**15. Многозадачность**

Встроенная поддержка многозадачности является уникальной и широко известной особенностью языка программирования Ада, которая выгодно отличает его от большинства современных языков программирования. Следует особо подчеркнуть, что поддержка многозадачности обеспечивается не с помощью каких-либо расширений или внешних библиотек, а с помощью строго стандартизированных средств, которые встроены непосредственно в язык программирования.

Обеспечение поддержки многозадачности на уровне языка программирования имеет важное значение. Благодаря этому достигается стабильность семантики, которая избавляет разработчика программного обеспечения как от необходимости использования разнородных внешних библиотек, так и от необходимости разработки своих собственных решений для обеспечения поддержки многозадачности. В результате, облегчается общее понимание исходных текстов многозадачных программ и обеспечивается лучшая переносимость программного обеспечения.

В качестве стандартных средств поддержки многозадачности Ады используются задачи (*tasks*), которые хорошо известны со времен стандарта Ada83 и описывают последовательности действий, которые способны выполняться одновременно, а также объекты защищенных типов (*protected types*), которые являются нововведением стандарта Ada95.

Программа на языке Ада состоит как минимум из одной, а возможно и множества задач, выполняющихся одновременно. Каждая задача выполняется независимо от остальных задач. Механизмы межзадачного обмена данными и синхронизации основаны на высокоуровневых концепцях рандеву и использовании защищенных объектов. Следует заметить, что рандеву и защищенные объекты обладают более высоким уровнем абстракции по сравнению с семафорами. Они предоставляют средства защитной блокировки и таймаутов, а также средства для выполнения выборочного перестроения очередей клиентов и аварийного завершения.

"Гибель" одной задачи не обязательно оказывает влияние на выполнение других задач, за исключением случаев, когда они пытаются с ней взаимодействовать. Специализированные задачи отсутствуют. Возможны ситуации, когда основная процедура Ада-программы может завершиться, а остальные задачи продолжат свое выполнение.

**15.1 Задачи**

**15.1.1 Типы и объекты задач**

Конструкция задачи обладает свойствами, которые характерны для пакетов, процедур и структур данных:

* Подобно пакетам, задача имеет спецификацию и тело, однако она не может быть самостоятельно компилируемой единицей, помещенной в свой собственный файл с исходным текстом. Вместо этого, задача должна быть помещена в другую структуру (например, пакет или процедуру)
* Подобно процедуре, задача содержит раздел описаний и исполнительную часть, однако она не вызывается как процедура. Вместо этого, она начинает выполняться автоматически, как часть структуры, в которой она описана.
* Подобно структуре данных, задача имеет тип и может существовать как переменная этого типа. Кроме того, подобно записи, задача может иметь дискриминанты.

Спецификация задачи, начинающаяся зарезервированными словами **task type**, определяет тип задачи (или задачный тип). Значение объекта (переменная) типа задачи представляет собой задачу. Спецификация задачи, не содержащаяя зарезервированного слова **type**, определяет одиночную задачу, а описание задачи с такой спецификацией равносильно описанию анонимного типа задачи с одновременным описанием объекта этого типа.

Простым примером многозадачной программы может служить следующая программа:

|  |
| --- |
| **with** Ada.Text\_IO;  **procedure** Multitasking\_Demo **is**  -- спецификация анонимной задачи  **task** Anonimous\_Task;  -- тело анонимной задачи  **task body** Anonimous\_Task **is**  **begin** -- для Anonimous\_Task  **for** Count **in** 1..5 **loop**  Ada.Text\_IO.Put\_Line("Hello from Anonimous\_Task");  **end loop**;  **end** Anonimous\_Task;  -- спецификация типа задачи  **task type** Simple\_Task (Message: Character);  -- тип задачи имеет тело  **task body** Simple\_Task **is**  **begin** -- для Simple\_Task  **for** Count **in** 1..5 **loop**  Ada.Text\_IO.Put\_Line("Hello from Simple\_Task " & Message);  **end loop**;  **end** Simple\_Task;  -- переменная задачного типа  Simple\_Task\_Variable: Simple\_Task(Message => 'A');  **begin** -- для Multitasking\_Demo  -- в отличие от процедур, задачи не вызываются,  -- а активируются автоматически  -- выполнение обоих задач начинается как только  -- управление достигнет этой точки, то есть, сразу  -- после "begin", но перед выполнением первой инструкции  -- головной процедуры Multitasking\_Demo  **null**;  **end** Multitasking\_Demo; |

В данном примере описана одиночная задача анонимного типа Anonimous\_Task, тип задачи Simple\_Task и переменная задачи Simple\_Task\_Variable, имеющая тип Simple\_Task. Примечательно, что описание типа задачи Simple\_Task содержит дискриминант, значение которого используется как параметр задачи и указывается при описании переменной задачи Simple\_Task\_Variable. Следует также обратить внимание на то, что любой тип задачи является лимитированным и, таким образом, для него не существует предопределенных операций присваивания или сравнения. Алгоритмы работы обоих задач просты и подобны - каждая задача выводит пять приветственных сообщений и завершает свою работу.

