освобождение мутекса

Если нить, захватившая мутекс, завершает свой исключительный доступ к общему ресурсу, то она должна освободить мутекс с помощью функции:

int pthread_mutex_unlock (pthread_mutex_t *mutex);

13.3.6. Уничтожение мутекса

Если необходимость в мутексе отпадает, то, для освобождения выделенных ему ядром системных ресурсов, его целесообразно уничтожить с помощью функции:

int pthread_mutex_destroy (pthread_mutex_t *mutex);

Пример:

```
//Создание мутекса
pthread_mutex_t mutex;//Определение переменной для управления мутексом
…
pthread_mutex_init (&mutex,NULL);//Инициализация мутекса с атрибутами по умолчанию
…
/*
Захват мутекса, если он уже захвачен, то ядро переводит нить в состояние ожидания
освобождения мутекса
*/
pthread_mutex_lock (&mutex);
/*
Исключительный доступ к ресурсу, контролируемому мутексом
…
*/
/* Завершение доступа к ресурсу, освобождение мутекса */
pthread_mutex_unlock (&mutex);
…
/* Уничтожение мутекса */
pthread_mutex_destroy (&mutex);
…
```

Создание мутекса можно отложить до его первого использования. Для этого при определении мутекса ему необходимо присвоить начальное значение равное системной константе PTHREAD_MUTEX_INITIALIZER. Это указание ядру, что при первом обращении к мутексу его необходимо предварительно создать.

Пример:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
/*
Мутекс помечен как требующий создания
*/
…
/* Первый захват мутекса. Ядро предварительно его создаст */
```

```
pthread_mutex_lock (&mutex);
…
```

13.3.7. Создание рекурсивного мутекса

В некоторых случаях требуется определять мутекс как рекурсивный (считающий). Это необходимо, когда выполнение нити, захватившей мутекс, предполагает, например, вызов различных функций, которые в свою очередь в процессе выполнения осуществляют захват этого же мутекса. Если не допускать повторного захвата мутекса одной и той же нитью, то возникнет "мёртвая" блокировка (deadlock). Рекурсивный мутекс позволяет избежать этого. Повторный захват одной и той же нитью рекурсивного мутекса не приводит к блокированию нити. При этом ядро контролирует количество захватов и освобождений рекурсивного мутекса.

Чтобы мутекс был рекурсивным его при определении необходимо явно проинициализировать значением PTHREAD_RMUTEX_INITIALIZER.

Пример:

```
pthread_mutex_t func_mutex=PTHREAD_RMUTEX_INITIALIZER;/*Помечен как рекурсивный*/
…
high_level_func(){
pthread_mutex_lock (&func_mutex);
//Критическая секция
…
        low_level_func();
        …
//Конец критической секции
        pthread_mutex_unlock (&func_mutex);
}
low_level_func(){
        pthread_mutex_lock (&func_mutex);
//Критическая секция
        …
//Конец критической секции
        pthread_mutex_unlock (&func_mutex);
}
```

13.4. Блокировки чтения/записи

Блокировка *чтения/записи* - метод синхронизации, согласующий поведение нитей по отношению к содержимому общей области памяти, требуемой одновременно нескольким нитям для чтения или записи. При этом нити-*читатели* могут одновременно читать содержимое памяти, исключая при этом доступ к ней нитей-*писателей*, или одна нить-*писатель* может изменять содержимое памяти, исключая при этом доступ к ней нитей-*читателей* и других нитей-*писателей*.

Блокировка чтения/записи определяется как программный объект системного типа pthread_rwlock_t с заданными свойствами. А именно, блокировка чтения/записи может быть *локальной* или *глобальной*. Локальная блокировка чтения/записи доступна нитям только одного

процесса, которому они принадлежат. Глобальная блокировка чтения/записи должна быть создана в разделяемой системной памяти как разделяемая процессами, она позволяет синхронизировать нити, находящиеся в разных процессах.

13.4.1. Создание блокировки чтения/записи

Блокировка чтения/записи создаётся в программе как объект системного типа pthread_rwlock_t. Для формирования блокировки чтения/записи с явно заданными свойствами необходимо предварительно сформировать её свойства в созданной в программе и проинициализированной атрибутной переменной системного типа pthread_rwlockattr_t, и затем сформированную атрибутную запись использовать для инициализации созданной блокировки чтения/записи с помощью функции:

int pthread_rwlock_init (pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);

Если аргумент attr положить равным NULL, то свойства устанавливаются по умолчанию.

13.4.2. Свойства блокировки чтения/записи

Свойства блокировки чтения/записи устанавливаются явно с помощью атрибутной переменной системного типа pthread_rwlockattr_t, которую необходимо определить в программе и затем проинициализировать с помощью функции:

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

Блокировка чтения/записи проинициализированная по умолчанию является локальной. Для создания глобальной блокировки чтения/записи она должна находиться в разделяемой процессами системной памяти ОСРВ. Свойство глобальной блокировки чтения/записи устанавливается в атрибутной переменной с помощью функции:

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);

Аргумент pshared должен получить значение PTHREAD_PROCESS_SHARED (значение по умолчанию - PTHREAD_PROCESS_PRIVATE). Заметим, однако, что эта возможность поддерживается только в том случае, если в системном заголовочном файле unistd.h определена системная константа _POSIX_THREAD_PROCESS_SHARED:

define _POSIX_THREAD_PROCESS_SHARED

Для проверки установленного свойства блокировки чтения/записи используется функция:

int pthread_rwlockattr_getpshared(pthread_rwlockattr_t *attr, int *valptr);

Значение свойства "разделяемости" блокировки чтения/записи - PTHREAD_PROCESS_SHARED или PTHREAD_PROCESS_PRIVATE, помещается в переменную целого типа, на которую указывает valptr.

Для своевременного освобождения выделенных программе системных ресурсов ядра ОСРВ атрибутную запись attr можно аннулировать с помощью функции:

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);

13.4.3. Захват блокировки чтения/записи

Для синхронизации нити-*читателя* блокировкой чтения/записи она должна выполнить функцию захвата блокировки чтения/записи для чтения:

int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);

Функция либо блокирует выполнение нити-*читателя*, если блокировка чтения/записи ранее уже была захвачена другой нитью-*писателем*, либо осуществляется захват блокировки чтения/записи нитью.

Для синхронизации нити-*писателя* блокировкой чтения/записи она должна выполнить функцию захвата блокировки чтения/записи для записи:

int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);

Функция либо блокирует выполнение нити-*писателя*, если блокировка чтения/записи ранее уже была захвачена другой нитью-*писателем* или нитью-*читателем*, либо осуществляется захват блокировки чтения/записи нитью-писателем.

13.4.4. Осторожный захват блокировки чтения/записи

Если блокирование нити при попытке захвата блокировки чтения/записи не допустимо, то нити-*читателю* следует воспользоваться функцией:

int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);

Нити-*писателю* следует воспользоваться функцией:

int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);

При этом необходимо контролировать значения, возвращаемые функциями pthread_rwlock_trywrlock() и pthread_rwlock_tryrdlock(). Если блокировка чтения/записи свободна, то она будет захвачена нитью. В противном случае - нить получит отказ захвата и продолжит своё выполнение. В этом случае для анализа ситуации нити необходимо контролировать возвращаемое функциями значение:

EOK – успешный захват блокировки чтения/записи;

EBUSY – отказ, блокировка чтения/записи ранее уже захвачена.

13.4.5. Освобождение блокировки чтения/записи

Если нить, захватившая блокировку чтения/записи, завершает свой доступ к общему ресурсу, то она должна освободить блокировку чтения/записи с помощью функции:

int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);

13.4.6. Уничтожение блокировки чтения/записи

Если необходимость в блокировке чтения/записи отпадает, то для освобождения выделенных ей ядром системных ресурсов, её целесообразно уничтожить с помощью функции:

int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);

Все функции в случае успешного завершения возвращают EOK, а в случае ошибки - положительное значение (код ошибки):

EINVAL – недопустимая блокировка чтения/записи;

EAGAIN – недостаточно ресурсов системы для захвата блокировки чтения/записи.

13.5. Условные переменные

В ряде случаев захват ресурса не предполагает безусловную готовность ресурса к его обработке захватившей нитью. Нить предварительно должна проверить условие готовности ресурса к обработке. Если условие выполняется, то обработка осуществляется и при её завершении ресурс освобождается. Если условие не выполняется, то нить вынуждена освобождать ресурс, чтобы предоставить возможность доступа к нему другим нитям, которые, в общем случае, могут влиять на формирование результата проверяемого условия. Например, нить должна дождаться, когда X станет равным Y прежде, чем продолжит своё выполнение. Если использовать только мутексы, то получим следующее решение:

```
pthread_mutex_t condMutex;
//Создание мутекса
        pthread_mutex_init (&condMutex,NULL);
        …
        pthread_mutex_lock(&condMutex);
        while(x!=y){
        pthread_mutex_unlock(&condMutex);
//Немного подождать, чтобы позволить изменить значения x, y
        pthread_mutex_lock(&condMutex);
}
//Выполнить работу
        pthread_mutex_unlock(&condMutex);
…
```

Очевидно, что в этом случае проверка условия (опрос) осуществляется явно, а хотелось бы не заниматься опросом для выяснения этого события. Применение метода условной переменной позволяет избежать опроса.

Разумно нити вновь пытаться захватывать ресурс только в том случае, если произошли какие-то события, могущие повлиять на условие готовности ресурса к обработке данной нитью. А до этого находиться в состоянии ожидания таких событий. Метод условной переменной как раз и предполагает реализацию такого механизма синхронизации (входит в стандарт POSIX). Метод основан на управлении мутексом и программным объектом системного типа *pthread_cond_t*, называемым *условной переменной*. Метод позволяет одним нитям ждать уведомления, посылаемые другими нитями, использующими в качестве посредника одну и ту же условную переменную.

Функции метода, следующие:

```
/*Создание условной переменной*/
int pthread_cond_init (pthread_cond_t *condvar, pthread_condattr_t * attr);
/*Ожидание уведомления по условной переменной*/
int pthread_cond_wait (pthread_cond_t * condvar, pthread_mutex_t *mutex);
/* Отправка уведомления по условной переменной */
int pthread_cond_broadcast (pthread_cond_t *condvar);
int pthread_cond_signal (pthread_cond_t *condvar);
```

В отличие от безусловного использования мутекса метод условной переменной позволяет нити переходить в состояние ожидании уведомлений без необходимости явного освобождения нитью захваченного мутекса. При этом ядро неявно освобождает мутекс, когда нить переходит в состояние ожидания уведомления, и захватывает мутекс, когда выводит нить из состояния ожидания, поддерживая для нити иллюзию её владения мутексом и, следовательно, ресурсом. Из состояния ожидания нить выводится, если какая-то другая нить пошлёт уведомление по условной переменной, выполнив функцию pthread_cond_signal() или pthread_cond_broadcast(). Отличие этих функций в том, что в случае отправки уведомления функцией pthread_cond_signal() из состояния ожидания будет выведена только одна из возможно нескольких заблокированных мутексом нитей, ждущих уведомления. Это будет нить с максимальным приоритетом и максимальным временем ожидания среди прочих нитей, заблокированных этим мутексом. Остальные нити будут продолжать ждать следующего уведомления. Если же уведомление послано функцией pthread_cond_broadcast(), то из состояния ожидания одновременно выводятся все нити, ожидающие его по соответствующей условной переменной. Последовательность, в которой ядро будет восстанавливать для них исключительный доступ к ресурсу, определяется приоритетами нитей, а если приоритет одинаковый, то временем ожидания.

Выведенная из состояния ожидания нить должна вновь проверить возможность использования ресурса и затем либо выполняет работу, либо обратно возвращается в состояние ожидания уведомления.

Отметим, что между мутексом и условной переменной нет прямой зависимости. Одна и та же условная переменная может использоваться для получения нитью уведомления при захвате различных мутексов. Кроме того, можно анализировать состояние одного и того же ресурса (один мутекс), используя поочерёдно различные условные переменные.

Рассмотренный выше пример, при использовании метода условной переменной, примет следующий вид:

```
…
//*********************
//Нить 1
//*********************
…
        pthread_mutex_lock(&condMutex);
        while(x!=y){
//Ждать, пока нить 2 или 3 не изменят значения x или y
        pthread_cond_wait (&condvar, &condMutex);
}
//Выполнить работу
        pthread_mutex_unlock(&condMutex);
…
}
//*********************
//Нить 2
//*********************
```

```
…
pthread_mutex_lock(&condMutex);
//Модифицировать значение x
        pthread_cond_signal (&condvar);
        pthread_mutex_unlock(&condMutex);
…
//*********************
//Нить 3
//*********************
…
pthread_mutex_lock(&condMutex);
//Модифицировать значение y
        pthread_cond_signal (&condvar);
        pthread_mutex_unlock(&condMutex);
…
// **********Окончание Нить 3***********

//*********************
void main(void){
…
…
pthread_mutex_t condMutex;
pthread_cond_t condvar;
        pthread_mutex_init (&condMutex,NULL); /*Инициализация мутекса по умолчанию*/
        pthread_cond_init (&condvar, NULL); /*Инициализация условной переменной по
        умолчанию*/
//Создание нитей 1,2 и 3
        …
}
```

Локальная условная переменная, инициированная в процессе по умолчанию, не может разделяться нитями различных процессов. Однако можно создать условную переменную в глобальной (именованной) памяти и с помощью явной установки атрибутов pthread_condattr_t *attr сделать её разделяемой нитями разных процессов. Параметр attr управляется с помощью функций:

```
int pthread_condattr_init( pthread_ condattr_t *attr );
int pthread_condattr_destroy( pthread_ condattr_t *attr );
int pthread_condattr_getpshared( pthread_ condattr_t*att, int *valptr );
int pthread_condattr_setpshared( pthread_ condattr_t, int value );
```

Эти функции позволяют создать (pthread_condattr_init()) или аннулировать (pthread_condattr_destroy()) атрибутную запись attr, получить ткущее значение атрибута в виде целого, на которое указывает valptr (pthread_condattr_getpshared()) или установить значение атрибута равное значению value (pthread_condattr_setpshared()) Значение value может быть либо

PTHREAD_PROCESS_PRIVATE, либо PTHREAD_PROCESS_SHARED. Последнее значение позволяет совместное использование процессами условной переменной, расположенной в разделяемой процессами области памяти. Обычно и соответствующий мутекс также делают разделяемым, но не обязательно

13.6. Ждущие блокировки

Этот метод является частным случаем метода условной переменной и не относится к стандарту POSIX. Метод неявно использует уникальный системный объект sleepon - "*ждущая блокировка*", играющего роль неявно создаваемого ядром системы мутекса и неявно используемой ядром системы в роли условной переменной любая доступной нитям переменной программы. При этом яром системы для согласования ожидающих и уведомляющих нитей будет использоваться только уникальная ссылка (адрес) этой переменной, а содержимое переменной не затрагивается. С одной ждущей блокировкой в качестве аналогов условной переменной одновременно могут использоваться разные переменные программы.

Функции управления ждущей блокировкой, следующие:

```
/*Захват ждущей блокировки*/
int pthread_sleepon_lock (void);
/*Освобождение ждущей блокировки*/
int pthread_sleepon_unlock (void);
/*Ожидание уведомления*/
int pthread_sleepon_wait (const volatile void *addr);
/*Отправка уведомления*/
int pthread_sleepon_broadcast(const volatile void *addr);
int pthread_sleepon_signal (const volatile void *addr);
```

Для доступа к ресурсу нить должна захватить ждущую блокировку. При успешном захвате, если условие доступа к ресурсу не выполняется, то нить может перейти в состояние ожидания уведомления по указанному адресу в вызове pthread_sleepon_wait(). Другие нити могут посылать по нужному адресу уведомления о наступлении события, используя функции pthread_sleepon_signal() или pthread_sleepon_broadcast(). Отличие этих функций в том, что в случае pthread_sleepon_signal() будет разБлокирован только одна из ждущих уведомления по указанному адресу нитей с наивысшим приоритетом. Если приоритет одинаковый, то порядок выбора активируемой нити *не определён!* А в случае использования функции pthread_sleepon_broadcast() будут разблокированы все ожидающие нити. При этом ядро затем обеспечивает для них корректный порядок продолжения выполнения и исключительного доступа к ресурсу с учётом их приоритета и длительности ожидания.

Пример:

```
/* Нить - потребитель данных, поставляемых устройством */
volatile int data_ready=0; //флаг готовности данных в устройстве
Consumer(){
        while(1){
                pthread_sleepon_lock();
                while(!data_ready){
                pthread_sleepon_wait(&data_ready);/*используется только адрес переменной
```

```
                                                    data_ready*/
        }
//Обработать данные
data_ready=0;
        pthread_sleepon_unlock();
}
/* Нить - информирующая о событии возникновения данных в устройстве */
Producer(){
        while(1){
        //ждать прерывания от оборудования …
        pthread_sleepon_lock();
        data_ready=1;
        pthread_sleepon_signal(&data_ready);
        }
        pthread_sleepon_unlock();
}
```

Замечание. Метод обеспечивается "родными" функциями QNX/Neutrino, не предусмотренными стандартом POSIX. Их использование ограничивает мобильность приложений.

13.7. Семафоры

Семафоры — это метод синхронизации, основанный на создании и управлении программным объектом системного типа `sem_t`, регулирующим количество нитей, осуществляющих одновременный доступ к некоторому ресурсу. Управление семафором выражается в его захвате и освобождении. Семафор ведёт счётчик захватов. Начальное значение счётчика захватов семафора устанавливается при его создании. Если нить пытается захватить семафор, счётчик которого не больше 0, то она блокируется до момента, когда значение счётчика не станет больше 0. Семафоры бывают *неименованные* и *именованные*.

13.7.1. Неименованный семафор

Для создания неименованного семафора необходимо предварительно определить переменную типа sem_t и затем выполнить функцию инициализации неименованного семафора, используя указатель этой переменной:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned value);
```

Если аргумент pshared отличен от нуля, то семафор может разделяться нитями различных процессов, при условии, что семафор создан в разделяемой процессами памяти. С помощью аргумента value задаётся начальное значение счётчика семафора. После инициализации семафора указатель sem используется в функциях управления семафором.

Для аннулирования неименованного семафора используется функция

```
int sem_destroy(sem_t *sem);
```

При аннулировании семафора освобождаются соответствующие системные ресурсы ядра.