Рассмотрим еще один простой пример, который демонстрирует использование задачного типа для описания более одного экземпляра объекта задачи.

|  |
| --- |
| **with** Ada.Text\_IO;  **procedure** Multitasking\_Demo\_2 **is**  -- спецификация типа задачи  **task type** Simple\_Task (Message: Character; How\_Many: Positive);  -- тело типа задачи  **task body** Simple\_Task **is**  **begin** -- для Simple\_Task  **for** Count **in** 1..How\_Many **loop**  Ada.Text\_IO.Put\_Line("Hello from Simple\_Task " & Message);  **delay** 0.1;  **end loop**;  **end** Simple\_Task;  -- переменные задачного типа  Simple\_Task\_A: Simple\_Task(Message => 'A', How\_Many => 5);  Simple\_Task\_B: Simple\_Task(Message => 'B', How\_Many => 10);  **begin** -- для Multitasking\_Demo\_2  **null**;  **end** Multitasking\_Demo\_2; |

В этом примере, тип задачи Simple\_Task содержит два дискриминанта: дискриминант Message типа Character используется при выдаче приветственного сообщения, как и ранее, а дискриминант How\_Many используется для указания количества выдаваемых сообщений. Таким образом, переменная задачи Simple\_Task\_A выдаст пять приветствий, а переменная Simple\_Task\_B - десять.

**15.1.2 Инструкции задержки выполнения (*delay statements*)**

При внимательном рассмотрении исходного текста последнего примера можно обнаружить, что после инструкции вывода сообщения приветствия располагается инструкция "**delay** 0.1;", с помощью которой в данном случае осуществляется задержка выполнения задачи на 0.1 секунды.

Инструкции задержки выполнения могут быть использованы для приостановки выполнения тела задачи (или программы) на некоторое время. Рзличают два вида инструкций задержки выполнения: относительная задержка выполнения и абсолютная задержка выполнения.

Общий вид инструкции относительной задержки выполнения следующий:

|  |
| --- |
| **delay** *время\_задержки*; |

Здесь результат выражения *время\_задержки*, имеющего предопределенный вещественный тип с фиксированной точкой Duration (описан в пакете *Standard*), указывает длительность задержки выполнения в секундах, на которую необходимо задержать выполнения задачи (или программы). При этом отсчет времени задержки выполняется относительно текущего момента времени. Наглядным примером использования инструкции относительной задержки, для задержки выполнения задачи на одну секунду, может служить следующее:

|  |
| --- |
| **delay** 1.0; |

Общий вид инструкции абсолютной задержки выполнения следующий:

|  |
| --- |
| **delay until** *время\_задержки*; |

В этом случае результат выражения *время\_задержки*, любого неограниченного типа, указывает момент времени, до наступления которого необходимо задержать выполнение задачи (или программы). Следующий пример демонстрирует использование инструкции абсолютной задержки выполнения задачи:

|  |
| --- |
| **delay until** Time\_Of(2010, 1, 1, 0.0); -- задержка выполнения до 1 января 2010 года |

Следует также заметить, что описание предопределенного типа времени Time и ассоциируемых с ним операций предусматривается в стандартном пакете *Ada.Calendar*. Кроме того, пакет *Ada.Calendar* предоставляет функцию Clock, которая возвращает значение текущего момента времени (для получения более полной информации следует обратиться к спецификации этого пакета).

Отметим также, что при построении систем, которые должны работать в реальном масштабе времени, вместо типа времени Time, описанного в стандартном пакете *Ada.Calendar*, следует использовать тип времени Time, который описывается в пакете *Ada.Real\_Time* (для получения более полной информации следует обратиться к спецификации пакета *Ada.Real\_Time*).

**15.1.3 Динамическое создание объектов задач**

Ада предоставляет возможность динамически порождать объекты задач с помощью **new**, что может оказаться полезным, когда необходимо множество объектов задач одного типа. Предположим, что у нас есть следующая спецификация типа задачи:

|  |
| --- |
| **task type** Counting\_Task **is**  **entry** Add (Amount : **in** Integer);  **entry** Current\_Total (Total : **out** Integer);  **end** Counting\_Task; |

Мы можем описать ссылочный тип Counting\_Task\_Ptr, который позволяет ссылаться на объекты задач типа Counting\_Task. Затем мы можем описывать переменные ссылочного типа Counting\_Task\_Ptr и с помощью **new** создавать динамические объекты задач, как показано в следующем примере:

|  |
| --- |
| **declare**  . . .  **type** Counting\_Task\_Ptr **is access** Counting\_Task;  Task\_Ptr : Counting\_Task\_Ptr; -- переменная ссылающаяся на объект задачи  . . .  **begin**  . . .  Task\_Ptr := **new** Counting\_Task; -- динамическое создание объекта задачи  . . .  **end**; |

Подобным образом можно динамически создать любое необходимое количество объектов задач. Поскольку реализация Ада-системы не обязана освобождать память, занятую при динамическом создании объекта задачи, то это необходимо выполнять самостоятельно, используя предопределенную настраиваемую процедуру Ada.Unchecked\_Deallocation.

**15.1.4 Принудительное завершение abort**

Ада позволяет принудительно завершать выполнения объекта задачи. Это может быть выполнено с помощью инструкции прекращения, которая может иметь следующий вид:

|  |
| --- |
| **abort** Some\_Task\_Name; |

Здесь Some\_Task\_Name - это имя какого-либо объекта задачи. Считается, что принудительно прекращенная задача находится в "ненормальном" (*abnormal*) состоянии и не может взаимодействовать с другими задачами. После того, как состояние задачи отмечено как "ненормальное", выполнение ее тела прекращается. Это подразумевает, что прекращается выполнение любых инструкций, расположенных в теле задачи, за исключением тех, которые вызывают операции, отложенные до принудительного прекращения (*abort-deffered operations*).

Следует заметить, что использование принудительного прекращения выполнения задачи является "аварийным" действием, и должно применяться только в тех случаях, когда принудительное прекращение выполнения задачи действительно необходимо (например, когда задача "зависла").

Следует понимать, что использование такого "сильнодействующего" средства остановки для задач, которые выполняют сохранение каких-либо дисковых данных, может повлечь за собой не только потерю самих данных, но и повреждение логической структуры дискового накопителя.

**15.1.5 Приоритеты задач**

Каждая задача Ады может обладать своим собственным приоритетом выполнения, который задается с помощью директивы компилятора Priority:

|  |
| --- |
| **pragma** Priority ( *expression* ); |

Непосредственное использование этой директивы компилятора допускается:

* внутри спецификации задачи,
* внутри спецификации защищенного типа или объекта,
* в описательной части тела подпрограммы

Значение результата выражения *expression*, используемого для непосредственного указания приоритета, должно принадлежать целочисленному типу Integer, причем при указании директивы Priority в описательной части тела подпрограммы выражение *expression* должно быть статическим, а его значение должно принадлежать диапазону значений подтипа Priority, который описан в пакете *System*. Например:

|  |
| --- |
| **task** Some\_Task **is**  **pragma** Priority (5);  . . .  **end** Some\_Task; |

Приоритет выполнения задачи определяет ее привилегии на обладание системными ресурсами (например, процессором). В простейшем случае, если две задачи с разными приоритетами готовы к выполнению, то к выполнению будет допущена та задача, приоритет которой выше. При равенстве приоритетов порядок выполнения задач не определен.

В Ada83 приоритет задачи строго фиксировался при ее описании. Согласно стандарту Ada95, приоритет задачи может быть изменен в процессе существования задачи (кроме приоритета, указанного для подпрограммы), то есть задачи могут иметь динамически изменяемые приоритеты.

Средства динамического определения и изменения текущего приоритета задачи предоставляются предопределенным стандартным пакетом *Ada.Dynamic\_Priorities*. Спецификация этого пакета проста и имеет следующий вид:

|  |
| --- |
| **with** System;  **with** Ada.Task\_Identification; -- спецификация этого пакета обсуждается в 15.2.7,  -- при рассмотрении вопроса идентификации задач  **package** Ada.Dynamic\_Priorities **is**  **procedure** Set\_Priority  (Priority : **in** System.Any\_Priority;  T : **in** Ada.Task\_Identification.Task\_ID :=  Ada.Task\_Identification.Current\_Task);  **function** Get\_Priority  (T : Ada.Task\_Identification.Task\_ID :=  Ada.Task\_Identification.Current\_Task)  **return** System.Any\_Priority;  **end** Ada.Dynamic\_Priorities; |

Следует заметить, что правила планирования выполнения задач на основе статических и динамических приоритетов рассматриваются в приложении *D* (*Annex D*) стандарта Ada95, в котором указываются требования для систем реального времени.

**15.2 Взаимодействие задач**

Показанные ранее примеры многозадачных программ демонстрировали только способность задач выполняться одновременно. При этом организация взаимодействия различных задач между собой не рассматривалась. В реальности обеспечение такого взаимодействия между одновременно выполняющимися задачами с целью обмена данными или взаимной синхронизации работы имеет важное значение.

**15.2.1 Концепция рандеву**

Как один из механизмов обеспечения надежного межзадачного обмена данными и взаимной синхронизации работы задач, Ада предоставляет механизм рандеву. Основополагающая идея механизма рандеву достаточно проста. В спецификации задачи публикуются различные входы (*entry*) в задачу, в которых она готова ожидать обращения к ней от других задач. Далее, в теле задачи указываются инструкции принятия обращений к соответствующим входам, указанным в спецификации этой задачи.

Необходимо обратить внимание на несимметричность такого механизма взаимодействия. Это означает, что в процессе взаимодействия одна из задач рассматривается как сервер, а вторая - как клиент, причем задача-сервер не может быть инициатором начала взаимодействия.

В простейшем случае, когда рассматривается взаимодействие только двух задач, задача-клиент, желающая обратиться к другой задаче (задаче-серверу), инициирует обращение к входу задачи-сервера. После этого задача-сервер откликается на обращение задачи-клиента, принимая обращение к этому входу. Таким образом, взаимодействие двух задач осуществляется в ситуации, когда задача-клиент обратилась к входу, а задача-сервер готова принять это обращение. Этот способ взаимодействия двух задач называется рандеву.

Поскольку задача-клиент и задача-сервер выполняются независимо друг от друга, то нет никакой гарантии, что обе задачи окажутся в точке осуществления рандеву одновременно. Поэтому, если задача-сервер оказалась в точке рандеву, но при этом нет ни одного обращения к входу (запроса на взаимодействие), то она должна ждать появления такого обращения. Аналогично, если задача-клиент обращается к входу, а задача-сервер не готова обслужить такое обращение, то задача-клиент должна ждать, пока задача-сервер обслужит это обращение. В процессе ожидания как задача-клиент, так и задача-сервер не занимают ресурсы процессора, находясь в состоянии, которое называют приостановленным или состоянием блокировки.

В случаях, когда вызовы к входу задачи-сервера осуществляют сразу несколько задач-клиентов, эти вызовы ставятся в очередь. Порядок обслуживания такой очереди зависит от соответствия конкретной реализации Ада-системы требованиям приложения *D* (*Annex D*) стандарта Ada95, в котором указываются требования для систем реального времени. Если реализация Ада-системы не обеспечивает соответствия этим требованиям, то очередь обслуживается в порядке поступления вызовов (*FIFO - First-In-First-Out*).