13.7.2. Именованные семафоры

Именованный семафор создаётся как файл особого типа, регистрируемый в файловой системе в каталоге /dev/sem. Для создания именованного семафора необходимо предварительно определить переменную-указатель именованного семафора типа sem_t*, и затем выполнить функцию открытия именованного семафора - sem_open(), используя эту переменную для получения значения указателя семафора, возвращаемого этой функцией. Функция sem_open() позволяет создать в файловой системе новый именованный семафор и открыть к нему доступ, если семафор с указанным именем в системе отсутствует, либо только открыть доступ к уже существующему с указанным именем семафору. Поэтому функция sem_open() является функцией с переменным числом аргументов. Если требуется открыть доступ к именованному семафору, а в случае его отсутствия создать новый семафор с указанным именем, выполняется вызов функции:

sem_t* sem_open(const char *sem_name, int oflags, mode_t mode, unsigned int value);

Если требуется только открыть уже существующий именованный семафор, а в случае его отсутствия создавать новый семафор не требуется, функция используется без последних двух аргументов:

sem_t* sem_open(const char *sem_name, int oflags);

Функция sem_open() создает и/или открывает в каталоге /dev/sem именованный семафор, возвращая дескриптор управления семафором. Значение аргумента sem_name должно начинаться с символа </>, например, /myprog.sem1. Имена семафоров должны быть меньше чем (NAME_MAX - 8) символов. Например, семафор с именем /myprog.sem1 в файловой системе будет учитываться как /dev/sem/myprog.sem1. QNX позволяет использовать в именах семафоров более одного символа </>. Однако для поддержания POSIX-мобильности не рекомендуется использовать в именах семафоров более одного символа </>.

Аргумент oflags управляет возможностью создания нового семафора. Если при отсутствии семафора с указанным именем предполагается создание нового семафора, то в oflags должен быть установлен флаг O_CREAT. Если нет - (O_CREAT|O_EXCL).

O_CREAT – указывает на возможность создания нового именованного семафора. Если именованный семафор с указанным именем уже существует, то к нему будет открыт доступ. Если такого семафора нет, то он будет создан. При создании нового именованного семафора будут использованы значения аргументов mode и value. Аргумент mode указывает режимы доступа к семафору (точно так же, как при создании файла), а value задает начальное значение семафора (не должно превышать SEM_VALUE_MAX). Значение value > 0 делает семафор открытым, а значение равное 0 означает, что создаваемый семафор будет закрытым. Для мобильности в mode, необходимо установить флаги доступа для чтения, записи и выполнения, используя константы из <sys/stat.h>: S_IRWXG - доступ для группы, S_IRWXO - доступ для других, S_IRWXU - например: mode=S_IRWXG|S_IRWXO|S_IRWXU.

(O_CREAT|O_EXCL) – если семафор с указанным именем уже существует, то функция возвращает SEM_FAILED, и в системную переменную errno записывается код ошибки EEXIST, иначе семафор создаётся и к нему открывается доступ.

Заметим, что не требуется устанавливать в oflags флаги O_RDONLY, O_RDWR, или O_WRONLY. Поведение семафора с этими флагами не определено. Функции QNX игнорируют эти флаги, но их использование может понизить мобильность программного кода.

Функция возвращает указатель на семафор или -1 в случае неудачи, а в системной переменной errno устанавливается номер ошибки.

Доступ к именованному семафору можно закрыть, используя функцию

int sem_close(sem_t * sem);

Если требуется аннулировать именованный семафор, то используется функция

int sem_unlink(const char * sem_name);

Функция sem_unlink() уничтожает открытые именованные семафоры таким же образом, как удаляются открытые файлы. То есть, процессы, которые имеют открытый семафор, могут все ещё использовать его, но семафор исчезнет, как только последний процесс использует функцию sem_close(), чтобы закрыть доступ к семафору. Попытка выполнить sem_open() по отношению к уничтоженному семафору будет рассматриваться как создание нового семафора.

Семафоры сохраняются в системе не зависимо от создавших их процессов и существуют в ней пока не будут явно уничтожены или система не завершит свою работу.

13.7.3. Управление семафорами

Управление неименованными и именованными семафорами осуществляется с помощью одних и тех же функций:

```
#include <time.h>
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_timedwait (sem_t * sem,
const struct timespec * abs_timeout);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t *sem, int *value);
```

Функция sem_wait() уменьшает значение счётчика семафора на 1. При этом, если значение семафора оказывается не большее ноля, то нить, вызвавшая функцию, блокируется до тех пор, пока счётчика семафора не станет больше 0. Предполагается, что в какой-то момент времени некоторая другая нить, вызовет функцию sem_post(), увеличивая счётчик семафора на 1. Заметим, что запрос может быть завершён и в связи с приходом сигнала. Поэтому для корректного выхода из блокированного состояния sem_wait() необходимо контролировать возвращаемое функцией значение: 0 – значение семафора уменьшилось, -1 - значение семафора не изменилось.

Функция sem_trywait() уменьшает значение счётчика семафора на 1, если его значение большее ноля (возвращает значение 0 – значение семафора уменьшилось). В противном случае счётчик семафора не изменяется, и функция завершается, возвращая значение -1.

Функция sem_timedwait() захватывает семафор sem, как и функция sem_wait(). Однако, если семафор не может быть захвачен, то ожидание завершается, когда указанное время ожидания истекает. Время ожидания истекает, когда проходит момент абсолютного времени, указанный в abs_timeout. При этом время измеряется системными часами, на которых базируются таймауты (то есть, когда значение тех часов равняется или превышает abs_timeout),

или если абсолютное время, указанное abs_timeout, уже прошло во время реализации запроса. Если выбор таймеров поддерживается, то указание ожидаемого момента времени базируется на часах реального времени CLOCK_REALTIME. Если выбор таймеров не поддерживается, то указание ожидаемого момента времени базируется на системных часах, время которых возвращается функцией time().

Функция sem_timedwait() уменьшает значение счетчика семафора на 1, если его значение большее ноля, и возвращает значение 0. В противном случае счётчик семафора не изменяется, а функция завершается, возвращая значение -1, а в errno устанавливается код ошибки ETIMEDOUT – прошёл указанный абсолютный момент времени.

Функция sem_post() увеличивает счётчик семафора sem на 1. Если имеются нити, которые в настоящее время блокированы, ожидая семафор, то одна из этих нитей возвратится успешно из вызова sem_wait. Нить, которая будет разБлокирован первой, определяется в соответствии с приоритетом и временем ожидания (с наибольшим приоритетом, которая ждала дольше всех). Функция sem_post() может быть вызвана обработчиком сигналов.

Функция sem_getvalue() позволяет определить текущее значение счётчика семафора sem, которое заносится по адресу, заданному в value.

В случае успеха все функции возвращают значение 0. В противном случае возвращается -1 и в errno устанавливается код ошибки.

Именованные семафоры работают медленнее, чем неименованные семафоры. Но зато нити разных процессов могут открывать доступ к именованному семафору как к файлу, указывая его символическое имя, и использовать его как глобальный.

Пример:

```
#include <stdio.h>
#include <semaphore.h>
#include <time.h>

main(){

  struct timespec tm;
  sem_t sem;
  int i=0;

  sem_init(&sem,0,0);//Не именованный локальный семафор, счётчик = 0

  do {
    clock_gettime(CLOCK_REALTIME, &tm);//Текущее время
    tm.tv_sec += 1;//Увеличить на 1 сек
    i++;
    printf("i=%d\n",i);
    if(i==10) sem_post(&sem);//Увеличить счётчик семафора на 1
  } while (sem_timedwait(&sem, &tm) == -1);//таймаут
```

```
    printf("Семафор захвачен после %d таймаутов\n", i);
    return;
}
```

14. Разделяемая системная память

Нити не могут осуществлять прямой доступ к адресам памяти за пределами выделенного процессу ядром QNX адресного пространства (локальная память процесса). Такие адреса памяти называются системными областями памяти или системной памятью и доступ к ним контролируется операционной системой. К системной памяти относятся:

- *именованная* область оперативной памяти;
- адреса памяти, связанные с *физическими* устройствами.

Если процессам требуется доступ к адресам физических устройств или возникает потребность в оперативной памяти за пределами локальной памяти процесса (например, разделяемой различными процессами), то операционная система предоставляет процессам такую возможность посредством механизма отображения адресов системных областей памяти в адресное пространство процесса. Отображение адресов системных областей памяти реализуется процессами с помощью специальных запросов к операционной системе, в результате успешного выполнения которых между адресами системной памяти и выделенными адресами адресного пространства процесса устанавливается взаимно-однозначное соответствие. В результате обращение процесса к этим адресам памяти преобразуется операционной системой в обращение к соответствующим адресам системной памяти.

Области адресов, связанные с физическими устройствами, однозначно определены архитектурой вычислительной платформы и заранее известны. Области же оперативной памяти прежде, чем быть отображёнными в память процесса, должны быть предварительно выделены операционной системой из общего объёма системной оперативной памяти. При этом они специфицируются как именованные области оперативной памяти или коротко - *именованная память* [10]. Создаваемая именованная память регистрируется в файловой системе ОС как файл устройства специального типа. Для доступа к именованной памяти, зарегистрированной как файл в файловой системе, процесс должен предварительно присоединить её к себе, получив в результате дескриптор присоединённой именованной памяти.

14.1. Создание именованной памяти

Для создания и/или открытия (присоединения) существующей именованной памяти процессом используется функция:

```
#include <fcntl.h>
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

Функция shm_open() создаёт и/или присоединяет к процессу именованную память с заданным именем name и возвращает дескриптор именованной памяти как дескриптор файла с установленным флагом FD_CLOEXEC. Такие дескрипторы не наследуются дочерними процессами (см. функцию fcntl()), и они должны присоединять существующую именованную память самостоятельно.

Аргумент name должен содержать в качестве значения символьную строку, определяющую место в файловой системе и имя создаваемой или открываемой именованной памяти в файловой системе. Обозначим имя текущего рабочего каталога процесса как CWD (Current Working Directory). Тогда возможные варианты символьных строк в аргументе name и

соответствующие им места расположения именованной памяти с именем, например, sharemap в файловой системе будут следующие:

- "sharemap" - */CWD/sharemap*;
- "/sharemap" - */dev/shmem/sharemap*;
- "catalog/sharemap" - *CWD/catalog/sharemap*;
- "/catalog/sharemap" - */catalog/sharemap*.

Длина строки в name не должна превышать значения, представленного системной константой NAME_MAX.

Режим присоединения к именованной памяти определяется значениями флагов, указанных в аргументе oflag. Значение oflag формируется операцией поразрядного логического сложения следующих флагов, представленных системными константами, которые определены в <fcntl.h>:

O_RDONLY - открыть только для чтения.

O_RDWR - открыть для чтения и записи.

O_CREAT – выполнить присоединение к ранее созданной именованной памяти. Иначе, именованная память создаётся и регистрируется в файловой системе с правами доступа владельцев, установленными в соответствии со значением аргумента mode и маской прав доступа, назначенных процессу (атрибут процесса) для создания файлов. В результате именованная память присоединяется к процессу с режимами доступа, заданными значениями флагов, указанных в аргументе oflag:

O_CREAT | O_EXCL – эксклюзивное создание именованной памяти. Если именованная память с указанным именем в файловой системе не зарегистрирована, то будет создана и зарегистрирована в файловой системе новая именованная память. Если именованная память с указанным именем существует, то shm_open() возвращает ошибку. Проверка существования именованной памяти и её создание, если она не существует, является атомарной операцией по отношению к другим процессам, также выполняющим shm_open(), указывая ту же самую именованную память с набором флагов O_CREAT|O_EXCL. Это обеспечивает корректность одновременного выполнения функции shm_open() несколькими процессами при условии создания новой именованной памяти.

O_TRUNC - если именованная память существует, и она успешно присоединена с флагом O_RDWR, то память урезается до нулевой длины, а права доступа владельца и владелец не изменяются. Размер памяти может затем быть установлен с помощью функции ftruncate().

Аргумент mode используется процессом для формирования прав доступа пользователей к создаваемой именованной памяти таким же образом, как при создании обычного файла, с учётом установленной процессу маски, ограничивающей его возможности по формированию прав доступа (см. описание функции umask()).

При успешном выполнении функция возвращает неотрицательное целое положительное число, которое является дескриптором именованной памяти. Если при этом произошло создание новой именованной памяти, то в соответствующем каталоге появиться файл с именем памяти.

В случае ошибки - значение (-1) и заносит код ошибки в системную переменную errno.

Ошибки:

EACCES - В создании именованной памяти или присоединении к ней отказано, либо объект именованной памяти уже существует, а указанные в oflag флаги доступа не соответствуют установленным правам доступа, либо указан флаг O_TRUNC, и разрешение на запись отклонено.

EEXIST - Установлены флаги O_CREAT и O_EXCL, а именованный объект общей памяти уже существует.

EINTR - Вызов shm_open() был прерван сигналом.

EINVAL - Базовый вызов resmgr_open_bind() завершился неудачей.

ELOOP - Слишком много уровней символических ссылок или префиксов.

EMFILE - Слишком много файловых дескрипторов используется этим процессом.

ENAMETOOLONG - Длина аргумента name превышает NAME_MAX.

ENFILE - Открыто слишком много объектов общей памяти.

ENOENT - Флаг O_CREAT не задан, а именованный объект общей памяти не существует, или O_CREAT задан, и либо префикс имени не существует, либо аргумент name указывает на пустую строку.

ENOSPC - Недостаточно места для создания нового объекта общей памяти.

ENOSYS Функция shm_open() не поддерживается в этой версии.

Заметим, если в результате выполнения функции shm_open() была создана и зарегистрирована новая именованная память (файл) или процессом была открыта существующая именованная память с использованием флага O_TRUNC, то размер памяти будет равен нулю. Нужный размер памяти необходимо установить с помощью функции ftruncate():

```
#include <unistd.h>
int ftruncate( int fildes,//дескриптор именованной памяти
        off_t length );//длина в байтах
```

Если необходимость в доступе к именованной памяти у процесса отпадает, он может отсоединиться от неё, выполнив обычную функцию закрытия дескриптора файла - close(). При этом объект именованной памяти, включая содержимое, хранится в файловой системе до тех пор, пока она не будет явно удалена из файловой системы функцией:

```
#include <sys/mman.h>
int shm_unlink(const char *name);
```

Замечание. Функция shm_unlink() может быть выполнена процессами многократно и асинхронно. Однако реальное удаление будет отложено операционной системой до момента, когда будет закрыт последний дескриптор доступа к именованной памяти и она не будет присоединена ни к одному процессу.

14.2. Организация доступа к именованной памяти

Открытие процессом именованной памяти не открывает процессу сразу же доступа к ней, а только по запросу к ядру операционной системы отобразить требуемую ему область именованной памяти нужного размера в адресное пространство процесса. Через это собственное адресное пространство процесс и получает возможность доступа к открытой им именованной памяти. Отображение осуществляется с помощью функции

```
#include <sys/mman.h>
void *mmap( void *addr, // начальный адрес
            size_t len, // длина в байтах
            int prot, // порядок использования
            int flags, // порядок отображения
            int fd, // дескриптор
            off_t off // смещение
            );
```

В результате выполнения функции в адресное пространство процесса отображается выделенная в пределах именованной памяти с дескриптором fd область, начинающаяся со смещением off от её начала и длиною len байтов. Доступ к этой области именованной памяти в адресном пространстве процесса начинается с адреса, возвращённого функцией mmap(). Аргумент addr – указывает начальный адрес области памяти процесса, с которой заданная область именованной памяти будет ассоциирована (отображена, например, в специально созданный в программе массив). Однако, обычно нет принципиальной необходимости явно указывать в процессе какое-то конкретное место отображения заданной области именованной памяти, можно просто addr задать NULL. Однако, если addr не NULL, то нет гарантии, что область именованной памяти будет отображена в указанное процессом адресное пространство. Если в аргументе flags установлен флаг MAP_FIXED, то либо область именованной памяти отображается с адреса addr, либо функция завершается с ошибкой. Если флаг MAP_FIXED не установлен, то значение addr рассматривается как желаемый адрес начала области доступа к именованной памяти в адресном пространстве процесса, но не обязательный и заданное значение addr может быть проигнорировано.

Аргумент prot определяет порядок использования именованной памяти. В prot можно устанавливать следующие флаги (символические константы флагов определены в <sys/mman.h>).

PROT_EXEC – память может быть использована для размещения исполняемых модулей.

PROT_NOCACHE - запрещает кэширование памяти (например, может использоваться, чтобы обратиться к двухпортовой памяти).

PROT_NONE – запрещает какой-либо доступ к памяти.

PROT_READ – разрешает доступ к памяти для чтения.

PROT_WRITE – разрешает доступ к памяти для записи.

Аргумент flags определяет возможность разделения отображённой в процесс области именованной памяти с другими процессами:

MAP_PRIVATE – отображаемая область памяти не может разделяться с другими процессами.

MAP_SHARED – отображаемая область памяти может разделяться с другими процессами.

Функция возвращает начальный адрес отображения именованной памяти в локальной памяти процесса или MAP_FAILED в случае ошибки. Код ошибки помещается в errno.

Замечания:

- Гарантируется, что новое отображение не будет перекрывать никакое из ранее выполненных процессом отображений.
- Свойства присоединённой именованной памяти можно изменить, используя функцию shm_ctl().

– Для отображения в процесс системных адресов памяти доступа к устройствам следует использовать функцию mmap_device_memory() (см. ниже).

Пример.

```
/* Присоединение или создание разделяемой именованной памяти с именем "/common" со всеми правами доступа для всех пользователей*/
fd = shm_open("/common", O_RDWR, 0777);
addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

Если необходимость в отображении отпала, то его можно аннулировать с помощью функции munmap().

```
#include <sys/mman.h>
int munmap( void * addr, size_t len );
```

Важно отметить, что функция munmap() аннулирует все ранее выполненные отображения, пересекающиеся с диапазоном адресов, начинающимся в addr и длиною len байтов (округлённый в большую сторону кратно размеру страницы страничной памяти). Если нет никаких отображений в определённом адресном интервале, то munmap() не имеет никакого эффекта. Функция возвращает в случае ошибки значение -1, а код ошибки заносится в errno. Любое другое значение означает успешное завершение.