**15.2.2 Описание входов**

Чтобы некоторая задача-клиент могла инициировать рандеву с задачей-сервером, описание спецификации задачи-сервера должно содержать описание соответствующего входа. Следует заметить, что описания входов могут быть помещены только в описание спецификации задачи, кроме того, задача может иметь приватные входы. Таким образом, описание входа в спецификации задачи-сервера может рассматриваться как декларация сервиса, предоставляемого задачей-сервером для задач-клиентов.

Для описания входов задачи-сервера используется зарезервированное слово **entry**, а семантика входов задач очень похожа на семантику процедур:

* Так же, как и процедуры, входы задач имеют имена и могут иметь различные параметры. Для имен входов допускается совмещение имен, что подразумевает наличие у одной задачи нескольких входов с одинаковыми именами, но различными параметрами.
* Параметры входов задачи, так же, как и параметры процедур, могут использоваться в режимах "**in**", "**in out**" и "**out**", и могут иметь значения по умолчанию.

При описании параметров входа задачи-сервера следует учитывать, что, в отличие от процедур, для входов задач не допускаются ссылочные параметры, хотя допускаются параметры ссылочного типа. Кроме того, при описании входа может быть опционально указан дискретный тип, который будет использоваться для целого семейства входов в качестве типа индекса, значения которого применяются для определения индивидуальных входов в семействе. Рассмотрим примеры следующих описаний:

|  |
| --- |
| **task** Anonimous\_Task **is**  **entry** Start;  **end** Anonimous\_Task;  **type** Level **is** (Low, Middle, Hight);  **task type** Simple\_Task **is**  **entry** Read (Value: **out** Integer);  **entry** Request (Level) (Item: **in out** Integer);  **end** Simple\_Task; |

Здесь описание спецификации объекта задачи Anonimous\_Task содержит описание единственного входа Start, который не имеет ни одного параметра.

Спецификация типа задачи Simple\_Task содержит описания двух входов. Вход Read имеет один "**out**"-параметр Value типа Integer. Описание входа Request имеет один "**in out**"-параметр Item типа Integer и использует указание дискретного типа индекса (перечислимый тип Level) представляя, таким образом, целое семейство входов.

**15.2.3 Простое принятие обращений к входам**

Для организации взаимодействия задач, кроме описаний входов, в спецификации задачи-сервера, тело задачи-сервера должно содержать инструкции принятия рандеву, которые описывают действия, выполняемые задачей-сервером в случае вызова рандеву соответствующими входами задачи-сервера.

Для описания инструкций принятия рандеву используется зарезервированное слово **accept**, при этом следует учесть, что инструкции принятия рандеву не могут располагаться в подпрограммах, вызываемых в теле задачи, и не могут быть вложены в другие инструкции принятия рандеву этого же тела задачи. Таким образом, тело задачи может содержать инструкции принятия рандеву только для тех входов, которые указываются в спецификации задачи.

Рассмотрим схематические тела задач, демонстрирующие примеры описания инструкций принятия рандеву (тела задач соответствуют спецификациям задач, которые были показаны при обсуждении описания входов):

|  |
| --- |
| **task body** Anonimous\_Task **is**  **begin**  **accept** Start;  . . .  **end** Anonimous\_Task;  **task body** Simple\_Task **is**  **begin**  . . .  **accept** Read (Value: **out** Integer) **do**  Value := Some\_Value;  **end** Read;  . . .  **accept** Request (Low) (Item: **in out** Integer) **do**  Some\_Low\_Item := Item;  . . .  Item := Some\_Low\_Item;  **end** Request;  **accept** Request (Middle) (Item: **in out** Integer) **do**  Some\_Middle\_Item := Item;  . . .  Item := Some\_Middle\_Item;  **end** Request;  **accept** Request (Hight) (Item: **in out** Integer) **do**  Some\_Hight\_Item := Item;  . . .  Item := Some\_Hight\_Item;  **end** Request;  . . .  **end** Simple\_Task; |

Тело задачи Anonimous\_Task содержит единственную инструкцию принятия рандеву, которая соответствует входу Start, у которого нет параметров. Данная инструкция принятия очень проста и не содержит никакой последовательности инструкций, выполняемой в процессе принятия рандеву.

Следует заметить, что инструкция принятия **accept** может не содержать в себе последовательность инструкций, даже когда соответствующий вход имеет параметры, и наоборот, она может содержать последовательность каких-либо инструкций, если параметры у соответствующего входа отсутствуют. Кроме того, следует заметить, что последовательность инструкций, вложенная в инструкцию принятия рандеву, может содержать инструкцию выхода **return** (это подобно использованию **return** в теле процедуры). Также следует учесть, что формальные параметры, указанные в инструкции принятия рандеву для соответствующего входа, будут локальными для этой инструкции принятия.

Тело задачи Simple\_Task более интересно. Первая инструкция принятия рандеву соответствует входу Read с "**out**"-параметром Value. Внутри этой инструкции принятия рандеву параметру Value присваивается значение переменной Some\_Value (предполагается, что эта и другие переменные были где-либо описаны).

Далее следуют три инструкции принятия рандеву, образующие "семейство". Все они соответствуют описанию входа Request, в спецификации задачи Simple\_Task, которая описывалась с указанием типа Level, значения которого (Low, Middle и Hight) используются в качестве индекса.

**15.2.4 Простой вызов входа**

Задача-клиент осуществляет вызов входа задачи-сервера, идентифицируя как объект задачи-сервера, так и необходимый вход задачи-сервера. Для демонстрации описаний простых вызовов входов задачи-сервера, заданных в задаче-клиенте, рассмотрим следующий пример:

|  |
| --- |
| **declare**  . . .  Simple\_Task\_Variable : Simple\_Task;  . . .  Read\_Value : Integer;  Request\_Item : Integer;  . . .  **begin**  . . .  Anonimous\_Task.Start;  . . .  Simple\_Task\_Variable.Read (Read\_Value);  Simple\_Task\_Variable.Request (Middle) (Request\_Item);  . . .  **end**; |

Как видно из этого примера, простые вызовы входов очень похожи на вызовы процедур. Вызов входа может иметь параметры, значения которых могут передаваться в обоих направлениях между задачей-клиентом и задачей-сервером.

При непосредственной обработке рандеву происходит передача любых параметров "**in**" и "**in out**" в задачу-сервер. Затем задача-сервер выполняет последовательность инструкций, которая расположена внутри инструкции принятия (если такая последовательность существует), а задача-клиент остается в приостановленном состоянии. Далее, после завершения этой последовательности инструкций, происходит передача любых параметров "**out**" и "**in out**" к задаче-клиенту, и обработка рандеву завершается. После этого обе задачи продолжают свое выполнение независимо друг от друга.

Рассмотрим простой пример программы, которая использует свойства рандеву для своеобразного управления последовательностью запуска задач. Следует заметить, что стандарт Ады никак не определяет последовательность запуска множества задач одной и той же программы, и язык не предоставляет стандартных средств для непосредственного управления процессом активации множества задач. Следовательно, разные реализации Ада-систем могут использовать различные правила для определения последовательности активации множества задач, а это означает, что последовательность активации множества задач одной и той же программы может отличаться в различных реализациях Ада-системы. Однако, используя свойства рандеву, можно программно осуществлять взамную синхронизацию выполнения задач. Таким образом, выполнение тела задачи может быть приостановлено в самом начале задачи, то есть до того, как задача начнет выполнять свою непосредственную работу. Такой способ использования рандеву демонстрируется в следующем примере программы:

|  |
| --- |
| **with** Ada.Text\_IO;  **procedure** Multitasking\_Demo\_3 **is**  -- спецификация типа задачи  **task type** Simple\_Task (Message: Character; How\_Many: Positive) **is**    **entry** Start; -- этот вход будет использоваться для реального  -- запуска задачи на выполнение    **end** Simple\_Task;  -- тело задачи  **task body** Simple\_Task **is**  **begin** -- для Simple\_Task  **accept** Start; -- в этом месте, выполнение задачи будет заблокировано  -- до поступления вызова входа  **for** Count **in** 1..How\_Many **loop**  Ada.Text\_IO.Put\_Line("Hello from Simple\_Task " & Message);  **delay** 0.1;  **end loop**;  **end** Simple\_Task;  -- переменные задачного типа  Simple\_Task\_A: Simple\_Task(Message => 'A', How\_Many => 5);  Simple\_Task\_B: Simple\_Task(Message => 'B', How\_Many => 3);  **begin** -- для Multitasking\_Demo\_3  -- в момент, когда управление достигнет этой точки,  -- все задачи начнут свое выполнение,  -- но будут приостановлены в инструкциях принятия рандеву  Simple\_Task\_B.Start;  Simple\_Task\_A.Start;  **end** Multitasking\_Demo\_3; |

Как видно из исходного текста этого примера, здесь описаны два объекта задач: Simple\_Task\_A и Simple\_Task\_B. Каждая задача имеет вход Start, который используется для управления очередностью запуска задач. В результате сначала запускается задача Simple\_Task\_B, а затем Simple\_Task\_A.

Рассмотрим еще один простой пример программы, в котором демонстрируется использование входов с параметрами.

|  |
| --- |
| **procedure** Demo **is**  X : Duration := Duration(Random(100));  Y : Duration;  **task** Single\_Entry **is**  **entry** Handshake(Me\_Wait : **in** Duration; You\_Wait : **out** Duration);  **end task**;  **task body** Single\_Entry **is**  A : Duration := Duration(Random(100));  B : Duration;  **begin**  **Delay** A;  **accept** Handshake(Me\_Wait : **in** Duration; You\_Wait : **out** Duration) **do**  B := Me\_Wait;  You\_Wait := A;  **end** Handshake;  **Delay** B;  **end**;  **begin**  **Delay**(X);  Handshake(X, Y);  **Delay**(Y);  **end** Demo; |

В этом примере, две задачи обмениваются значением длительности задержки выполнения, используя для этого рандеву на входе Handshake задачи Single\_Entry.

Следует заметить, что возможна ситуация, когда задача ожидает рандеву, но при этом нет ни одной задачи, с которой она может его осуществить. В этом случае задача будет аварийно завершена с исключением *Tasking\_Error*.

**15.2.5 Селекция принятия рандеву**

Предположим, что нам необходима задача-сервер, которая будет обслуживать множество задач-клиентов. Предположим также, что задача-сервер должна иметь не один, а целое множество входов для предоставления различных сервисов задачам-клиентам, а один из входов будет указывать на необходимость завершения работы задачи-сервера, то есть задача-сервер должна выполняться до тех пор, пока не получит явной команды на завершение своей работы.