Пример:

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/stat.h>

char *progname = "sharemem";

void main(int argc, char *argv[]){//получает имя памяти в argv[1]
    int fd, len, i;
    char *ptr, *name;

        if(argc != 3){
        fprintf(stderr, "Ошибка параметров вызова\n");
exit(EXIT_FAILURE);
}
        name = argv[1];//Имя именованной памяти
        len = atoi(argv[2]);//Длина именованной памяти
```

```
/* Присоединить существующую именованную память для чтения и записи*/
        fd = shm_open(name, O_RDWR, 0);
        if (fd == -1){
            fprintf(stderr, "%s: Ошибка присоединения именованной памяти'%s': %s\n", progname,
            name, strerror(errno));
            exit(EXIT_FAILURE);
        }

/* Отображение разделяемой именованной памяти для чтения и записи*/

ptr = mmap(0,len,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
if(ptr == MAP_FAILED){
fprintf(stderr, "%s: Ошибка отображения: %s\n", progname, strerror(errno) );
exit(EXIT_FAILURE);
}
printf( "%s: Печать содержимого именованной памяти:", progname );
for (i = 0; i < len; i++) printf("%c", ptr[i]);
printf("\n");

close(fd);
munmap(ptr, len);
}
```

14.3. Организация доступа к устройствам ввода/вывода

Взаимодействие ПРВ со своим окружением осуществляется посредством периферийных служб, которые обеспечивают транспонирование данных между внешним окружением ПРВ и базой темпоральных данных вычислительного ядра. Процессы, которые реализуются периферийными службами осуществляют взаимодействие с устройствами (модулями) ввода/вывода, обеспечивающими доступ к физическим параметрам объекта посредством датчиков (для мониторинга состояния объекта) или исполнительных механизмов (для управления объектом). Взаимодействие периферийных служб с устройствами ввода/вывода осуществляется либо посредством ассоциированных с ними специальных *ячеек* оперативной памяти с заданными физическими адресами, либо посредством специальных регистров – *портов* ввода/вывода. При этом процесс должен предварительно запросить у ядра ОСРВ и получить разрешение (привилегию) на использование системных адресов оперативной памяти или портов ввода/вывода, связанных с устройствами ввода/вывода.

Если разрешение для доступа к адресам оперативной памяти устройства ввода/вывода получено, то процесс должен выполнить вызов ядра для их отображения в адресное пространство процесса с помощью функции:

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

```
void *mmap_device_memory(void * addr, // начальный адрес
                         size_t len, // длина в байтах
                         int prot, // порядок использования
                         int flags, // порядок отображения
                         uint64_t physical // начальный физический адрес памяти
                         );
```

Функция mmap_device_memory() отображает в адресное пространство процесса блок физических адресов оперативной памяти длинной len байтов, начиная с адреса physical, и возвращает начальный адрес отображения.

Аргумент addr предназначен для указания адреса в адресном пространстве процесса, начиная с которого желательно выполнить отображение. Если в этом нет принципиальной необходимости, то следует просто задать NULL и получить адрес, сформированный ядром.

Аргумент prot предназначен для задания разрешений по использованию процессом отображаемой области памяти:

PROT_EXEC - область можно использовать для исполняемого кода.

PROT_NOCACHE – запретить кэширование содержимого памяти.

PROT_NONE - область памяти недоступна.

PROT_READ - область памяти доступна для чтения.

PROT_WRITE - область памяти доступна для записи.

Аргумент flags определяет дополнительную информацию об обработке отображённой области. Если addr был установлен в NULL, то для flags достаточно указать значение 0.

Аргумент physical задаёт начальный физический адрес оперативной памяти, отображаемой в адресное пространство процесса.

При успешном выполнении функция возвращает начальный адрес отображения в адресном пространстве процесса. Если ошибка, то - MAP_FAILED, (код ошибки помещается в errno).

Пример:

```
/*Отображение в память процесса текстовой видеопамяти режима VGA, 0xb8000*/

ptr=mmap_device_memory(NULL,len,PROT_READ|PROT_WRITE|PROT_NOCACHE,0, 0xb8000);
if(ptr==MAP_FAILED) return (EXIT_FAILURE);
//указатель ptr логически ссылается на начальный адрес блока памяти доступа к устройству
```

Если устройство ввода/вывода подключено к портам ввода/вывода и привилегия предоставлена, то для доступа процесса к портам он должен отобразить их адреса регистров в адресное пространство процесса.

Для отображения используется функция:

```
#include <sys/mman.h>
uintptr_t mmap_device_io(size_t len, uint64_t io);
```

Функция mmap_device_io() отображает заданный в io адрес регистра устройства ввода/вывода размером len байтов в адресное пространство процесса. В результате выполнения функция возвращает логический указатель регистра типа uintptr_t в адресном пространстве процесса, который можно использовать в качестве адреса порта в функциях, реализующих

вызовы ядра для доступа к регистру порта устройства ввода/вывода. В случае ошибки возвращает MAP_FAILED и устанавливает errno. При успешном выполнении функции mmap_device_io() возвращаемый результат используется в следующих функциях чтения/записи в порт.

Следующие функции читают данное из регистра порта port устройства ввода и возвращают полученное значение как беззнаковое целое, соответствующее разрядности регистра порта:

```
#include <hw/inout.h>
uint8_t in8(uintptr_t port);//Чтение 8-разрядного порта
uint16_t in16(uintptr_t port);//Чтение 16-разрядного порта
uint32_t in32(uintptr_t port); //Чтение 32-разрядного порта
```

Следующие функции записывают беззнаковое целое значение val, соответствующее размеру порта port, как в регистр устройства вывода:

```
#include <hw/inout.h>
void out8(uintptr_t port, uint8_t val);//Запись в 8-разрядный порт
void out16( uintptr_t port, uint16_t val);/*Запись в 16-разрядный порт*/
void out32( uintptr_t port, uint32_t val);/*Запись в 32-разрядный порт*/
```

15. Сигналы

Сигналы инициируются и посылаются процессу для уведомления его о том, что в системе произошло спорадическое событие, требующее асинхронной реакции процесса, изменяющей его "естественный" ход выполнения.

15.1. Механизм сигналов

Сигналы являются программным механизмом уведомления процесса о возникновении контролируемых ядром ОС системных программных или аппаратных событий, асинхронно изменяющих ход выполнения процесса. Ядро ОС контролирует определённый набор событий, каждому из которых соответствует свой сигнал.

При поступлении сигнала процессу, находящемуся в активном состоянии, процесс приостанавливается ядром, а реакция ядра на сигнал выражается в том, что текущий ход выполнения процесса асинхронно переключается на выполнение программного действия, явно или по умолчанию установленного в программном модуле процесса для данного сигнала.

В качестве реакции процесса на сигнал может быть выбрано одно из стандартных действий ядра для данного сигнала, либо можно указать ядру собственное действие в виде специальной функции, называемой *обработчиком сигнала*, которой ядро асинхронно передаст управление. После завершения реакции на сигнал, процессу восстанавливается ход выполнения с команды, прерванной сигналом [10][12][15].

В ядре ОС определён конечный набор стандартных сигналов, семантически связанных с возникновением контролируемых ядром соответствующих системных событий. Каждому сигналу соответствует уникальный целочисленный номер от 1 до 64, который представлен системной символьной константой (см. Приложение). Среди них 8 POSIX-сигналов службы реального времени, которые имеют значения в диапазоне SIGRTMIN до SIGRTMAX (описание сигналов приведено в документации к функции SignalAction()). Сигнал доставляется процессу ядром либо по инициативе самого *ядра* при возникновении системных событий, либо по инициативе *пользователя*, либо по инициативе некоторого *процесса*.

Часть сигналов инициируются процессам исключительно ядром ОС, например, процесс попытался выполнить недопустимую инструкцию (сигнал SIGILL) или осуществить допуск к адресам за пределом локальной памяти процесса (сигнал SIGSEGV). Но есть и сигналы, инициация которых вызывается действиями пользователя или процесса. Такие сигналы находятся в диапазоне от 1 до (NSIG - 1). Например, событие, связанное с нажатием пользователем на клавиатуре, связанной с процессом, клавиш <Ctrl>/<C>, вызывает посылку процессу сигнала с номером SIGINT. А, например, для принудительного завершения процесс может послать себе (т.е. инициировать посылку сигнала ядром) сигнал SIGKILL.

Пользователь может инициировать посылку сигнала процессу вручную, введя посредством клавиатуры команду shell:

kill -<системный_номер_сигнала> <PID процесса>

По инициативе самого процесса сигнал, например, инициируется вызовом kill():

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signo);
```

Аргумент pid адресует либо конкретный процесс, либо группу процессов, которым посылается сигнал. Если pid>0, то адресуется конкретный процесс. Если pid=0, то сигнал одновременно посылается самому процессу и всем процессам, входящим в группу, которой принадлежит пославший сигнал процесс. Если pid<0, то сигнал посылается каждому процессу, являющемуся членом группы процессов с GID = -pid. Аргумент sig равен 0 или определяет системный номер отправляемого сигнала. Если signo равен 0, то сигнал не посылается, а проверяется возможность послать сигнал по указанному pid (проверка наличия адресата). С помощью функции kill() процесс может послать сигнал другому процессу (в частности, самому себе) или группе процессов, если они имеют те же реальный и эффективный идентификаторы пользователя и группы, что и у посылающего сигнал процесса. Это ограничение не распространяется на процессы, обладающие привилегиями суперпользователя. Такие процессы имеют возможность отправлять сигналы любым процессам системы.

Реакция процесса на поступление сигнала, называется *действием* или *диспозицией сигнала*. Для установления процессом диспозиции сигнала могут использоваться функции signal() или sigaction(). С помощью этих функций процесс может установить одну из трёх его возможных реакций на поступление сигнала:

- действовать по умолчанию,
- игнорировать сигнал,
- асинхронно передать управление специально объявленной в процессе функции-обработчику сигнала.

Если установлено игнорирование сигнала, то это означает, что при поступлении данного сигнала он будет просто аннулирован и не окажет на процесс никакого воздействия. Существуют, однако, сигналы, которые нельзя ни игнорировать, ни блокировать, ни перехватить. Например, сигнал SIGKILL и SIGSTOP. Сигнал SIGKILL, в частности, является средством принудительного завершения процесса. Если процесс вообще не устанавливает для сигнала диспозицию или явно устанавливает диспозицию по умолчанию, то будет действовать диспозиция, реализуемая ядром по умолчанию. Действие по умолчанию определено в ядре для каждого сигнала. В большинстве случаев по умолчанию при получении процессом сигнала происходит принудительное завершение его выполнения, а для сигналов SIGCHLD, SIGIO, SIGURG и SIGWINCH в качестве действия по умолчанию предписано игнорировать сигнал. Например, игнорирование по умолчанию сигнала SIGCHLD, позволяет процессу не ждать завершения дочерних процессов, а они после завершения не превратятся в зомби.

Функция signal() имеет следующее определение:

```
#include <signal.h>
void(*signal(int signo,void(*act)(int)))(int);
```

Аргумент signo определяет сигнал, для которого устанавливается диспозиция. Аргумент act определяет диспозицию сигнала. В качестве значения act может быть присвоен указатель функции обработчика сигнала или одно из следующих системных значений:

- SIG_DFL – выполнить действие по умолчанию,
- SIG_IGN – игнорировать сигнал.

При успешном завершении signal() возвращает предыдущую диспозицию. Это может быть функция-обработчик сигнала или системные значения - SIG_DFL, SIG_IGN. Возвращаемое значение можно использовать для восстановления диспозиции.

Пример:

```
#include <signal.h>

/*Функция-обработчик сигнала*/
static void sig_hndlr(int signo){
    printf("Получен сигнал SIGINT = %i\n",signo);//Нажатие <ctrl/c>
}
int main(){
/*Установка диспозиций*/
        signal(SIGINT,sig_hndlr);//событие нажатия <Ctrl>/<C>
        signal(SIGUSR1,SIG_IGN);//игнорировать сигналы
        signal(SIGUSR2,SIG_DFL);//действовать по умолчанию
/*Бесконечный цикл*/
        while(1){
                pause();//Блокировка на неопределённое время
                puts("pause() завершена\n");//вывод текста
        }
}
```

В примере после запуска процесс блокируется на неопределённое время функцией pause(). При нажатии пользователем клавиш <Ctrl>/<C> ядро посылает процессу сигнал SIGINT. При поступлении сигнала в процессе запускается установленный для данного сигнала обработчик sig_hndlr(). Обработчик выдаёт сообщение "Получен сигнал SIGINT = 2" и завершает своё выполнение. После завершения реакции на сигнал SIGINT "естественный" ход выполнения процесса восстанавливается, процесс переходит к "команде" puts(), следующей за "командой" pause() (функция pause() завершается, возвращая –1). В результате выводиться текст "pause() завершена" и вновь в цикле выполняется pause().

Если пользователь, используя команду kill, посылает процессу сигнал SIGUSR1, то этот сигнал процесс игнорирует, приход сигнала SIGUSR1 не вызывает в процессе никаких реакций. Если же пользователь посылает процессу сигнал SIGUSR2, то выполняется действие по умолчанию (процесс терминируется).

15.2. Механизм надёжных сигналов

Использование для установки диспозиции сигнала функции signal() не во всех случаях позволяет процессу реализовать необходимые ему условия организации необходимой или предсказуемой работы с сигналами. Например, функция signal() не позволяет процессу отложить реакцию на сигнал ("замаскировать" действие сигнала) на период выполнения критического участка кода, когда прерывание ткущего выполняемого кода крайне нежелательно или недопустимо. Кроме того, нельзя предотвратить прерывания текущего выполнения ранее вызванного обработчика, для реализации вложенного вызова обработчика

сигнала, если очередной сигнал поступил в момент работы обработчика как реакции на ранее поступивший сигнал. Кроме того, нет возможности с помощью функции signal() установить выборочное влияние сигнала на конкретную нить процесса, если их в процессе более одной. В связи с этим стандарт POSIX 1003.1 определил набор функций управления сигналами, лишённый указанных недостатков. Введённая POSIX новая семантика управления сигналами (новый стиль) рассматривается как "*механизм надёжных сигналов*".

15.2.1. Набор сигналов и маска блокирования

Механизм надёжных сигналов вводит логический объект со значением 64-разрядного двоичного числа и соответствующий системный тип, называемый набором сигналов. Номера разрядов набора сигналов взаимно-однозначно соответствуют номерам сигналов, которые могут инициироваться в системе. Процесс может явно выразить в наборе сигналов те сигналы, которыми он предполагает управлять.

Процесс формирует контролируемый им набор сигналов с помощью заданной им переменной типа sigset_t. Каждый бит такой переменной ассоциируется с соответствующим сигналом (номер бита, начиная с 1 и до 64, соответствует номеру сигнала в системе). При формировании процессом контролируемого набора сигналов он устанавливает соответствующие биты в 1, а остальные сбрасывает в 0. Для сигналов, включённых в набор, процесс, как правило, явно задаёт диспозицию. Остальные сигналы либо просто не ожидаются процессом, либо процесс явно или неявно выбирает реакцию по умолчанию.

Если в старом варианте сигналы адресовались только процессу, при этом в общем случае влияние сигнала на конкретную нить процесса было неопределённым, то в новом варианте сигнал можно адресовать и конкретной нити. Такой сигнал будет оказывать влияние только на конкретную нить, когда она становится активной. Кроме того, механизм надёжных сигналов предоставляет нитям процесса возможность блокировать (задерживать) действие сигнала на нить. Для этих целей нити процесса могут задать и использовать переменную типа sigset_t, логически интерпретируемую как *маска блокирования* набора сигналов, и использовать её в запросе на их блокирование. При формировании маски блокирования сигналов нитью следует учитывать, что для блокирования сигнала соответствующий сигналу бит маски блокирования устанавливается в 1, в противном случае – сбрасывается в 0 (сигнал деблокируется). Часто в качестве маски блокирования используется переменная, содержащая ранее определённый набор сигналов.

Для формирования набора сигналов или маски блокирования сигналов процессу предоставляются следующие функции:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set,int signo);
int sigdelset(sigset_t *set,int signo);
int sigismember(const sigset_t *set,int signo);
```

Функция sigemptyset() инициализирует набор сигналов или маску блокирования, очищая все биты. Набор сигналов становится пустым, а такая маска может использоваться только для деблокирования всех системных сигналов. Функция sigfillset() формирует набор, включающий все системные сигналы, а соответствующая маска используется для блокирования всех

сигналов (кроме сигналов, которые нельзя заблокировать). Функции sigaddset()и sigdelset() позволяют добавлять и удалять сигналы набора, а также устанавливать и сбрасывать соответствующие биты маски блокирования. Функция sigismember() позволяет проверить наличие указанного сигнала в наборе или установку соответствующего бита маски блокирования.

15.2.2. Установка диспозиции сигнала

В установке диспозиций сигналов могут участвовать все нити процесса. Однако в итоге сигналу в процессе можно назначить только одну диспозицию. Это будет та диспозиция, которую сформировала нить, последней выполнившая установку диспозиции этому сигналу. Все сформированные нитями диспозиции сигналов принадлежат процессу, а не конкретной нити.

Вместо функции signal() механизм надёжных сигналов использует функцию sigaction(), позволяющую установить диспозицию сигнала, узнать текущую диспозицию или выполнить и то и другое одновременно. Функция имеет следующее определение:

```
#include <signal.h>
int sigaction( int signo, const struct sigaction *act, struct sigaction *oact );
```

Аргумент signo определяет номер сигнала. Диспозиция сигнала задаётся в структуре типа sigaction, которая передаётся через указатель act. Если текущая диспозиция не меняется, то указатель act равен NULL. Если указатель oact не NULL, то в соответствующей структуре запоминается текущая диспозиция.

Структура sigaction включает в себя следующие поля:

void (*sa_handler)(int signo); - указатель обработчика сигналов старого типа или системный тип действия.

void (*sa_sigaction)(int signo, siginfo_t *info, void *other); - указатель обработчика сигналов нового типа или системный тип действия.

sigset_t sa_mask; - маска блокирования, которая используется для блокирования сигналов, в течение времени выполнения обработчика сигнала.

int sa_flags; - специальные флаги. По умолчанию все флаги 0. Явно, можно установить флаги SA_NOCLDSTOP и SA_SIGINFO.

Установка флага SA_NOCLDSTOP предписывает ядру не генерировать родительскому процессу сигнал SIGCHLD, когда его дочерний процесс был остановлен сигналом SIGSTOP. По умолчанию ядро инициирует сигнал SIGCHLD родительскому процессу, чтобы проинформировать его об остановке дочернего процесса и предотвратить тем самым блокирование родительского процесса в состоянии ожидания завершения дочернего процесса на неопределённое время.

Флаг SA_SIGINFO управляет режимом учёта поступающих сигналов. По умолчанию (sa_flags=0) режим учёта поступления сигнала таков, что многократное инициирование процессу или нити одного и того же сигнала при условии, что сигнал заблокирован или нить находится в ожидании выделения ей процессорного времени, оставляет актуальной только одну последнюю инициацию сигнала. Все предыдущие инициации сигнала теряются. Если же установлен флаг SA_SIGINFO, то все инициации сигнала ставятся ядром в очередь и в итоге будут доставлены адресату.

Поле структуры sigaction, предназначено для задания обработчика сигналов (sa_handler или sa_sigaction). Это поле определено как union, и используется для хранения указателя функции обработчика сигнала в старом или новом стиле в зависимости от выбора пользователем типа обработчика. Если флаг SA_SIGINFO установлен (для учёта всех инициаций сигнала), то целесообразно использовать обработчик сигналов нового типа (указатель функции обработчика сигналов заносится в sa_sigaction). Если используется функция-обработчик сигналов старого типа, то указатель функции следует заносить в поле sa_handler, а режим учёта инициаций сигнала целесообразно выбрать по умолчанию (следующая инициация аннулирует предыдущую) так как старый тип функции позволяет получить только номер сигнала.

Системный тип действия для сигнала задаётся как обычно символическими константами:

- SIG_DFL - определённое для сигнала действие по умолчанию;
- SIG_IGN - игнорировать сигнал.

Функция sigaction() возвращает 0, в случае успеха, и –1, в случае ошибки и устанавливается errno. Ошибка выдаётся и в случае, если задаётся сигнал, который нельзя ни заблокировать, ни установить обработчик сигнала. Ниже рассматривается пример использования функции.

Пример:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

using namespace std;

int main(void){
extern void handler(int signo);// обработчик сигналов старого типа
struct sigaction act;
sigset_t set;

/*Формирование набора контролируемых процессом сигналов*/
sigemptyset(&set);//очистить набор сигналов
sigaddset(&set,SIGUSR1);//установить в наборе SIGUSR1
sigaddset(&set,SIGUSR2); //установить в наборе SIGUSR2

/* Обработчик для сигнала SIGUSR1 устанавливается так, что когда он запускается,
   маскируются сигналы, заданные в наборе */
act.sa_mask = set;//задать маску сигналов
act.sa_flags = 0;//учитывать только последнюю инициацию сигнала
act.sa_handler = &handler;//Используется старый тип обработчика
```

```
sigaction(SIGUSR1,&act,NULL);//диспозиция сигнала SIGUSR1
/*на сигнал SIGUSR2 - действие по умолчанию (процесс терминируется)*/

cout << "Диспозиция установлена" << endl;
kill(getpid(), SIGUSR1);//послать сигнал себе
pause();
return EXIT_SUCCESS;//До return процесс не доходит
}
//---------------------------------------------------------------
void handler(int signo){//обработчик сигнала
static int first = 1;// Флаг первого входа в обработчик

  cout << "Обработка сигнала SIGUSR1= " << signo << endl;
if(first){
  first = 0;//Сбросить флаг входа в обработчик
  kill(getpid(),SIGUSR2); /*Генерация сигнала, но сигнал автоматически замаскирован*/
  kill(getpid(),SIGUSR1); /*Генерация сигнала, но сигнал автоматически замаскирован*/
}
  cout << "Завершение обработчика сигнала.\n" << endl;
}
```

В начале в main() инициируется сигнал SIGUSR1, в результате чего вызывается обработчик сигнала и автоматически маскируются сигналы SIGUSR1 и SIGUSR2. В обработчике при первом входе в него последовательно инициируются сигналы SIGUSR2 и SIGUSR1 (т.к. first=1), однако, пока не завершится текущее выполнение обработчика сигнала, нет реакции на новые сигналы (они замаскированы и задерживаются). При завершении обработчика демаскируются сигналы SIGUSR1 и SIGUSR2. Задержанные сигналы SIGUSR1 и SIGUSR2 теперь актуализируются. Первым срабатывает сигнал SIGUSR1, его приоритет выше (так как номер меньше). Вновь запускается handler(), но сигналы в нем больше не инициируются (т.к. first=0). После очередного завершения обработчика сигналов демаскируется и срабатывает ожидающий сигнал SIGUSR2. В результате выполняется действие по умолчанию - процесс принудительно терминируется. В итоге до завершения функции main() (выполнения оператора return EXIT_SUCCESS;) процесс не доходит.

15.3. Надёжное управление сигналами

15.3.1. Посылка сигнала

Использование функции kill() для посылки процессу сигнала имеет ограниченные возможности. Во-первых, адресатом сигнала, посылаемого с помощью функции kill(), является процесс (процессы). Воздействие такого сигнала на нити процесса (если их более одной), не всегда очевидна. Во-вторых, если работа осуществляется в сети, функция предоставляет возможность адресовать сигнал процессам только местного узла. Поэтому механизм надёжных сигналов вводит для посылки сигнала функцию SignalKill(), которая расширяет возможность

управления посылкой сигнала. Сигнал можно послать и на удалённый узел, и даже адресовать его непосредственно указанной нити процесса. Функция SignalKill() имеет вид:

```
#include <sys/neutrino.h>
int SignalKill(uint32_t nd, pid_t pid, int tid, int signo, int code, int value);
```

Аргумент nd определяет дескриптор сетевого узла. Если рассматривается только местный узел, то nd следует присвоить значение ND_LOCAL_NODE (или 0). Аргумент pid задаёт значение ID процесса, которому посылается сигнал, или указывается равным 0. Аргумент tid задает значение ID нити, которой посылается сигнал, или указывается равным 0. Аргумент signo определяет системный номер посылаемого сигнала. Аргументы code и value – это ассоциируемые с сигналом некоторый код и некоторое значение (либо как целое int, либо как указатель void*), позволяющие передать вместе с сигналом дополнительную информацию. Например, если передаётся указатель на структуру, то через полученный с сигналом указатель можно получить доступ к полям структуры. Если ядро инициирует системный сигнал в связи с возникновением контролируемого ядром системного события, то аргументы code и value формируются ядром и имеют соответствующие событию системные значения. Если сигнал генерируется процессом, то значения аргументов code и value формируются процессом произвольно с учётом системных ограничений на значения code для прикладных процессов.

Функция SignalKill позволяет послать сигнал *группе процессов*, *процессу* или *нити*. Сочетание значений pid и tid определяют, кому адресуется сигнал:

pid	tid	Адресат
=0	-	Группа процессов, которой принадлежит процесс, пославший сигнал
<0	-	Группа процессов (GID = -pid)
>0	=0	Процесс (ID процесса равен pid)
>0	>0	Нить в процессе (ID нити равен tid, ID процесса равен pid)

Вызов не является блокирующим. Функция SignalKill() в случае ошибки возвращает –1, и устанавливает значение errno. Любое другое значение говорит об успешном завершении. Успешное завершение функции означает, что сигнал доставлен адресату. Если аргументу signo присваивается значение 0, то сигнал не посылается, но таким способом можно проверить актуальность адресованных в вызове процесса и нити.

15.3.2. Доставка сигнала процессу и реакция адресата

Если сигнал (который можно игнорировать) игнорируется процессом, то никакой реакции на доставленную процессу инициацию сигнала не последует, а инициация сигнала аннулируется. Если сигнал не игнорируется, но адресат (процесс или нить) замаскировал сигнал, то действие сигнала задерживается до момента сброса адресатом маски сигнала.

Воздействие инициации сигнала на процесс выражается в следующем. Во-первых, прерывается текущий "ход выполнения" процесса, он переключается на выполнение диспозиции, установленной в процессе сигналу, а во-вторых (при завершении выполнения диспозиции), асинхронно изменяет состояние *некой* нити процесса (если сигнал адресован процессу) или *конкретной* нити (которой он адресован).

Если при инициации сигнала адресатом указан только процесс, сигнал не игнорируется и не замаскирован всеми нитями, то ядро доставит его какой-то одной из нитей, у которых не

замаскирован соответствующий сигнал. В результате, если, например, все эти нити в процессе находятся в блокированном состоянии, то блокирующий вызов ядра одной из нитей завершается с возвратом сообщения о выходе из блокированного состояния по сигналу, и нить становится готовой к выполнению. Важно учитывать, что на какую конкретно нить окажет воздействие сигнал, адресованный процессу, не определено. Если некоторая и этих нитей процесса в момент прихода сигнала была *активна*, то именно она прерывается сигналом на время выполнения диспозиции. Чтобы избежать неопределённости воздействия инициаций сигналов, адресованных процессу, на нити можно придерживаться следующего правила - все нити явно маскируют все сигналы за исключением одной нити, которая и будет "ощущать на себе" инициации всех сигналов, поступающих процессу.

Каждая нить имеет возможность установить собственную маску блокирования сигналов (см. ниже). Если при посылке сигнала указан адрес нити, но нить замаскировала сигнал, адресованный процессу или этой нити, то воздействие сигнала задерживается до тех пор, пока нить не демаскирует сигнал, сбросив соответствующий бит маски блокирования.

Если сигнал адресован процессу, но все его нити установили собственную маску блокирования этого сигнала, то инициация сигнала будет задержана процессом. Нить, которая первой демаскирует сигнал, получает задержанную инициацию сигнала.

Если инициация сигнала адресована конкретной нити, то она будет доставлена только этой нити. Сигнал никогда не перенаправляется другой нити процесса.

Если сигнал адресуется группе процессов, то сигнал доставляется описанным выше способом каждому процессу в группе.

Для управления маскированием сигналов нитями используется функция:

```
#include <sys/neutrino.h>
int SignalProcmask( pid_t pid, int tid, int how, const sigset_t* set, sigset_t* oldset);
```

Функция позволяет изменить или проверить маску блокирования сигналов в направлении нити tid в процессе pid. Если pid равен 0, то рассматривается текущий процесс. Если tid равен 0, то pid игнорируется, а в качестве нити-адресата выступает сама нить, выполнившая запрос.

Аргумент set используется для изменения текущей маски блокирования. Если нет необходимости изменять текущий набор масок блокирования, то в качестве аргумента следует задать NULL.

Аргумент oldset используется для сохранения предыдущего значения маски блокирования (при изменении) или для получения текущего значения маски блокирования (когда set равен NULL). Если необходимость в аргументе отсутствует, принимает значение NULL.

Аргумент how определяет способ, которым изменяется маска блокирования сигналов. Он может принимать следующие значения:

SIG_BLOCK – итоговая маска блокирования формируется как объединение текущего значения маски и значения, на которое указывает аргумент set.

SIG_UNBLOCK - итоговая маска блокирования формируется как пересечение текущего значения маски и значения, на которое указывает аргумент set.

SIG_SETMASK - итоговая маска блокирования определяется значением, на которое указывает аргумент set.

SIG_PENDING – не изменяет маски блокирования, а позволяет получить сведение о наличии задержанных сигналов, адресованных данной нити или процессу, результат

сохраняется в переменной типа sigset_t, на которую указывает oldset. Значение set игнорируется.

Если сигнал при выполнении функции деблокируется, ядро проверяет наличие инициаций данного сигнала, доставленных нити и ожидающих обработки. Если нет доставленных нити инициаций сигналов, ждущих обработки, то ядро проверяет наличие инициаций сигналов, доставленных процессу. Если имеется задержанная процессом инициация сигнала, то она доставляется нити и немедленно действует. Если нет задержанной инициации сигнала, то никакое действие не выполняется. Невозможно блокировать сигналы SIGSTOP или SIGKILL.

Функция не является блокирующей. В случае ошибки функция возвращает -1 и устанавливает errno. Любое другое значение означает успешное завершение функции.

15.3.3. Реакция процесса на сигнал

Реакция нитей процесса на сигнал во времени определяется масками блокирования сигналов, установленных нитями. Если сигнал адресован процессу и блокируется всеми нитями процесса, то реакция процесса на него задерживается до момента снятия его блокирования любой из нитей. Если сигнал адресован нити процесса и блокируется ею, то реакция процесса на него задерживается до момента снятия ею блокирования сигнала.

Если сигнал не замаскирован, то при его поступлении реализуется диспозиция, установленная процессом. Если в качестве диспозиции установлено действие по умолчанию, то реакция процесса соответствует действию по умолчанию, заданного для данного сигнала ядром.

Если в качестве диспозиции установлено игнорирование сигнала, то приход сигнала не вызывает в поведении нитей процесса никаких изменений (исключение составляют сигналы, которые не могут быть игнорированы).

Если установлен обработчик сигнала, то в результате прихода инициации сигнала вызывается установленная процессом функция-обработчик сигнала (кроме сигналов, которые не допускают перехвата процессом, например, SIGKILL).

Прикладная функция, запускаемая в качестве обработчика сигнала, может объявляться в старом или новом стиле:

void handler(int signo) – старый стиль,
void handler(int signo, siginfo_t* info, void* other) - новый стиль.

Независимо от выбранного стиля обработчика, ядро всегда вызывает его как обработчик нового стиля. Если установлен обработчик сигналов старого стиля, то дополнительные аргументы формально передаются ему ядром (функция их просто не использует).

При вызове ядро передаёт в обработчик номер сигнала signo и структуру info типа siginfo_t. Аргумент other в настоящее время не применяется и зарезервирован на будущее.

Структура siginfo_t, содержит следующие поля:
int si_signo – номер сигнала, который равен значению аргумента signo обработчика handler.
int si_code – код сигнала, который формируется инициатором сигнала и служит в качестве дополнительной информации, например, для идентификации источника и/или причины посылки сигнала.

union sigval si_value – значение, связанное с данной инициацией сигнала, которое задаётся инициатором сигнала, оно может быть представлено либо в виде целого типа int, либо - указателя неопределённого типа данного - void*:

```
union sigval { int sival_int;
               void *sival_ptr;
};
```

Аргумент info обработчика сигнала позволяет программе получить доступ к следующим полям структуры siginfo_t:

info->si_signo – номер сигнала;

info->si_code – переданный с сигналом код;

info->si_errno – переданный системный код ошибки;

info->si_value.sival_int – если значение задано как int;

info->si_value.sival_ptr – если значение как void*.

Значение si_code является 8-разрядным целым со знаком. Важно отметить, что пользовательскими значениями могут быть значения в диапазоне -128 <= si_code <= 0. А вот значения 0 < signo <= 127 являются сугубо системными значениями, генерируемыми ядром, и НЕ МОГУТ ИСПОЛЬЗОВАТЬСЯ ПОЛЬЗОВАТЕЛЯМИ (будет выдана ошибка).

В качестве примера ниже приведены некоторые из системных значений si_code, определённых стандартом POSIX, которые используются ядром:

SI_USER - сигнал сгенерирован функцией kill().

SI_QUEUE - сигнал сгенерирован функцией sigqueue().

SI_TIMER - сигнал сгенерирован таймером[9].

SI_ASYNCIO - сигнал сгенерирован асинхронным вводом-выводом.

IOSI_MESGQ - сигнал сгенерирован очередью сообщений POSIX (не QNX).

Пока обработчик сигнала выполняется, последующие воздействия сигнала на процесс автоматически блокируются, предотвращая тем самым вложенные вызовы обработчика как реакции на инициации того же сигнала. Кроме того, при установке диспозиции сигнала можно в sa_mask при необходимости указать номера дополнительных сигналов, которые по "ИЛИ" добавляются к маске блокирования при вызове обработчика. Когда обработчик нормально завершает своё выполнение, предыдущее значение маски блокирования сигналов восстанавливается (изменения маски, сделанные в обработчике с использованием функции SignalProcmask(), теряются) и задержанные, но теперь демаскированные сигналы, начинают действовать. Однако, для возврата из обработчика сигнала можно использовать функцию longjmp(). Тогда маска не восстанавливается, и сигналы остаются замаскированными. При этом для восстановления исходной маски можно воспользоваться функцией siglongjmp(), предварительно сохранив маску с помощью функции sigsetjmp().

После завершения обработки сигнала управление передаётся прерванной нити для её продолжения. Важно учитывать, что если нить, принявшая воздействие сигнала, была Блокирован ядром, когда ей был доставлен сигнал, то блокирующий запрос завершается и возвращает EINTR (исключение составляют запросы ChannelCreate() и SyncMutexLock()).

[9]Заметим, что системные сигналы службы реального времени с кодом SI_TIMER никогда в очередь не ставятся.

Код и значение всегда доставляются с сигналом, несмотря на то, установлен или нет флаг SA_SIGINFO для сигнала signo. Если SA_SIGINFO установлен, можно использовать сигналы, чтобы без потерь доставлять небольшое количество данных. Если signo, code и value сигнала не изменяются, то ядро выполняет сжатие инициаций сигнала в очереди, изменяя 8-разрядный счётчик инициаций соответствующего сигнала, находящегося в очереди.

Приведём пример управления сигналами в рамках механизма надёжных сигналов.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <errno.h>
#include <sys/netmgr.h>

int code, value;

int main( void ){

   extern void handler1(int signo,siginfo_t *info,void *over);
   extern void handler2();

   struct sigaction act1, act2;

   sigset_t set;

   sigemptyset(&set);
   sigaddset( &set, SIGUSR1 );
   sigaddset( &set, SIGUSR2 );
   sigaddset( &set, SIGINT );//Сигнал по <ctrl/c>

   /*Определение обработчика сигнала SIGUSR1 так, что когда он доставляется, маскируются сигналы SIGUSR1 и SIGUSR2.*/

   act1.sa_flags = SA_SIGINFO;//инициации ставятся в очередь
   act1.sa_mask = set;/Маска для SIGUSR1 и SIGUSR2
   act1.sa_sigaction = &handler1;

   act2.sa_flags = 0;//оставляется последняя инициация сигнала
   act2.sa_mask = set;/Маска для SIGUSR1 и SIGUSR2
   act2.sa_handler = &handler2;
```

```
   sigaction( SIGUSR1, &act1, NULL ); //диспозиция сигнала SIGUSR1
   sigaction( SIGUSR2, &act2, NULL ); //диспозиция сигнала SIGUSR2

   code=-12; value=12;//Обратить внимание, что code отрицательное!