Вполне резонно на основе полученных ранее сведений, попытаться реализовать что-нибудь подобное следующему:

|  |
| --- |
| **task** Server\_Task **is**  **entry** Service\_1 *[ параметры для Service\_1 ]* ;  **entry** Service\_2 *[ параметры для Service\_2 ]* ;  . . .  **entry** Service\_N *[ параметры для Service\_N ]* ;  **entry** Stop;  **end task**;  **task body** Server\_Task **is**  . . .  **begin**  **loop**  **accept** Service\_1 *[ параметры для Service\_1 ]* **do**  . . .  **end** Service\_1;  **accept** Service\_2 *[ параметры для Service\_2 ]* **do**  . . .  **end** Service\_2;  . . .  **accept** Service\_N *[ параметры для Service\_N ]* **do**  . . .  **end** Service\_N;  **accept** Stop **do**  **exit** ; -- выход из цикла, и, следовательно,  -- завершение задачи  **end** Stop;  **end loop**;  **end** Server\_Task; |

Однако при внимательном рассмотрении логики работы данного примера оказывается, что задача-сервер будет блокироваться (приостанавливаться) в каждой инструкции принятия рандеву, ожидая поступления вызова на соответствующем входе от какой-либо задачи-клиента. Причем, находясь в состоянии ожидания, задача-сервер никак не будет реагировать на наличие и поступление вызовов от задач-клиентов на других входах. Следовательно, для ситуаций, которые подобны показанной в этом примере, необходима возможность селекции (или выбора) принятия рандеву на входах задачи-сервера.

Для обеспечения селекции принятия рандеву, Ада предоставляет различные варианты инструкции отбора с ожиданием, которая задается с помощью зарезервированного слова **select**. Использование инструкции отбора в теле задачи-сервера позволяет:

* одновременно ожидать более одного рандеву
* выполнять таймаут, если за указанный период времени не поступило ни одного вызова рандеву
* осуществлять рандеву только в случае наличия вызова рандеву
* завершать выполнение задачи при отсутствии задач-клиентов, которые потенциально могут вызвать рандеву

Рассмотрим следующий пример использования инструкции отбора в теле задачи-сервера:

|  |
| --- |
| **task** Server\_Task **is**  **entry** Service\_1 *[ параметры для Service\_1 ]* ;  **entry** Service\_2 *[ параметры для Service\_2 ]* ;  . . .  **entry** Service\_N *[ параметры для Service\_N ]* ;  **entry** Stop;  **end task**;  **task body** Server\_Task **is**  . . .  **begin**  **loop**  **select**  **accept** Service\_1 *[ параметры для Service\_1 ]* **do**  . . .  **end** Service\_1;  . . . -- дополнительная последовательность инструкций,  -- которая выполняется после принятия рандеву  -- на входе Service\_1  **or**  **accept** Service\_2 *[ параметры для Service\_2 ]* **do**  . . .  **end** Service\_2;  **or**  . . .  **or**  **accept** Service\_N *[ параметры для Service\_N ]* **do**  . . .  **end** Service\_N;  **or**  **accept** Stop;  **exit** ; -- выход из цикла, и, следовательно,  -- завершение задачи  **end select**  **end loop**;  **end** Server\_Task; |

Как видно из исходного текста примера, инструкции принятия рандеву указываются как альтернативы выбора инструкции отбора (это несколько подобно инструкции **case**).

В данном случае при выполнении инструкции отбора задача-сервер циклически "опрашивает" свои входы на наличие вызова рандеву от задач-клиентов (без блокировки в состоянии ожидания вызова рандеву на каком-либо входе). Опрос продолжается до тех пор, пока не будет обнаружен вызов рандеву на каком-либо входе, который соответствует одной из перечисленных инструкций принятия рандеву. После обнаружения вызова выполняется соответствующая альтернатива (примечание: завершение обработки альтернативы приводит к завершению инструкции отбора).

Следует обратить внимание, что если в процессе опроса, выполняемого инструкцией отбора, одновременно появятся два и более вызова рандеву, то инструкция отбора выберет для обработки только один из поступивших вызовов, причем правила выбора альтернативы для такого случая не определяются стандартом. Другими словами, когда инструкция отбора помещена в тело цикла, при одновременном появлении двух и более вызовов рандеву, инструкция отбора будет осуществлять обработку только одного вызова рандеву в течение одного "витка" цикла, а одновременно поступившие вызовы рандеву будут поставлены в очередь порядок которой не определяется стандартом.

В данном примере интересную ситуацию представляет альтернатива принятия рандеву на входе Stop. В этом случае происходит выход из бесконечного цикла выполнения инструкции отбора, что, в свою очередь, приводит к завершению работы задачи-сервера. Если в процессе завершения работы задачи-сервера поступит вызов рандеву на какой-либо из входов Service\_1 - Service\_N, то задача-клиент, вызывающая рандеву, скорее всего получит исключение *Tasking\_Error*. Недостаток такого подхода состоит в том, что завершение работы задачи-сервера требует явного указания вызова рандеву на входе Stop.

Чтобы избавиться от необходимости явного указания вызова рандеву на входе Stop, можно использовать инструкцию отбора, в которой указывается альтернатива завершения задачи:

|  |
| --- |
| **task** Server\_Task **is**  **entry** Service\_1 *[ параметры для Service\_1 ]* ;  . . .  **entry** Service\_N *[ параметры для Service\_N ]* ;  **end task**;  **task body** Server\_Task **is**  . . .  **begin**  **loop**  **select**  **accept** Service\_1 *[ параметры для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **accept** Service\_N *[ параметры для Service\_N ]* **do**  . . .  **end** Service\_N;  **or**  **terminate**; -- завершение работы задачи  **end select**  **end loop**;  **end** Server\_Task; |

В таком случае альтернатива завершения задачи будет завершать работу задачи-сервера, когда отсутствуют вызовы рандеву и отсутствуют задачи-клиенты, которые потенциально способны вызвать рандеву с задачей-сервером. Таким образом, для завершения работы задачи-сервера не требуется явного выполнения никаких специальных действий. Следует также учитывать, что альтернатива завершения работы задачи должна указываться последней в списке альтернатив инструкции отбора.

Может возникнуть ситуация, когда необходимо, чтобы в процессе ожидания вызова рандеву от задач-клиентов задача-сервер выполняла какие-либо дополнительные действия. Для этого можно использовать инструкцию отбора, в которой вместо альтернативы завершения работы используется раздел **else**. Использование такого варианта инструкции отбора может иметь следующий вид:

|  |
| --- |
| **loop**  **select**  **accept** Service\_1 *[ параметры для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **accept** Service\_N *[ параметры для Service\_N ]* **do**  . . .  **end** Service\_N;  **else**  . . . -- последовательность инструкций, которая выполняется  -- если нет ни одного вызова рандеву  **end select**  **end loop**; |

Подобным образом последовательность инструкций, указанная в разделе **else**, может быть использована для организации "фоновой" работы задачи-сервера.

Еще одной разновидностью инструкции отбора служит инструкция отбора, использующая альтернативу таймаута (или задержки). Такая инструкция отбора может иметь следующий вид:

|  |
| --- |
| **loop**  **select**  **accept** Service\_1 *[ параметры для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **accept** Service\_N *[ параметры для Service\_N ]* **do**  . . .  **end** Service\_N;  **or**  **delay** 1.0;  . . . -- последовательность инструкций таймаута,  -- которая выполняется в случае  -- когда нет ни одного вызова рандеву  -- в течение одной секунды  **end select**  **end loop**; |

Подход, демонстрируемый этим примером, удобно использовать для организации "сторожевого таймера", сигнализирующего о некорректной работе программного обеспечения.

Альтернатива таймаута, указанная в инструкций отбора, позволяет задаче-серверу выполнять определенную последовательность действий, если в течение указанного интервала времени не поступило ни одного вызова рандеву. Интервал времени указывается в альтернативах таймаута аналогично интервалу времени в инструкциях задержки выполнения (необходимо однако заметить, что несмотря на внешнее подобие, не следует путать альтернативы таймаута инструкций отбора с инструкциями задержки выполнения). Таким образом, может быть указан относительный (как в примере выше) или абсолютный интервал времени. Если выражение, указывающее относительный интервал времени, имеет отрицательное или нулевое значение или значение абсолютного интервала времени оценивается как прошедшее, то альтернатива таймаута инструкции отбора может быть расценена как эквивалент раздела **else**.

В одной инструкции отбора, допускается наличие более одной альтернативы таймаута. При этом, будет обрабатываться та альтернатива таймаута, которая имеет наименьший интервал времени. Отметим также, что в пределах одной инструкции отбора не допускается совместное использование различных альтернатив таймаута для которых одновременно заданы относительные и абсолютные интервалы времени.

При применении инструкций отбора следует учитывать, что использование альтернативы завершения работы задачи (**terminate**), альтернативы таймаута (**delay**) и раздела **else** в пределах одной инструкции отбора является взаимно исключающим. Кроме того, любой вариант инструкции отбора обязан содержать хотя бы одну альтернативу выбора, в которой указана инструкция принятия рандеву **accept**.

Еще одной особенностью инструкций отбора является опциональная возможность указания дополнительной проверки какого-либо условия для альтернатив принятия рандеву, завершения работы задачи и таймаута. Для указания такой проверки используется конструкция вида: "**when** *условие* =>", - где проверяемое *условие* описывается с помощью логического выражения, результат вычисления которого должен иметь значение предопределенного логического типа Standard.Boolean. Как правило, такую проверку назвают защитной или охранной (*guard*), а ее использование может иметь следующий вид:

|  |
| --- |
| **declare**  . . .  Service\_1\_Counter : Integer;  . . .  Service\_N\_Counter : Integer;  . . .  **begin**  . . .  **loop**  . . .  **select**  **when** (Service\_1\_Counter > 0) =>  **accept** Service\_1 *[ параметры для Service\_1 ]* **do**  . . .  **end** Service\_1;  **or**  . . .  **or**  **when** (Service\_N\_Counter > 100) =>  **accept** Service\_N *[ параметры для Service\_N ]* **do**  . . .  **end** Service\_N;  **end select**  **end loop**;  . . .  **end**; |

Вычисление значения логического выражения осуществляется в начале выполнения инструкции отбора. Если результат вычисления выражения True, то соответствующая альтернатива имеет право быть выбранной инструкцией отбора, если результат False, альтернатива выбрана не будет, причем даже в том случае, когда какая-либо задача-клиент осуществила вызов соответствующего входа и ждет обслуживания. Следует также учесть, что в случае, когда проверки условий используются для всех альтернатив инструкции отбора, и результаты проверок всех условий имеют значение False, это приведет к возбуждению исключения *Program\_Error*.

Альтернативу отбора можно назвать открытой, когда для нее не указана дополнительная проверка условия или когда значение результата проверки условия есть True. В противном случае альтернативу отбора можно назвать закрытой.

**15.2.6 Селекция вызова рандеву**

Инструкции отбора **select** могут использоваться не только для селекции принятия рандеву в задаче-сервере, но и для селекции вызова рандеву в задаче-клиенте. В подобных случаях различные формы инструкций отбора позволяют задаче-клиенту выполнять условный вызов рандеву, временный вызов рандеву или асинхронную передачу управления.