/*Посылаемый сигнал маскируется процессом*/
   if(SignalKill(ND_LOCAL_NODE,getpid(),0,SIGUSR1,code,value)==-1)
     perror("SignalKill: ");//Печать ошибки

/*Процесс завершится после обработки сигнала SIGUSR2*/
   puts("OK");
   return EXIT_SUCCESS;//
}
/********** Обработчики сигналов ********************/
void handler1(int signo,siginfo_t *info,void *over){
   static int first = 1;

   printf("Вошли в handler1 по сигналу %d.\n", signo);
   printf("code = %d.\n", info->si_code);
   printf("value = %d.\n", info->si_value);

   if(first){
    first = 0;
    SignalKill(ND_LOCAL_NODE,getpid(),0,SIGUSR1,code,value); /* Сигнал замаскирован */
    kill(getpid(),SIGUSR2); /* Сигнал замаскирован */
   }
   printf( "Завершение handler1.\n" );
 }
/***************/
void handler2( signo ){
   printf( "Вошли в handler2 по сигналу %d.\n", signo );
   printf( "Завершение handler2.\n" );
}
```

В данном примере определяется набор из двух сигналов SIGUSR1 и SIGUSR2 и устанавливаются два обработчика сигналов handler1() и handler2() так, что при их запуске оба сигнала маскируются. При запуске программы в main() инициируется сигнал SIGUSR1, в результате чего вызывается обработчик сигнала handler1(), а сам сигнал маскируется. При первом входе в handler1() в нём последовательно инициируются сигналы SIGUSR1 (функцией SignalKill()) и SIGUSR2 (функцией kill()), однако, пока не произошло возврата в main(), сигналы остаются замаскированными и реакция на сигналы отсутствует. Возвращение в main() демаскирует сигналы SIGUSR1 и SIGUSR2 и теперь они актуализируются. Первым

актуализируется сигнал SIGUSR1, так как его номер меньше и поэтому приоритет выше. Он приводит к повторному запуску handler1(), но сигналы в нем больше не инициируются (first равно 0). После завершения обработчика сигналов handler1() актуализируется ожидающий сигнал SIGUSR2, который доставляется процессу. В результате выполняется обработчик сигналов handler2(). Так как сигналы больше не поступают, то ни что не мешает процессу завершиться, и он завершается по return EXIT_SUCCESS.

15.3.4. Ожидание сигнала

Помимо асинхронной реакции на приход сигнала в ряде случаев нити может потребоваться явно планировать обработку сигналов, предварительно задерживая их, устанавливая маску блокирования. Для этого может использоваться функция:

```
#include <sys/neutrino.h>
int SignalWaitinfo(const sigset_t* set, siginfo_t* info);
```

При выполнении функции нить блокируется в состоянии ожидания прихода сигнала (состояние блокирования STATE_SIGWAITINFO), если в указанном наборе сигналов, на который указывает set, нет ни одного задержанного сигнала. Нить выходит из этого состояния блокирования и функция SignalWaitinfo() завершается в двух случаях. Во-первых, когда инициации одного или более сигналов из указанного набора оказываются задержанными маской блокирования. Во-вторых, когда ей в этом состоянии доставляется инициация сигнала, не блокированного маской. В первом случае функция SignalWaitinfo() извлекает задержанный сигнал из набора сигналов типа sigset_t, на который указывает set, возвращает номер извлечённого сигнала и сохраняет информацию, полученную с извлечённым сигналом, в структуре siginfo_t, на которую указывает аргумент info. Во втором случае реализуется диспозиция сигнала, после выполнения которой SignalWaitinfo() завершается с ошибкой, возвращает -1 и устанавливает в errno код ошибки EINTR.

Аргумент info может принимать значение NULL, если кроме номера сигнала другая информация с инициацией сигнала не важна или не передаётся.

16. Механизмы синхронизации нитей с реальным временем

16.1. Системное реальное время

Реальное время (РВ) рассматривается как особый процесс, периодически генерирующий события, называемые моментами времени. Синхронизация нитей с реальным временем преследует цель согласовать работу нитей с наступлением тех или иных моментов реального времени, при возникновении которых должно начаться выполнение нитью того или иного действия или, наоборот, то или иное выполняемое нитью действие должно завершиться.

Ядро включает в свой состав службу реального времени (РВ), которая обеспечивает функционирование системных часов реального времени и синхронизацию работы нитей с системным реальным временем [10][16].

16.1.1. Основные понятия

Системные часы ОСРВ — это механизм измерения течения времени с наивысшей разрешающей способностью, идущие при включённом компьютере, использующие в качестве источника периодических импульсов аппаратный таймер. Каждый отсчёт системных часов называется системным тиком (system tick) - интервал дискретности отсчёта времени (период хода, clock_period, tick size, size of timer tick) системных часов. Системные часы распространяются на всю систему и доступны всем процессам.

Системное время (system time) – время, отсчитываемое системными часами ОСРВ. Начало отсчёта системного времени - 00 часов 00 минут 00 секунд 1 января 1970 года Универсального Координированного Времени - UTC (Universal Time Coordinated). Единицей времени UTC является секунда. В операционных системах предусмотрена как ручная, так и автоматическая корректировка времени в компьютерах путём получения значений UTC по сети со специальных серверов точного времени на основе протокола NTP-Network Time Protocol, или его упрощённой (Simple) версии - SNTP. Внутренний формат представления системного времени в ОСРВ QNX позволяет выражать время в секундах (абсолютное) и/или наносекундах (относительное), имеет тип uint64_t.

Абсолютное время (absolute time) – период системного времени, отсчитываемый в *секундах* от 00 часов 00 минут 00 секунд 1 января 1970 года и рассчитано на использование до января 2554 г.

Календарное время (calendar time) – системное время в формате астрономического календаря с точностью до секунд. С помощью специальных функций абсолютное время преобразуются в обычную календарную дату и время и наоборот.

Местное время (local time) – то же, что и календарное время, но с учётом часового пояса, заданного на компьютере.

Разделённое время (broken-down time) – форма представления времени по астрономическому календарю в виде, привычном для человека – год, месяц, день и т.д. Основой является календарное время.

Таймер – создаваемый процессом программный объект, позволяющий процессу задать интервал времени, по истечении которого процесс получает от таймера уведомление заданной формы.

Относительное время (relative time) – интервал времени, который может быть задан с предельно допустимый точностью, отсчитываемый от момента запуска таймера.

Тип часов (timing base) – определяет особенности поведения системных часов и таймеров в некоторых специфичных условиях. Предусмотрены следующие типы часов:

CLOCK_REALTIME – часы непрерывно отсчитывают системное время, но могут быть подкорректированы с помощью системных вызовов ClockAdjust() и ClockTime();

CLOCK_SOFTTIME – часы, аналогичные CLOCK_REALTIME, но останавливающие отсчёт времени, когда процессор находится в «спящем режиме». Таймер, основанный на этом типе часов, не «разбудит» спящий процессор в тот момент времени, когда приложение, ждущее уведомления от таймера должно было бы разблокироваться. Если процессор не находится в спящем режиме, CLOCK_SOFTTIME идентичен CLOCK_REALTIME;

CLOCK_MONOTONIC – часы отсчитывают постоянно возрастающее время с заданной частотой, и показания этих часов не могут быть скорректированы.

Режимы CLOCK_SOFTTIME и CLOCK_MONOTONIC не реализованы в ранних версиях ОС РВ QNX/Neutrino.

Служба РВ ведёт учёт астрономического времени (абсолютное время) с точностью до секунды и позволяет отсчитывать интервалы времени (интервальное время) с точностью до тика. Величина тика определяется частотой поступления аппаратных прерываний от таймера, которые обслуживаются системной подпрограммой обработки прерываний (ISR) службы РВ. В компьютерах PC аппаратный таймер строится на базе высокочастотного аппаратного генератора синхроимпульсов. Высокочастотный меандр этого генератора делится при помощи аппаратного счётчика, который понижает частоту импульсов до 100.00684 герц. Эта частота и является частотой аппаратных прерываний таймера, которые уже могут быть обработаны ISR. В результате величина тика (период между импульсами прерываний) соответственно равна 0.0099993160 с и округляется до 10 мс. Так как реальный тик несколько отличается от 10 мс, служба РВ ядра вводит соответствующие поправки при вычислении времени.

16.1.2. Разрешающая способность РВ

Разрешающая способность (точность) системных часов службы реального времени с одной стороны определяет возможность ОС быть использованной для управления физическими процессами в требуемом темпе, а с другой - определяет эффективность использования системных часов. Служба времени ядра работает в темпе поступления прерываний аппаратного таймера, который определяет величину тика и, следовательно - предельно максимальный различимый во времени темп синхронизируемых процессов. Внутри тика "течение времени" не различимо. Тик — это "протяжённость" текущего "момента времени". Если в рамках одного тика могут произойти более одной реализации одного и того же контролируемого события, то все они будут помечены одним и тем же значением момента времени и, следовательно, во времени не различимы. Это означает, что разрешающая способность используемых системных часов не достаточна и точность часов надо повышать.

С разрешающей способностью системных часов связано и такое понятие, как флуктуация отсчёта времени. Темп работы ядра ОС значительно превышает точность системных часов, поэтому в общем случае ядро фиксирует запросы активных нитей на планирование времени на протяжении всего текущего тика (когда системные часы "стоят"). Это и приводит к флуктуации

отсчёта времени. Возникновение флуктуации отсчёта времени объясняется на **Ошибка! И сточник ссылки не найден.**

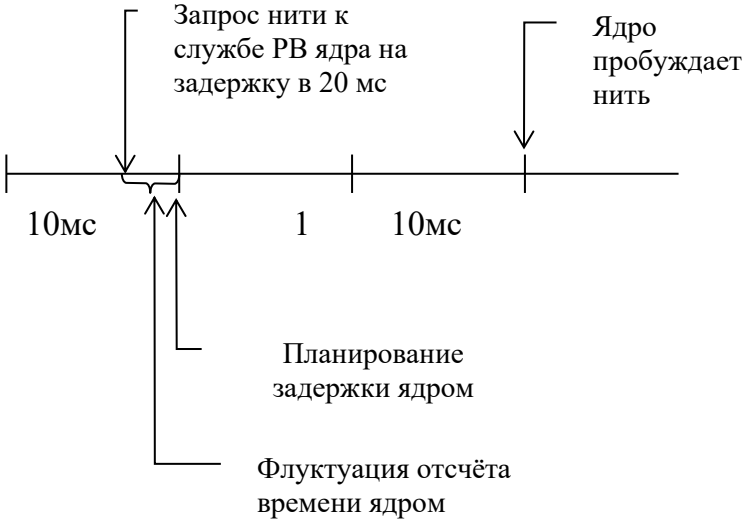


Рис. 2 Флуктуации отсчёта времени

Из рисунка видно, что если нить планирует задержку в 20мс, то в итоге реальный интервал задержки будет лежать в диапазоне от 20 до 30мс - в зависимости от того, насколько близко или далеко от очередного отсчёта времени (прерывания аппаратного таймера) ядро получает запрос от нити. Заметим, что чем ближе темп управляемого физического процесса к темпу системных часов, тем существеннее влияние флуктуации на эффективность управления, что может потребовать увеличения точности системных часов. Однако необходимо при этом учитывать, что избыточная точность системных часов приводит к неоправданным затратам процессорного времени на обработку прерываний службы времени, что снижает эффективность ядра ОС по управлению запросами от нитей прикладных процессов.

16.1.3. Установка значений абсолютного и относительного времени

Рассмотрим порядок задания значений абсолютного или относительного (интервального) времени. Для задания значения как абсолютного, так и интервального времени используется одна и та же системная структура данных timespec (определена в time.h):

```
struct timespec{time_t tv_sec; //секунды
                uint64_t tv_nsec; //наносекунды, 1сек=10^9нс
                };
```

Структура timespec задаёт время в секундах и долях секунды, которую принято выражать в наносекундах. Если время в структуре timespec задаётся как абсолютное, то указанное количество секунд рассматривается как значение интервала времени, отделяющего устанавливаемый момент абсолютного времени от базового момента, определённого как 00 час 00 мин 00 с, 1 января 1970 года по Гринвичу (01.01.1970 00:00:00).

Рассмотрим следующий пример задания системного времени:

```
struct timespec it_value;
…
it_value.tv_sec = 987654321;
```

```
it_value.tv_nsec = 0;
…
```
Заданная комбинация параметров системного времени в контексте абсолютного времени соответствует моменту календарного времени - 19.04.2001 04:25:21. Заметим, что в контексте относительного времени это бы соответствовало интервалу в 31.3 года, что очевидно не имеет практического значения.

В качестве примера задания интервала времени рассмотрим следующее значение:

```
struct timespec it_value;
…
it_value.tv_sec = 5;
it_value. tv_nsec = 500 000 000;
…
```
Очевидно, что такое значение предполагает его интерпретацию как интервала времени в 5,5 секунды. А в контексте абсолютного времени это значение соответствовало бы моменту календарного времени в прошлом 01.01.1970 00:00:05, что для синхронизации нитей в абсолютном времени не имеет практического значения.

Существует набор системных стандартных функций, которые помогают преобразовывать время, заданное в системном формате времени, в формат календарного времени с точностью до секунд, удобный для восприятия человеком. И наоборот, для задания времени в системных вызовах, например, для планирования временных интервалов, может потребоваться переводить интервал календарного времени в системный формат. Это функции time(), localtime(), mktime(), ctime(), asctime() и др.

Для преобразования текущего момента времени, заданного в формате календарного времени, в количество прошедших секунд абсолютного времени используется функция:

```
#include <time.h>
time_t time(time_t * tloc);
```

Функция time() возвращает текущее значение местного календарного времени (UTC), установленного в операционной системе компьютера, выраженное в количестве секунд абсолютного времени. Если tloc не NULL, то текущее абсолютное время в секундах сохраняется также в объекте, на который указывает tloc.

Заданное абсолютное время в секундах можно преобразовать в формат местного календарного времени с помощью функции

```
#include <time.h>
struct tm *localtime(const time_t * t_sec);
```

Функция localtime() преобразует значение абсолютного времени в секундах в значение местного календарного времени, и возвращает указатель на структуру:

```
#include <time.h>

struct tm {
        int tm_sec;//        секунды [0,61];
        int tm_min;//        минуты [0,59];
        int tm_hour;//       часы [0,23];
```

```
int tm_mday;//      число [1,31];
int tm_mon;//       месяц с января [0,11];
int tm_year;//      годы с 1900;
int tm_wday;//      дни с воскресенья [0,6];
int tm_yday;//      дни с 1 января [0,365];
int tm_isdst;//     флажок летнего времени;
long int tm_gmtoff;//       смещение от времени по UTC (по Гринвичу);
const char * tm_zone;//     название часового пояса.
};
```

Если необходимо наоборот, имея значение абсолютного времени в формате местного календарного времени, представленного в структуре struct tm, преобразовать его в количество секунд абсолютного времени, следует воспользоваться функцией:

```
#include <time.h>
time_t mktime(struct tm* timeptr);
```

Функция преобразовывает местное календарное время в формате структуры struct tm в формат time_t – количество секунд абсолютного времени.

Если необходимо получить значение календарного времени, представленное в виде строки, то следует использовать функции:

```
#include <time.h>
char* ctime( const time_t* timer );
char* asctime( const struct tm* timeptr );
```

Функции преобразовывают значение абсолютного времени, заданное в формате time_t или struct tm, в символьную строку в виде - Tue May 7 10:40:27 2002\n\0.

16.2. Таймеры

Для согласования своей работы во времени процессы создают объекты типа timer_t, называемые таймерами, с помощью которых нити планируют получение уведомлений о наступлении абсолютного момента календарного времени или истечении интервала относительного времени. Процессы могут создавать для своих нужд произвольное количество различных таймеров. Дальнейшая работа с таймером нитей процесса заключается в том, что нити планируют таймеру посылку уведомлений, информирующих нить о наступлении во времени запланированных ими событий, позволяющих нитям синхронизироваться как с наступлением заданных абсолютных моментов времени, так и с моментами истечения заданных интервалов времени.

16.2.1. Создание и удаление таймеров

Таймер создаётся как объект типа timer_t с помощью функции

```
#include <time.h>
#include <sys/siginfo.h>
int timer_create(clockid_t clock_id, struct sigevent *event, timer_t *timerid);
```

Созданный таймер предоставляется для использования посредством указателя timerid, являющегося третьим аргументом функции. Первый аргумент clock_id сообщает функции timer_create() какой тип часов реального времени должен быть выбран: CLOCK_REALTIME, CLOCK_SOFTTIME, CLOCK_MONOTONIC.

Второй аргумент event представляет собой указатель на объект типа struct sigevent, определяющий тип уведомления нити о наступлении заданного момента времени.

Когда необходимость в таймере пропадает, его для освобождения системных ресурсов можно удалить с помощью функции

```
#include <time.h>
int timer_delete(timer_t timerid);
```

16.2.2. Типы уведомлений нитей

При создании таймера указывается тип уведомления, которое посылается ожидающей нити при наступлении запланированного временн'ого события. Существуют следующие типы уведомлений нитей:

- "послать импульс";
- "послать сигнал";
- "создать нить".

Второй аргумент функции timer_create() указывает структуру event типа struct sigevent, значение которой задаёт тип уведомления и его параметры. Структура sigevent включает в себя поля:

- sigev_notify,
- sigev_signo,
- sigev_coid,
- sigev_priority,
- sigev_code,
- sigev_value.

Значения этих полей устанавливается в зависимости от выбранного типа уведомления.

Файл <sys/siginfo.h> определяет системные макрокоманды, использование которых позволяет упростить формирование полей структуры event в соответствии с выбранным типом уведомления. Все макрокоманды используют в качестве первого аргумента - event, указатель на struct sigevent. Рассмотрим далее, как формируются значения полей в зависимости от выбранного типа уведомления.

16.2.3. Уведомление типа "послать импульс"

Импульс — это отправляемое процессу-серверу специальное сообщение системного типа:

```
struct _pulse {_uint16          type;
               _uint16          subtype;
               _int8            code;
               _uint8           zero[3];
               union sigval     value;
               _int32           scoid;
               };
```

Элементы type и subtype для импульса равны нулю (признак импульса). Содержимое элементов code и value задаются отправителем. Обычно code указывает причину, по которой был отправлен импульс, а value содержит 32 бита данных, посылаемых с импульсом (т.е. всего 40 бит). Код может быть любым 8-битным значением меньшим ноля (-127÷-1), чтобы избежать конфликта с ядром или менеджерами QNX, генерирующими импульсы. То есть, ядро предоставляет для программистов 127 отрицательных значений code для использования по своему усмотрению. Все безопасные системные значения кодов начинаются с _PULSE_CODE_ и определены в <sys/neutrino.h>. Они заключены в диапазоне от _PULSE_CODE_MINAVAIL до _PULSE_CODE_MAXAVAIL. Элемент value имеет тип объединения вида:

```
union sigval { int  sival_int;
               void *sival_ptr;
             };
```

Посылка процессом-клиентом импульса и его приём процессом-сервером имеет существенные особенности. Посылка импульса не блокирует процесса-клиента. Приём импульса выполняется как приём обычного сообщения. Отличие только в том, что функция MsgReceive() возвращает ноль (признак прихода импульса) и не требуется посылать ответ, используя функцию MsgReply(). С импульсом можно передать только 40 бит полезной информации (8-битный код и 32 бита данных). Если требуется принимать только импульсы, оставляя без внимания все другие сообщения, то в этом случае серверной нити необходимо использовать функцию MsgReceivePulse():

Чтобы создаваемый с помощью функции timer_create() таймер настроить на посылку уведомлений-импульсов, необходимо установить полю sigev_notify структуры event значение SIGEV_PULSE и дополнительно задать значения ряду соответствующих полей. Сформировать значения полей структуры event удобно с помощью макрокоманды

```
SIGEV_PULSE_INIT(struct sigevent *event,
                 int coid,
                 short priority,
                 short code,
                 union sigval value)
```

где:

int coid - ID соединения (связи) с каналом, по которому уведомляющий импульс будет посылаться. Если процесс планирует для себя уведомление импульсом, то ему необходимо создать канал и ID соединения с собственным каналом использовать в качестве coid в макрокоманде.

short priority – приоритет, связываемый с импульсом, который будет наследоваться нитью, принявшей импульс. Нулевое значение не допускается. Если нет необходимости в изменении приоритета принимающей импульс нити, то для priority следует установить специальное значение SIGEV_PULSE_PRIO_INHERIT.

short code – код импульса.

union sigval value - 32-битное значение импульса (типа int или void*).

Рассмотрим, в каких случаях в качестве уведомления целесообразно использовать импульс. Предположим, что разрабатывается сервер, который большую часть времени проводит в RECEIVE-блокированном состоянии, ожидая прихода сообщения по каналу. При

этом он планирует приход и приём уведомлений от таймера. В этом случае логично в качестве уведомления сервера таймером использовать импульс, который должен поступать в этот канал. При этом сервер должен предварительно создать собственное соединение с собственным каналом, ID которого будет указано в качестве параметра при формировании уведомления импульсом.

16.2.4. Уведомление типа "послать сигнал"

В этом случае нить должна установить обработчик сигналов (асинхронный приём сигналов) или ожидать прихода сигнала, используя функцию SignalWaitinfo() или sigwait(). Чтобы таймер настроить на посылку сигналов, необходимо установить полю sigev_notify структуры event одно из значений SIGEV_SIGNAL, SIGEV_SIGNAL_CODE или SIGEV_SIGNAL_THREAD. Использование первых двух значений планирует уведомление сигналом, адресуемым процессу. Использование третьего значения позволяет запланировать посылку сигнала, адресуемого конкретной нити.

В первом случае посылается просто сигнал без какой-либо дополнительной информации, адресуемый процессу. Для настройки уведомления event используется макрокоманда

SIGEV_SIGNAL_INIT(struct sigevent *event, int signo),

где int signo - номер сигнала, который должен быть в диапазоне от 1 до NSIG-1.

Если при посылке сигнала необходимо передать дополнительную информацию в виде значений sigev_code и sigev_value, то для формирования уведомления надо использовать макрокоманду:

SIGEV_SIGNAL_CODE_INIT(struct sigevent *event, int signo, void *value, short code)

где дополнительно:

void *value - 32-битное значение, предназначенное обработчику сигнала.

short code - код, который должен быть в диапазоне от SI_MINAVAIL до SI_MAXAVAIL и предназначен для интерпретации обработчиком сигнала.

Если сигнал с дополнительной информацией необходимо направлять конкретной нити, то для формирования уведомления используется макрокоманда:

SIGEV_SIGNAL_THREAD_INIT(struct sigevent *event, int signo, void *value, short code)

В этом случае сигнал доставляется той нити, которая его запланировала с помощью функции timer_settime().

16.2.5. Уведомление типа "создать нить"

Этот тип уведомления предполагает, что в результате срабатывания таймера в процессе создаётся новая нить. Для задания уведомления полю sigev_notify структуры event необходимо установить значение SIGEV_THREAD. Это удобно сделать с помощью макрокоманды

```
SIGEV_THREAD_INIT(struct sigevent  *event,
                  void             (*fn)(void * arg)
                  void             *arg,
                  pthread_attr     *attributes)
```

где:

void (*fn)(void * arg) - указатель на функцию, которую нужно запустить как нить.

pthread_attr *attributes - указатель на атрибутную запись нити, он должен указывать на структуру, которая будет использована функцией pthread_attr_init(), или быть NULL для значений атрибутов по умолчанию.

void *arg - значение, которое передаётся функции, запускаемой как нить.

Замечание. Если таймер будет срабатывать слишком часто, и при этом будут готовы к выполнению нити с более высоким приоритетом, задерживая их выполнение, то в системе быстро вырастет очередь готовых нитей. В результате высока вероятность, что они исчерпают все системные ресурсы ядра. Поэтому этот тип уведомления не стоит использовать без особой необходимости.

16.2.6. Планирование срабатывания таймеров

По способу планирования срабатывания и ожидания процессами уведомлений от таймера о наступлении временн'ых событий различают таймеры *абсолютные* и *относительные*. *Абсолютный таймер* позволяет нити планировать и получать уведомление о событии наступления момента календарного времени с точностью до секунды. *Относительный таймер* позволяет нити планировать и получать уведомление о событии истечения запланированного интервала реального времени с точностью до долей секунды. По количеству планируемых процессами срабатываний таймеры делят на *однократные* и *периодические*.

Однократный таймер — это таймер, который посылает уведомление только один раз, когда наступает запланированное нитью событие абсолютного или относительного реального времени.

Периодический таймер — это таймер, которому указываются интервалы относительного времени, и он каждый раз посылает процессу уведомление по истечении текущего интервала времени и автоматически планирует очередной временной интервал. В связи с этим различают три типа настройки таймеров:

- абсолютный однократный;
- относительный однократный;
- относительный периодический.

Тип настройки таймера задаётся при его запуске процессом. Запуск таймера и указание его типа (путём комбинирования значений аргументов) осуществляется процессом с помощью функции

```
#include <time.h>
int timer_settime (timer_t          timerid,
                   int              flags,
                   struct itimerspec *alarm,
                   struct itimerspec *oldalarm);
```

Аргумента timerid является дескриптором запускаемого таймера, созданного функцией Timer_Create(). С помощью аргумента flags таймер определяется как абсолютный (указывается

значение системной константы TIMER_ABSTIME) или относительный (указывается значение отличное от TIMER_ABSTIME, например - 0).

Если таймер относительный, то в аргументе alarm задаётся интервал относительного системного времени, по истечении которого происходит срабатывание таймера как *однократного* или *периодического*. В аргументе oldalarm отличном от NULL возвращается значение ранее запланированного интервала времени, иначе – значение игнорируется.

Значение абсолютного или интервалов относительного времени задаётся как структура itimerspec, которая имеет вид:

```
struct itimerspec{
    struct timespec it_value; /* момент абсолютного или интервал относительного системного
                                 времени первого срабатывания таймера */
    struct timespec it_interval; /* интервал относительного системного времени последующих
                                    циклических срабатываний таймера*/
};
```

Абсолютный таймер всегда однократный. Он срабатывает, посылая уведомление один раз, как только текущее значение абсолютного времени окажется не меньше значения, указанного в alarm.it_value. Значение alarm.it_interval для абсолютного таймера игнорируется.

Чтобы запланировать относительный интервал однократного срабатывания таймера, необходимо отличное от нуля значение планируемого периода задать в поле alarm.it_value, а значение поля alarm.it_interval задать равным нулю. По истечении запланированного периода однократный таймер один раз посылает уведомление, как только истекший с момента планирования интервал относительного системного времени окажется не меньше значения, указанного в alarm.it_value.

Чтобы получить относительный периодический таймер, необходимо, чтобы значения времени в alarm.it_value и alarm.it_interval были отличны от нуля. Таймер первый раз пошлёт уведомление, как только истекший интервал реального времени окажется не меньше значения, указанного в alarm.it_value, и с этого момента будет периодически посылать уведомления, как только истекший с момента предыдущего уведомления интервал реального времени окажется не меньше значения, указанного в alarm.it_interval.

Если необходимо отменить действие ранее запланированного таймера, следует успеть повторно выполнить функцию timer_settime() со значением времени в alarm.it_value равным нулю.

Рассмотрим пример создания и планирования процессом-сервером периодического таймера, уведомляющего его импульсом по истечении интервала времени, равного одной секунде.

```
/*
 * Пример сервера, получающего периодические импульсы от
 *  таймера и сообщения от клиента.
 *
*/
#include <cstdlib>
#include <iostream>
```

```
#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>
#include <pthread.h>

using namespace std;

// сообщения клиентов
#define MT_DATA1    2 // прикладное сообщение1 от клиента
#define MT_DATA2    3 // прикладное сообщение2 от клиента

// импульс
#define CODE_TIMER 1 // сообщение-импульс от таймера

// формат сообщения клиентской нити
struct ClientMessageT{
      int messageType; // тип сообщения клиента
      int messageData; // данное, соответствующее типу сообщения
};

union MessageT{
      ClientMessageT  message;   // сообщение от клиента
      _pulse   M_pulse; // импульс от таймера
} msg; //буфер приёма собщений от клиента или импульсов уведомлений от таймера;
int chid; // ID канала
int       coid; // ID соединения с каналом

//прототипы функций
static  void gotAPulse (void);//обработка импульса
static  void gotAMessage (int rcvid, ClientMessageT *msg);//обработка сообщения от нити
static void* thread_func(void*); //функция для запуска нити

/*
Программа создаёт таймер, посылающий в канал импульс с кодом CODE_TIMERЮ,
и собственную клиентскую нить, посылающую в тот же канал обычные сообщения*/
/************************ MAIN ****************************/
```

```
int main(void){

if ((chid = ChannelCreate (0)) == -1){
   perror ("не могу создать канал!\n");
   exit (EXIT_FAILURE);
}
// установить соединение с собственным каналом
   coid = ConnectAttach (0, 0, chid, 0, 0);
   if (coid == -1) {
      fprintf (stderr, "%s:  ошибка соединения!\n");
      perror ("не могу создать соединение с каналом!\n");
      exit (EXIT_FAILURE);
   }

         //запустить нить
         if((pthread_create(NULL,NULL,&thread_func,NULL))!=EOK){
              perror ("не могу запустить нить!\n");
              exit (EXIT_FAILURE);

         };

// определить для таймера тип уведомления
struct sigevent event;// уведомление
// "послать импульс"
   SIGEV_PULSE_INIT(&event,coid,SIGEV_PULSE_PRIO_INHERIT,CODE_TIMER,0);

   // создать таймер
timer_t  timerid; // ID таймера
   if(timer_create (CLOCK_REALTIME,&event,&timerid)==-1){
      perror ("не могу создать таймер!\n");
      exit (EXIT_FAILURE);
   }
   // запустить периодический таймер
struct itimerspec       timer;  // системный формат времени
         timer.it_value.tv_sec = 0;
         timer.it_value.tv_nsec = 500000000;//первый импульс через 0.5сек
         timer.it_interval.tv_sec = 2;//последующие каждые 2сек
         timer.it_interval.tv_nsec = 0;

   timer_settime (timerid, 0, &timer, NULL); // запуск относительного периодического таймера!
   for (;;) { // приём сообщений
```

```
        cout << "жду сообщение от клиента или импульс от таймера!" << endl;
        int rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
      // что за сообщение получено?
      if (rcvid == 0) {//получен импульс от таймера
        gotAPulse();
      } else {//получено сообщение клиента
        gotAMessage(rcvid, &msg.message);
      }
   }
   //сюда никогда не попадём, в цикле ждём приход сообщений в канал
   return (EXIT_SUCCESS);
}

/*********************************************
 *  Функция thread_func - для запуска нити
 *********************************************
*/
static void* thread_func(void*){
 MessageT msg;
 msg.message.messageType=MT_DATA2;
        if (coid == -1) {
           perror ("ошибка соединения нити с каналом!\n");
           exit (EXIT_FAILURE);
        }
        for(int i=0;i<3;i++){
                MsgSend(coid, &msg, sizeof(msg), NULL,0);
                sleep(1);
                msg.message.messageType=MT_DATA1;
        }
return NULL;
}
/*********************************************
 *  Функция gotAPulse
*  Выполняется, когда получен импульс от таймера
 *********************************************
*/
void
gotAPulse (void){
time_t  now;
time(&now);//Получаем текущий момент времени
//Печать строки времени
```

```
        cout << "получен импульс " << ctime(&now) << endl;
        cout << "код CODE_TIMER = " << (int)msg.M_pulse.code <<endl;
        cout << "данное: " << msg.M_pulse.value.sival_int << endl;

}
/*********************************************
 * Функция gotAMessage
 * Вызывается, когда приходит сообщение от клиента
 *********************************************/
void
gotAMessage (int rcvid, ClientMessageT *msg){
// Определить тип сообщения
        switch (msg->messageType){
                case  MT_DATA1:
                        cout << "получено сообщение типа MT_DATA1" << endl;
                        MsgReply (rcvid,EOK, NULL,0);
                        return;
                case  MT_DATA2:
                        cout << "получено сообщение типа MT_DATA2" << endl;
                        MsgReply (rcvid, EOK, NULL,0);
                        return;
                default:
                        cout << "получено сообщение неопределённогог типа" << endl;
                        MsgReply (rcvid, EOK, NULL,0);
        }
}
```

16.3. Таймауты ядра

Обращаясь к ядру с запросом, нить может вынужденно оказаться в заблокированном состоянии, например при попытке захвата мутекса. Однако, в не всегда у нити может быть возможность или принципиальная необходимость быть в заблокированном состоянии излишне долго. Для заблокированной нити может оказаться более эффективным временно отказаться от исполнения запроса и продолжить своё выполнение. Для таких случаев ядро предоставляет нитям возможность использования механизма планирования времени нахождения в заблокированном состоянии. Этот механизм называется таймаутом ядра. Ядро позволяет нитям перед выполнением любого блокирующего запроса устанавливать таймаут, по истечении которого ядро выводит нить из блокированного состояния и отправляет ей уведомление специального типа SIGEV_UNBLOCK.

Для формирования таймаута нить, перед тем, как обратиться с блокирующим запросом к ядру, должна выполнить функцию TimerTimeout():

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

```
int TimerTimeout( clockid_t              clock_id,
                  int                    flags,
                  const struct sigevent  *event,
                  const uint64_t         *timout, //в наносекундах
                  uint64_t               *oldtimout ); //в наносекундах
```

При успешном выполнении функция возвращает 0. Если ошибка, то -1, код ошибки помещается в errno.

Аргумент clock_id задаёт выбранный тип часов реального времени (обычно CLOCK_REALTIME или другой имеющийся в системе тип часов).

Аргумент flags специфицирует соответствующее запросу к ядру блокирующее состояние.

В аргументе event нужно задать тип уведомления - SIGEV_UNBLOCK. Для задания уведомления этого типа удобно использовать макрос:

SIGEV_UNBLOCK_INIT(struct sigevent *event).

Но можно также просто присвоить event значение NULL. Функция рассматривает это как установку уведомления типа SIGEV_UNBLOCK.

Аргумент timout указывает на относительное время в наносекундах (1 с = 10^9 нс), спустя которое ядро должно послать нити уведомление об истёкшем таймауте и вывести нить из блокированного состояния. Если указатель timout установить равным NULL, то блокирование вообще не допустимо. Это равносильно осторожной попытке выполнить блокирующий запрос.

Аргумент oldtimout сохраняет предыдущее значение таймаута, но если в этом нет необходимости, то присваивается NULL.

Значение флага, которое необходимо задать в аргументе flags, должно соответствовать блокирующему запросу. Полный список значений флага для всех блокирующих запросов приведён в справочной системе QNX в описании функции TimerTimeout().

В качестве примера рассмотрим некоторые из возможных значений флага flags:

- _NTO_TIMOUT_JOIN – установка таймаута при использовании блокирующей функции pthread_join();
- _NTO_TIMOUT_SEND – установка таймаута для SEND-блокированного состояния при использовании блокирующей функции MsgSend();
- _NTO_TIMOUT_REPLY – установка таймаута для REPLY-блокированного состояния при использовании блокирующей функции MsgSend().

Замечание. Сбрасывать таймаут после любого варианта завершения блокирующего запроса с предварительно установленным таймаутом не надо – это выполняется автоматически.

Ниже, в качестве примера, рассматривается использование таймаута для блокирующего запроса pthread_join(). Приводится определение прикладной функции pthread_join_nb(), осуществляющей осторожное присоединение к заданной нити с нулевым таймаутом блокировки, чтобы убедиться, что она терминирована. Функция использует блокирующий запрос pthread_join(), но предварительно устанавливает таймаут ядра с нулевым значением и флагом _NTO_TIMOUT_JOIN:

```
int pthread_join_nb(int tid,void **rval){
```

```
    TimerTimeout(CLOCK_REALTIME,_NTO_TIMOUT_JOIN,NULL,NULL,NULL);
return(pthread_join(tid,rval));
}
```

Функция pthread_join_nb() проверяет, терминирована или нет нить с дескриптором tid. Если нет, то вызов pthread_join(tid,rval) возвращает полученное уведомление ядра о досрочном деблокировании. Если да, то возвращается EOK. Действие таймаута после любого варианта завершения блокирующего запроса pthread_join() автоматически аннулируется.

16.4. Использование таймаутов ядра при посылке сообщения

При посылке сообщения, выполнив, например, функцию MsgSend(), процесс-клиент оказывается либо в SEND-блокированном, либо в REPLY-блокированном состоянии. Поэтому, при планировании таймаута для выхода из блокирующего запроса передачи сообщения, необходимо в аргументе flags предусмотреть оба блокирующих состояния, сформировав его значение, используя операцию поразрядного логического ИЛИ, в виде (_NTO_TIMOUT_SEND | _NTO_TIMOUT_REPLY). Это вызовет установку ядром таймаута, когда ядро переведёт клиента в SEND-блокированное, а затем и в REPLY-блокированное состояние. Если таймаут истекает в SEND-блокированном состоянии, то функция MsgSend() завершается, возвращая клиенту признак ETIMEDOUT. Так как сервер ещё не выполнил функцию MsgReceive(), то он практически не замечает, что клиентом осуществлялась попытка послать сообщение.

Если сервер выполнил MsgReceive() и принял сообщение, то возможность нахождения клиента в REPLY-блокированном состоянии зависит от свойства канала сервера, т.е. установил ли сервер флаг _NTO_CHF_UNBLOCK при создании канала или нет. Если флаг _NTO_CHF_UNBLOCK не установлен, то клиент при истечении таймаута для REPLY-блокированного состояния будет немедленно разблокирован. При этом сервер не получит об этом никакого оповещения и будет планировать выполнение функции MsgReply(), хотя клиент уже не ждёт ответа и ответ будет выполнен впустую. Если при создании сервером канала флаг _NTO_CHF_UNBLOCK был установлен, то при истечении таймаута нахождения клиента в REPLY-блокированном состоянии клиент продолжит оставаться заблокированным, пока сервер ему не ответит, а ядро посылает серверу в канал уведомление в виде импульса, информируя его об истечении таймаута ожидания клиентом ответа от сервера. Приняв импульс, сервер берёт на себя ответственность за дальнейшую задержку ответа клиенту в REPLY-блокированном состоянии.

17. Программирование нитей обработки прерываний

В процессе функционирования ПРВ посредством служб обмена данными с физическим объектом осуществляет взаимодействие в режиме реального времени с различными внешними устройствами, которые по отношению к ПРВ выступают в роли источников исходных и/или приёмников результатных данных от ПРВ. В качестве таких внешних устройств выступают различные аппаратные устройства (контроллеры), обеспечивающие взаимодействие ПРВ с датчиками или исполнительными механизмами на объекте управления. Одним из способов организации такого взаимодействия являются аппаратные прерывания. Аппаратные прерывания являются эффективным механизмом взаимодействия с устройствами. В отличие от режима циклического опроса состояния готовности устройства к обмену данными, занимающего вычислительные ресурсы системы, режим прерываний позволяет ПРВ асинхронно переключиться на взаимодействие с устройством, только когда устройство сигнализирует о готовности к обмену данными.

Программирование аппаратных прерываний существенно зависит от особенностей подключения контроллеров внешних устройств к используемой вычислительной системе. Операционная система QNX максимально скрывает эти особенности, предоставляя программисту эффективные высокоуровневые средства для управления прерываниями и организации обмена данными с внешними устройствами [10][17].

17.1. Механизм аппаратного прерывания

Организация взаимодействия ПРВ с внешними устройствами предполагает распределение между ними ответственности за инициацию взаимодействия. Та сторона, которая назначается ответственной за инициацию взаимодействия, считается *активной*. Противоположная сторона – *пассивной*. Если активной стороной назначается ПРВ, то ответственность за выявление готовности пассивного устройства предоставить или принять данное возлагается на ПРВ. В этом случае ПРВ вынуждено с определённой частотой опрашивать флаг готовности в котроллере пассивного устройства (статусный регистр), например, периодически планируя запуск соответствующей нити через интервал времени Δt. Если активной стороной назначается устройство, то ответственность за информирование пассивного ПРВ о готовности устройства предоставить или принять данное возлагается на устройство. В отличие от активного внешнего устройства пассивное ПРВ продолжает выполнять свою текущую работу, ожидая уведомления о готовности внешнего устройства к обмену данными. Поэтому по готовности внешнего устройства к взаимодействию контроллер посылает сигнал процессору, чтобы временно переключить процессор с текущих вычислений на выполнение процедуры взаимодействия с внешним устройством. Для этого контроллер устройства использует механизм аппаратного прерывания процессора. Этот механизм предоставляет контроллеру внешнего устройства возможность переключить внимание ПРВ на себя.

Механизм аппаратного прерывания встроен в процессор компьютера. Он заставляет процессор асинхронно переключиться по сигналу, выставленному контроллером устройства на линии прерывания процессора, на взаимодействие с устройством. Появление сигнала на линии прерывания заставляет процессор прервать выполнение текущего программного кода и переключиться на выполнение программного кода по адресу (принято говорить "по номеру"),

называемому "вектором прерывания", находящемуся в начальных адресах ячеек оперативной памяти, в так называемой системной области векторов прерывания. В области векторов прерывания каждому устройству соответствует некоторая ячейка, в которых ПРВ сохраняет вектора прерываний. Обращение к программному коду по адресу вектора прерывания рассматривается как вызов специальной функции, называемой *обработчиком прерывания*. Для выхода из обработчика прерывания и возврата к прерванному программному коду ПРВ используется специальная процессорная команда завершения прерывания, которая возвращает ПРВ к коду, следующему за точкой прерывания.

Механизм прерывания может быть инициирован не только по сигналу от контроллера внешнего устройства, но и программно посредством выполнения специальной процессорной команды передачи управления, которая в качестве аргумента использует номер вектора прерывания.

Рассмотрим подробнее порядок инициации аппаратного прерывания внешним устройством. Инициация прерывания осуществляется контроллером внешнего устройства, подключённого к шине, путём выставления на общей шине сигнала на специальной линии прерывания процессора (запрос прерывания). При этом прерывание становится возможным только при условии, что процессор переведён в состояние готовности принимать запросы прерывания (установлен соответствующий разряд регистра словосостояния процессора – *прерывания деблокированы*) и завершил выполнение текущей команды. До этого момента на запросы прерывания процессор не реагирует. Если процессор запускает механизм прерывания, то он:

- получает от контроллера соответствующий устройству номер вектора прерывания;
- сохраняет в стеке текущее содержимое регистров процессора и блокирует прерывания (предотвращая вложенные прерывания);
- в соответствующие регистры процессора загружается новое содержимое из вектора прерывания указанного номера (в частности, устанавливается новое значение регистра словосостояния процессора, что может привести к деблокированию прерываний);
- управление передаётся по адресу, которой задан в векторе прерывания, начинает выполняться подпрограмма обработки прерывания (Interrupt Service Routine – ISR);
- при завершении выполнения и выходе из обработчика прерываний ранее сохранённые в стеке значения регистров процессора восстанавливаются, что приводит к восстановлению выполнения прерванного программного кода (с команды, непосредственно следующей за командой, после выполнения которой управление асинхронно было передано обработчику прерываний), а запросы прерывания деблокируются.

Так как линия запроса прерывания у процессора одна, а внешних устройств в общем случае больше одного, то подключение к ней устройств и упорядочение (арбитраж) их запросов прерывания осуществляется с помощью специальных программируемых контроллеров прерывания (Programmable Interrupt Controller – PIC). Принципиальная схема PIC приведена на Рис. 3.

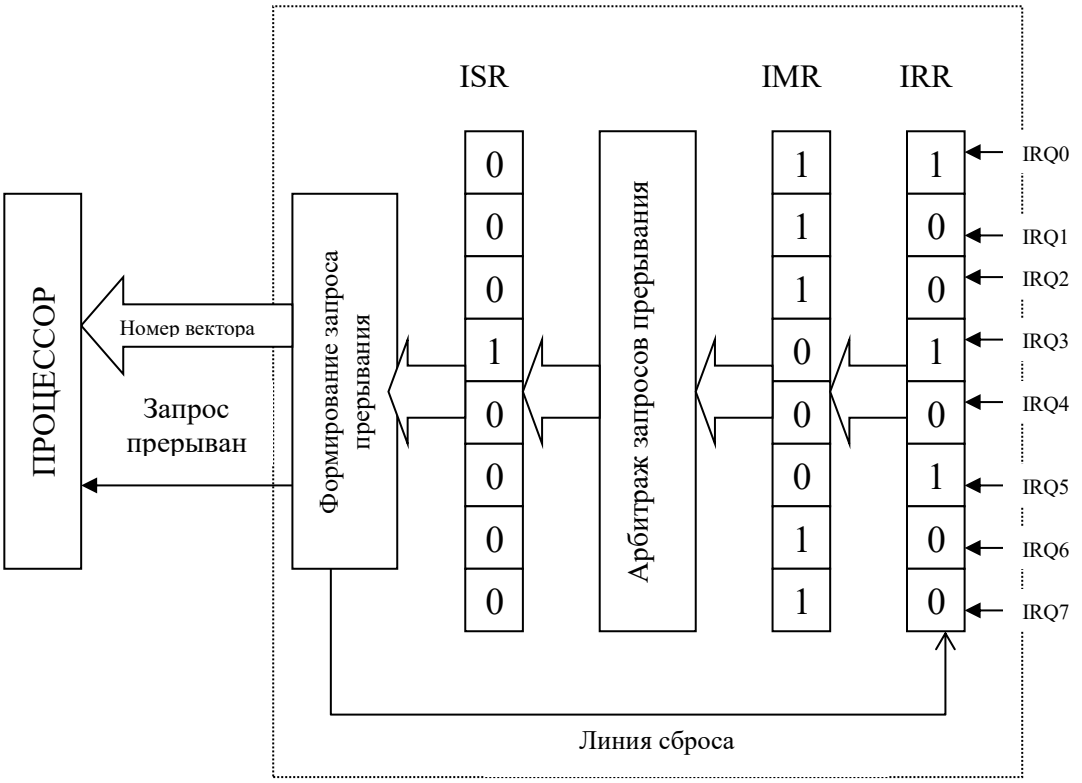


Рис. 3 Принципиальная схема PIC

Контроллер прерывания имеет восемь входов IRQ0,IRQ1,…, IRQ7 (IRQ - Interrupt Request Query) для подключения к ним линий запросов прерываний контроллеров внешних устройств. Поступающие на входы IRQ запросы прерывания от контроллеров внешних устройств устанавливают в 1 соответствующие разряды регистра IRR (Interrupt Request Register). В арбитраж запросов прерывания попадают только те запросы из IRR, которым в регистре маски запросов IMR (Interrupt Mask Register) соответствует нулевое значение разрядов (это запросы IRQ3 и IRQ5). В стандартном режиме настройки PIC более приоритетным считается запрос с меньшим номером. Поэтому в регистре источников прерывания единичное значение формируется в разряде, соответствующем запросу IRQ3. В соответствии с выбранным источником прерывания формируется номер вектора прерывания, после чего подаётся запрос на линию прерывания процессора, который обрабатывается процессором описанным выше способом. После того, как процессор среагирует на прерывание, по линии сброса автоматически осуществляется очистка соответствующего разряда в IRR (в данном случае - IRQ3). Имеется возможность программным способом изменять режимы работы PIC. Например, можно установить другой приоритет линий запроса прерывания, загрузив соответствующую информацию в статусный регистр PIC. Можно также отменить автоматический сброс по линии сброса. В этом случае сброс разрядов регистра IRR должен будет выполняться программно.

Линии запросов прерывания контроллеров устройств коммутируются с некоторым входом контроллера прерываний. Если количество устройств превышает 8, то используется ещё одна микросхема PIC, которая своим выходом связывается с IRQ7 предыдущего PIC, расширяя количество линий подключения запросов прерывания от устройств до 15. В то же время к одной и той же линии контроллера прерываний процессора можно присоединить одновременно несколько контроллеров устройств, которые смогут её разделять.

17.2. Обработка прерываний в QNX

Ядро ОСРВ QNX полностью берет на себя заботу по обслуживанию всех возникающих запросов прерываний от контроллеров устройств, а процессам ПРВ предоставляет программный интерфейс настройки и управления прерываниями, с помощью которого нити процесса могут указывать операционной системе требуемый источник прерываний и переходить в состояние ожидания уведомления от ядра о поступлении ожидаемого запроса прерывания от устройства. После этого при каждом поступлении запроса прерывания от указанного источника ядро уведомляет об этом ожидающую прерывания нить, выводя её из блокирующего состояния ожидания в состояние готовности к выполнению. За нитью сохраняется лишь необходимость выполнения соответствующего протокола взаимодействия с инициатором прерывания (например, с контроллером аналого-цифрового преобразования сигналов от датчиков температуры на физическом объекте).

В качестве источников аппаратных прерываний в QNX логически рассматриваются разряды регистра IRR, которые идентифицируются номерами 0÷15. Типичная семантика номеров прерываний для компьютеров с архитектурой x86 следующая:

0 - Прерывания тиков интервального таймера, поступающие с интервалом, устанавливаемым функцией ClockPeriod().
1 - Прерывания, генерируемые нажатием клавиши клавиатуры.
2 - Прерывания от ведомого контроллера прерываний 8259 (используются при каскадировании аппаратных прерываний).
3 - Прерывания асинхронного порта COM2.
4 - Прерывания асинхронного порта COM1.
5 - Прерывания сетевой (или звуковой) карты.
6 - Прерывания контроллера флоппи диска.
7 - Прерывания принтера (если адаптер принтера использует это прерывание).
8 - Переназначаемое прерывание (для нестандартных внешних устройств).
9 - Переназначаемое прерывание (для нестандартных внешних устройств).
10 - Прерывания сопроцессора.
11 - Прерывания сопроцессора.
12 - Прерывания сопроцессора.
13 - Прерывания сопроцессора.
14 - Прерывания первого IDE-контроллера.
15 - Прерывания второго IDE-контроллера.

Как только возникает сигнал прерывания, ядро переключается на участок кода, который выполняет необходимые подготовительные действия (настраивает окружение) для запуска на выполнение специально определённой в процессе функции, играющей роль *обработчика прерывания* - ISR.

Концепция обработчика прерывания в QNX такова, что он рассматривается как посредник между ядром и соответствующей нитью процесса, ожидающей прерывание. При подключении процесса к источнику прерыванию он указывает ядру свой обработчик прерывания, который каждый раз будет вызываться ядром на выполнение при возникновении прерывания. Эта функция запускается с приоритетом, который выше приоритета любой нити. С учётом этого время выполнения ISR должно быть минимальным, так как в противном случае время,

затрачиваемое на выполнение обработчика прерывания, может оказать серьёзное воздействие на диспетчеризацию нитей. Когда обработчик прерывания завершается, он может либо сообщить ядру, что ничего больше делать не надо (полностью выполнил работу, связанную с обработкой прерывания), либо инициировать посылку от ядра процессу, подключившему ISR, специального уведомления, вследствие которого нить, ожидающая уведомления о прерывании, выходит из состояния ожидания в состояние готовности. В этом случае нить продолжит выполнять действия, связанные с прерыванием.

Интервал времени с момента установки аппаратурой сигнала прерывания до выполнения первой инструкции обработчика прерываний называется *временем реакции на прерывание*. Время реакции на прерывание измеряется в микросекундах. Различные процессоры характеризуются различными временами реакции на прерывание. Это зависит от быстродействия процессора, архитектуры кэша, быстродействия памяти и, конечно, от эффективности операционной системы.

17.3. Программирование обработки прерываний

В общем случае процесс, предполагающий реагировать на прерывания, должен определить и указать ядру специальную функцию - *обработчик прерывания* - ISR, а также иметь в своём составе нить, которая должна подключить обработчик прерывания ISR на нужный номер разряда регистра IRR, связанный с нужной линией запроса прерываний IRQn, и перейти в состояние ожидания прерывания. При поступлении в процессор сигнала прерывания, ожидаемого нитью, выполнение процесса приостанавливается, и ядро асинхронно передаёт управление ISR. Если обработчик прерывания ISR завершается возвратом системного уведомления о прерывании, то ядро доставляет его ожидающей нити, и выводит её из состояния ожидания. В противном случае, сигнал прерывания игнорируется.

17.3.1. Определение обработчика прерываний

Функция, которая используется в качестве обработчика прерывания, должна быть объявлена в виде:

```
struct sigevent* isr(void*area, int id);
```

Функция isr() имеет два аргумента – area и id. Аргумент area – адрес статической области памяти процесса, используемой для обмена данными между нитью, ожидающей уведомление о прерывании и ISR. Это позволяет процессу подготовить в этой области памяти данные, которые будут доступны isr() после запуска ядром, и получить нитью результаты работы isr() после завершения. Если необходимости в обмене данными между процессом и isr() нет, то при установке isr() в качестве area указывается NULL. Второй аргумент – id, предназначен для получения от ядра в теле isr() идентификатора дескриптора источника прерываний. Это необходимо, если один и тот же обработчик прерываний используется для подключения к разным источникам прерываний, и позволяет обработчику отличить источник текущего прерывания.

После завершения работы обработчик прерывания isr() возвращает указатель на структуру struct sigevent*, специфицирующую тип уведомления о прерывании, которое ядро должно послать нити, выполнившей подключение обработчика прерывания. Уведомление о

прерывании должно специфицироваться как уведомление типа SIGEV_INTR. Для этого удобно использовать макрос:

SIGEV_INTR_INIT(struct sigevent *event);

Если обработчик прерывания не хочет инициировать посылку нити уведомления о прерывании, он должен возвратить значение NULL.

При написании обработчика прерываний под ОС QNX необходимо учитывать следующие особенности:

1. Размер стека, отводимого для ISR, ограничен 200 байтами, поэтому следует избегать многочисленных вложенных вызовов функций и рекурсии.
2. Если прерывание разрешено, то, как только происходит прерывание, ядро асинхронно запускает ISR, вытесняя текущую активную нить.
3. Обработчик прерывания выполняется с наивысшим приоритетом (приоритетом ядра), поэтому он должен быть максимально быстрым.
4. Обработчик прерывания не может использовать любые функции программного интерфейса, а только разрешённые ОС для безопасного использования в ISR (см. описание функций), например - InterruptMask(), InterruptUnmask(), InterruptLock(), InterruptUnlock();
5. Переменные, используемые для сохранения значений регистров и адресов памяти аппаратуры, должны объявляться в процессе с модификатором volatile (например, volatile int …), чтобы запретить компилятору кэшировать значения этих переменных, поскольку они могут быть изменены в любой точке выполнения процесса.

17.3.2. Подключение процесса к источнику прерываний

Ядро не позволяет любой нити процесса выполнять подключение обработчика прерываний ISR к источнику прерываний для получения от ядра уведомления прерываний. Такая нить должна предварительно получить право привилегированного ввода/вывода. Получить это право могут только нити, которые выполняются под идентификатором пользователя root или которые установили свой идентификатор пользователя в root при помощи функции setuid(). Следовательно, реально эти права есть только у пользователя root.

Для получения нитью права привилегированного ввода/вывода, перед тем как установить обработчик прерываний, используется функция:

#include <sys/neutrino.h>
int ThreadCtl(int cmd,void *data);

В качестве cmd следует использовать значение _NTO_TCTL_IO, а data – NULL. В итоге вызов функции должен иметь вид:

ThreadCtl(_NTO_TCTL_IO, NULL)

В случае ошибки функция возвращает -1 и заносит код ошибки в errno.

Процесс может подключаться к источнику прерываний с установкой собственного обработчика прерываний или с установкой обработчика прерываний по умолчанию.

17.3.2.1. Подключение собственного обработчика прерываний

Чтобы подключиться к прерыванию с указанием собственного обработчика прерываний ISR, нить должна использовать функцию:

```
#include <sys/neutrino.h>
int InterruptAttach(int intr,
                    const struct sigevent* (*isr)(void*area,int id),
                    const void *area,
                    int size,
                    unsigned flags);
```

intr – номер источника прерываний;

isr – адрес ISR;

area – адрес статической области памяти, используемой для обмена данными между процессом и ISR, который передаётся в isr() в качестве первого аргумента, или NULL, если память не используется;

size – размер области area или 0, если память не используется;

flags – флаги, специфицирующие порядок использования ISR.

Значение аргумента flags может быть равно 0 и тогда статус ISR устанавливается по умолчанию. Явно можно установить следующие опции:

_NTO_INTR_FLAGS_END – указывает, что данный ISR должен сработать после всех других обработчиков данного прерывания (если номер прерывания разделяется несколькими обработчиками);

_NTO_INTR_FLAGS_PROCESS – указывает на то, что ISR необходимо ассоциировать с процессом, а не с нитью, его подключившей. В результате ISR будет отключаться автоматически от прерывания только при завершении процесса, а не нити;

_NTO_INTR_FLAGS_TRK_MSK – указывает, что ядро должно отследить, сколько раз данное прерывание было маскировано. Это приводит к несколько большей загрузке ядра, но обеспечит корректное демаскирование прерывания при завершении нити или процесса.

Функция возвращает ID подключённого источника прерываний. В случае ошибки возвращается -1 и код ошибки устанавливается в errno.

17.3.2.2. Установка обработчика прерываний по умолчанию

При подключении процессом источника прерываний без явной установки своего обработчика прерываний используется функция:

```
#include <sys/neutrino.h>
int InterruptAttachEvent(int intr, const struct sigevent* event, unsigned flags);
```

В этом случае ядро по умолчанию сразу направляет соответствующей нити уведомление прерывания. Нить будет активироваться по каждому прерыванию, хотя для разделяемых источников прерываний различными инициаторами сигналов прерывания необходимость в этом для данной нити может отсутствовать (если используется ISR, то она могла бы проверить состояние своего источника сигнала прерывания непосредственно в контроллере внешнего устройства, и не инициировать уведомления для нити, если сигнал не установлен). Такой режим обработки прерываний увеличивает затраты ядра на перепланирование нитей. Однако преимущество использования этой функции заключается в том, что отсутствие явно заданного

ISR устраняет опасность разрушить систему, в случае ошибок программирования в ISR. Обработчик прерываний выполняется в пространстве ядра и ошибки ISR могут приводить к фатальным последствиям для системы реального времени. Выбор способа подключения к процессу источника прерываний в конечном счёте зависит от конкретных особенностей разрабатываемого ПРВ.

17.3.3. Отключение процесса от прерывания

Когда у процесса отпадает необходимость в обработке некоторого подключённого ранее источника прерываний, он может отключить его, используя функцию:

```
#include <sys/neutrino.h>
int InterruptDetach(int id);
```

Функция отключает прерывание, дескриптор которого, задан в id. Когда нить или процесс завершаются, то подключённые ими источники прерываний автоматически отключаются. Если окажется, что это последний процесс, связанный с данным источником прерываний, то ядро автоматически маскирует соответствующий источник прерываний, чтобы запретить прерывания от этого источника.

17.3.4. Управление прерываниями

Все процессы могут управлять подключёнными источниками прерываний на уровне PIC и на уровне процессора. Управление прерываниями на уровне PIC осуществляется путём маскирования/демаскирования процессом источников прерываний, устанавливая соответствующие значения разрядов регистра IMR. Это предохраняет процесс от нежелательных запросов прерывания, поступающих по замаскированному источнику прерываний. Для этих целей используются функции:

```
#include <sys/neutrino.h>
int InterruptMask(int intr,int id);
int InterruptUnmask(int intr,int id);
```

Функции позволяют маскировать/демаскировать прерывание с номером intr, подключённое к процессу с дескриптором id, который возвращается функциями InterruptAttach() или InterruptAttachEvent(). В качестве id можно задать –1, если необходимо, чтобы ядро автоматически отслеживало маскирование/демаскирование прерываний каждым обработчиком.

Параметр id игнорируется, если при подключении обработчика был установлен флаг _NTO_INTR_FLAGS_TRK_MSK.

Если необходимо запретить прерывать процессор при выполнении, например, критических участков программного кода, то можно блокировать/деблокировать прерывания на уровне процессора. Для этого используются функции:

```
#include <sys/neutrino.h>
void InterruptLock(intrspin_t* spinlock);
void InterruptUnlock(intrspin_t* spinlock);
```

Функция InterruptLock() блокирует, а InterruptUnlock() деблокирует прерывания процессора. Для управления прерываниями на уровне процессора с использованием этих

функций необходимо предварительно определить переменную системного типа intrspin_t и использовать указатель на неё в функциях в качестве параметра spinlock, разделяемого обработчиком прерывания и установившей его нитью, предназначенного для согласования их действий блокирования/деблокирования прерывания процессора. Если spinlock не является статическим (создан в динамически выделенной памяти), то его перед использованием необходимо проинициализировать с помощью вызова функции memset(spinlock,0, sizeof(*spinlock)).

Функция InterruptLock() пытается захватить spinlock, пока прерывания процессора блокированы. После этого последующее выполнение команд происходит без прерывания. Важно скорее выполнить команды и деблокировать прерывания процессора. Обычно это выглядит так:

```
…
InterruptLock (&spinner);
/* критическая секция */
InterruptUnlock (&spinner);
…
```

С помощью функций InterruptLock() и InterruptUnlock() решается общая для многих систем реального времени проблема синхронизации доступа к общим данным между обработчиком прерывания и нитью, его установившей.

17.3.5. Ожидание нитью уведомления о прерывании

Для ожидания уведомления о прерывании типа SIGEV_INTR нить должна вызвать функцию:

```
#include <sys/neutrino.h>
int InterruptWait(int flags, const uint64_t *timeout);
```

В рассматриваемой версии ОСРВ QNX flags должен быть равен 0, а timeout должен быть равен NULL. Эта функция переводит нить в INTR-блокированное состояние до момента прихода события типа SIGEV_INTR. Чтобы ограничить время блокировки можно воспользоваться таймаутом ядра. Если уведомление приходит раньше выполнения функции, то функция сразу же успешно завершается.

В случае ошибки функция возвращает -1, при успешном завершении – любое другое значение, отличное от -1.

17.3.6. Общий формат процесса с обработкой прерываний

Общая структура функции main() процесса, осуществляющего обработку прерываний, имеет вид:

```
#include <sys/neutrino.h>
#define IRQ3 3 //Номер подключаемого источника прерывания - 3
...
struct sigevent event;//Уведомление о прерывании
...
void *intr_thread (void *arg);//Нить ожидающая прерывания
```

```
const struct sigevent* isr(void *area, int id);//Функция ISR - обработчик прерывания
...
/********************** main() **********************************/
int main(){
/* Выполнить необходимые инициализации устройств и т.п. */
SIGEV_INTR_INIT(&event); //Уведомление типа SIGEV_INTR
...
/* Запустить нить для получения уведомлений о прерываниях от устройства*/
pthread_create (NULL, NULL, intr_thread, NULL);
...
/* Продолжить выполнение необходимых действий */
...
}

/********************** intr_thread () ********************/
/* Эта нить предназначена для получения уведомлений, инициируемых прерываниями */
void * intr_thread (void *arg){
/* Разрешить нити привилегированный доступ к адресам и портам устройств */
ThreadCtl(_NTO_TCTL_IO, NULL);
...
/* Инициализация устройств и т.п. */
...
/* Подключение ISR к линии IRQ3 - поступления прерываний от устройств */
InterruptAttach(IRQ3,isr,NULL,0,0);
...
/* Целесообразно увеличить приоритет нити, ожидающей прерывания */
...
/* Ждём уведомлений о прерываниях и выполняем необходимую обработку */
while(1){
InterruptWait(NULL, NULL);
/* Попадаем сюда, когда InterruptWait() деблокируется в результате прихода уведомления
типа SIGEV_INTR, сформированного ядром по заказу ISR, что говорит о необходимости
выполнить соответствующие действия */
...
/* Нить выполняет обработку уведомления о прерывании */
...
/* Если в isr() была выполнена InterruptMask(), то нить должна выполнить InterruptUnmask(),
чтобы разрешить прерывания от аппаратуры */
}
}
```

```
/********************** isr() ******************************/
// Это общая структура ISR
/* Объявление переменных, используемые для сохранения значений регистров и адресов
памяти аппаратуры.
Они должны объявляться с модификатором volatile (например, volatile int …), чтобы запретить
компилятору кэшировать значения этих переменных, поскольку они могут быть изменены в
любой точке выполнения процесса */
const struct sigevent* isr(void *area, int id){
        ...
/* Проанализировать установку запроса прерывания в контроллере внешнего устройства */
if(/* Признак отсутствует */){
        return(NULL); /*Не уведомлять нить*/
}
/* Пришёл сигнал прерывания. Сбросить в контроллере устройства запрос прерывания или,
по крайней мере, замаскировать сигнал прерывания в PIC, выполнив функцию
InterruptMask(), тем самым запретив повторные прерывания ядра */

/* Возвратить указатель на структуру event - тип уведомления SIGEV_INTR, для посылки
уведомления нити, выполнившая вызов InterruptWait(), она будет выведена из состояния
ожидания. */
return (&event);
}
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Эрджиес, К. Распределённые системы реального времени. Теория и практика / К. Эрджиес, пер. с анг. В. А. Яроцкий. - Москва: ДМК Пресс, 2020. - 382 с.

2. Колесов Н.В., Толмачева М.В., Юхта П.В. Системы реального времени. Планирование, анализ, диагностирование. - Санкт-Петербург: Электроприбор, 2014. - 180 с.

3. Системы реального времени: конспект лекций / сост. А. С. Голубев. – Владимир: Изд-во Владим. гос. ун-та, 2010. – 127 с. Электронный вариант книги по адресу https://dspace.www1.vlsu.ru/bitstream/123456789/1853/3/00727.pdf.

4. Михайлов А. А. Системы реального времени. Программно-технический комплекс: учеб. пособие / Юж. – Рос. гос. техн. ун-т. – Новочеркасск: ЮРГТУ, 2010. – 292 с.

5. Баландин, А.В., Николаев, А.В. Метод структуризации и РВ-верификации приложений реального времени для систем промышленной автоматизации // Надёжность и качество. - Труды международного симпозиума. – Пенза: Изд-во Пенз. гос. ун-та. 2003. – С.378-380.

6. Базаркин А.Н. Разработка темпоральной модели данных в медицинской информационной системе // Программные продукты и системы. - 2009. - № 2, - С. 34-40. Электронный вариант по адресу bugs@interin.ru.

7. Баландин, А.В. Модель параллельных и асинхронных темпоральных вычислений с автовалидацией // Перспективные информационные технологии (ПИТ 2015). – Том 2: труды международной научно-технической конференции / под ред. С.А. Прохорова. - Самара: Изд-во Самарского научного центра РАН. 2015. – С.3-7.

8. Баландин, А.В. Потоковые диаграммы асинхронных темпоральных вычислений для моделирования и РВ-верификации приложений реального времени [Текст] // Информационные технологии и нанотехнологии (ИТНТ-2016). – Сб. трудов международной конференции. – Самара: Изд-во СГАУ. 2016. – С.919-926.

9. Практика работы с QNX / Алексеев [и др.]. – Москва: Издательский Дом «КомБук», 2004. – 432 с.

10. Кёртен, Р. Введение в QNX Neutrino 2. Руководство для разработчиков приложений реального времени. – СПб.: БХВ-Петербург, 2005. – 400 с.: ил.

11. Операционная система реального времени QNX Neutrino 6.3. Руководство пользователя: Пер. с англ. – СПб.: БХВ-Петербург, 2009. – 480 с. Ил.

12. Зыль С. Н. Операционная система реального времени QNX: от теории к практике. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 192 с: ил. Электронный вариант книги по адресу https://studfile.net/preview/2820829/.

13. Цилюрик О., Горошко Е. QNX/UNIX: анатомия параллелизма. - СПб.: Символ-Плюс, 2006. – 288 с., ил.

14. Защищённая операционная система реального времени (ЗОСРВ) «Нейтрино». Редакция 2020. Электронный адрес - https://help.kpda.ru/help/index.jsp.

15. Дорогов А.Ю. Синхронизация и взаимодействие программных потоков в операционной среде реального времени: Учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2007. 64 с. Электронный вариант книги по адресу https://zzapomni.com/dorogov-sinhronizaciya-i-vzaimodey-2007-12221/1.

16. Никольский О.Л. Программирование приложений реального времени для исполнения в среде операционной системы реального времени QNX/Neutrino 2. Часть I Служба времени

операционной системы реального времени QNX/Neutrino (Версия 9-02-2007) Брянск: Брянский государственный технический университет, 2007. – 189 с. Электронный вариант книги по адресу http://swd.ru/files/pdf/nop/bgtu/BGTU_Posobie_pI-Slugba_vremeni.pdf.

17. Никольский О.Л. Программирование приложений реального времени для исполнения в среде операционной системы реального времени QNX/Neutrino 2. Часть II Обработка прерываний в операционной системе реального времени QNX/Neutrino (Версия 9-02-2007) Брянск: Брянский государственный технический университет, 2007. – 90 с. Электронный вариант книги по адресу http://swd.ru/files/pdf/nop/bgtu/BGTU_Posobie_pII-Obrabotka_prerivanii.pdf.

ПРИЛОЖЕНИЕ

Системные сигналы стандарта POSIX

Стандарт POSIX 1003.1a определяет 32 сигнала, включающие в себя следующие сигналы:

Системная символьная константа	Диспозиция по умолчанию	Событие инициации сигнала
SIGHUP	Завершить	Посылается лидеру сеанса, связанному с управляющим терминалом, когда ядро обнаруживает, что терминал отсоединился (потеря линии). Сигнал также посылается всем процессам текущей группы при завершении выполнения лидера. Сигнал удобно использовать для взаимодействия с демонами. Демон не имеет управляющего терминала и, соответственно, обычно не получает этот сигнал.
SIGINT	Завершить	Сигнал для уведомления процессов текущей группы о терминальном прерывании (пользователь может инициировать сигнал нажатием клавиш <Ctrl>+<C>).
SIGQUIT	Завершить+core	Сигнал уведомления процессов текущей группы о нажатии клавиш <Ctrl>+<\>.
SIGILL	Завершить+core	Сигнал уведомления процесса о попытке выполнить недопустимую инструкцию (после перехвата повторно не устанавливается).
SIGTRAP	Завершить	Сигнал уведомления процесса о выполнении им trap-прерывания при трассировке.
SIGIOT	Завершить	Сигнал уведомления процесса о выполнении им IOT-инструкции.
SIGABRT	Завершить+core	Сигнал уведомления процесса о выполнении им системного вызова abort().
SIGEMT	Завершить	Сигнал как реакция ядра на выполнение процессом EMT-инструкция.
SIGFPE	Завершить+core	Сигнал уведомления процесса о возникновении особой ситуации, такой как деление на 0 или переполнение операции с плавающей точкой.
SIGKILL	Завершить	Сигнал завершения процесса. При получении сигнала процесс завершается (не может быть перехвачен или игнорирован).
SIGBUS	Завершить+core	Сигнал уведомления процесса об ошибке шины. Сигнал свидетельствует о некоторой аппаратной ошибке (например, обращение к допустимому виртуальному адресу, для которого отсутствует физическая страница).
SIGSEGV	Завершить+core	Сигнал уведомления процесса о нарушении сегментации. Попытка обращения к недопустимому адресу или к области памяти, для которой у процесса недостаточно привилегий.
SIGSYS	Завершить+core	Сигнал уведомления процесса при попытке недопустимого системного вызова (об использовании ошибочного аргумента в системном вызове).
SIGPIPE	Завершить	Сигнал уведомления процесса о выполненной записи в устройство pipe, не открытого для чтения.

SIGALRM	Завершить	Сигнал для уведомления процесса службой часов реального времени.
SIGTERM	Завершить	Сигнал предупреждения, что процесс будет уничтожен. Позволяет процессу подготовиться к завершению.
SIGUSR1	Завершить	Сигнал, не инициируемый ядром, а только пользователем или прикладным процессом.
SIGUSR2	Завершить	Сигнал, не инициируемый ядром, а только пользователем или прикладным процессом.
SIGCHLD	Игнорировать	Сигнал для уведомления родительского процесса о завершении дочернего процесса.
SIGPWR	Игнорировать	Сигнал для уведомления процесса об угрозе потери питания. Обычно он отправляется, когда питание системы переключается на источник бесперебойного питания (UPS).
SIGWINCH	Игнорировать	Сигнал для уведомления процесса об изменении окна
SIGURG	Игнорировать	Сигнал для уведомления процесса о срочном событии на канале ввода-вывода
SIGPOLL	Завершить	Сигнал для уведомления процесса о наступлении определённого события для устройства, которое является опрашиваемым.
SIGIO	Игнорировать	Сигнал уведомления асинхронного ввода-вывода.
SIGSTOP	Остановить	Сигнал остановки, инициированный не с устройства tty (не может быть перехвачен или игнорирован).
SIGTSTP	Остановить	Сигнал остановки инициированный, с устройства tty.
SIGCONT	Продолжить	Сигнал запуска остановленного процесса для продолжения выполнения.
SIGTTIN	Остановить	Сигнал для уведомления процесса фоновой группы о попытке фонового чтения с управляющего терминала tty.
SIGTTOU	Остановить	Сигнал для уведомления процесса фоновой группы о попытке фоновой записи на управляющий терминал tty.

В дополнение к перечисленным сигналам существуют 24 сигнала, используемых службой реального времени (POSIX 1003.1b). Номер первого сигнала реального времени - SIGRTMIN, номер последнего сигнала реального времени. - SIGRTMAX.

Весь диапазон сигналов предполагает 64 сигнала. Диапазон начинается с _SIGMIN – 1, и заканчивается _SIGMAX - 64.

Заметим, что нельзя ни игнорировать, ни перехватить сигналы SIGKILL или SIGSTOP. Для них всегда выполняется действие по умолчанию. Кроме того, при завершении процесса по умолчанию в результате прихода сигнала в ряде случаев в текущем каталоге процесса создаётся файл **core** (в таблице такие случаи отмечены как "Завершить+core"), в котором храниться образ памяти процесса. Этот файл может быть проанализирован программой-отладчиком для определения состояния процесса непосредственно перед завершением.