Рассмотрим простой пример инструкции отбора для выполнения условного вызова рандеву на входе Service\_1 задачи-сервера Server\_Task:

|  |
| --- |
| **select**  Server\_Task.Service\_1 *[ параметры для Service\_1 ]* ;  **else**  Put\_Line ("Server\_Task is busy!");  **end select**; |

В этом случае, если задача-сервер Server\_Task не готова к немедленному приему рандеву на входе Service\_1, то вызов рандеву будет отменен и выполнится последовательность инструкций в разделе **else**, которая выводит сообщение "Server\_Task is busy!".

Инструкция отбора, предназначенная для условного вызова рандеву на входе задачи-сервера, как правило, используется для многократного обращения к задаче-серверу. Таким образом, общий вид ее применения может быть следующм:

|  |
| --- |
| **loop**  **select**  Server\_Task.Service\_1 *[ параметры для Service\_1 ]* ;  . . . -- опциональная последовательность инструкций,  **exit**; -- выполняемая после рандеву  **else**  . . . -- последовательность инструкций,  -- выполняемая в случае невозможности  -- осуществления рандеву  **end select**;  **end loop**; |

Инструкция отбора для временного вызова рандеву позволяет задавать ожидание принятия рандеву на входе задачи-сервера в течение заданного интервала времени. Пример такой инструкции для выполнения временного вызова рандеву на входе Service\_1 задачи-сервера Server\_Task может иметь следующий вид:

|  |
| --- |
| **select**  Server\_Task.Service\_1 *[ параметры для Service\_1 ]* ;  **or**  **delay** 5.0;  Put\_Line ("Server\_Task has been busy for 5 seconds!");  **end select**; |

В данном случае, если в течение указанного интервала времени (здесь задан относительный интервал времени длительностью 5 секунд) задача-сервер Server\_Task не приняла рандеву на входе Service\_1, то вызов рандеву отменяется и выполняется последовательность инструкций, заданная альтернативой задержки, которая в этом примере выдаст сообщение "Server\_Task has been busy for 5 seconds!".

Вместо относительного интервала может быть указано абсолютное значение времени:

|  |
| --- |
| **select**  Server\_Task.Service\_1 *[ параметры для Service\_1 ]* ;  **or**  **delay until** Christmas;  Put\_Line ("Server\_Task has been busy for ages!");  **end select**; |

Следует заметить, что когда, в случае относительного интервала времени, указанная задержка будет иметь нулевое или отрицательное значение, или заданное абсолютное значение времени расценивается как прошедшее, выполнение инструкции будет осуществляться подобно выполнению инструкции отбора для условного вызова рандеву.

Инструкция отбора для временного вызова рандеву удобно использовать в качестве сторожевого таймера. Таким образом, ее общий вид может быть следующм:

|  |
| --- |
| **loop**  **select**  Server\_Task.Service\_1 *[ параметры для Service\_1 ]* ;  . . . -- опциональная последовательность инструкций,  **exit**; -- выполняемая после рандеву  **or**  **delay** *интервал\_времени*  . . . -- последовательность инструкций,  -- выполняемая в случае невозможности  -- осуществления рандеву  **end select**;  **end loop**; |

Заметим, что вместо конструкции "**delay** *интервал\_времени*", задающей относительный интервал, может быть использована конструкция "**delay until** *интервал\_времени*", которая указывает абсолютное значение времени.

Инструкция отбора для асинхронной передачи управления (введена стандартом Ada95) позволяет задаче-клиенту продолжить выполнение какого-либо кода в то время, когда вызов рандеву ожидает обслуживания. Пример использования такой инструкции может иметь следующий вид:

|  |
| --- |
| **select**  **delay** 5.0;  Put\_Line ("Server\_Task is not serviced the Service\_1 yet");  **then abort**  Server\_Task.Service\_1 *[ параметры для Service\_1 ]* ;  **end select**; |

В данном случае начинается выполнение инструкций, расположенных между "**then abort**" и "**end select**", то есть вызов рандеву с задачей-сервером Server\_Task на входе Service\_1. При этом, если истечение интервала времени из инструкции **delay** (или **delay until**), расположенной после **select**, произойдет раньше завершения выполнения Server\_Task.Service\_1, то выполнение Server\_Task.Service\_1 принудительно прерывается и выполняется вывод сообщения "Server\_Task is not serviced the Service\_1 yet".

Примечательно, что использование асинхронной передачи управления не ограничено многозадачностью, например:

|  |
| --- |
| **select**  **delay** 5.0;  Put\_Line ("So\_Big\_Calculation abandoned!");  **then abort**  So\_Big\_Calculation;  **end select**; |

В данном случае использование асинхронной передачи позволяет прервать неопределенно долго выполняющуюся или "зависшую" в бесконечном цикле процедуру So\_Big\_Calculation.

При селекции вызова рандеву с помощью инструкций отбора (для осуществления условного вызова рандеву, временного вызова рандеву или асинхронной передачи управления) следует помнить, что если вызванная задача уже завершена, то выполнение инструкции отбора приведет к возбуждению исключению *Tasking\_Error*.

**15.2.7 Идентификация задач и атрибуты**

Каждая задача (объект) Ады имеет свой собственный уникальный идентификатор, с помощью которого она может быть идентифицирована в других задачах. Механизм, с помощью которого задача может получить свой уникальный идентификатор, обеспечивается средствами стандартного пакета *Ada.Task\_Identification*, описанного в приложении *C* (*Annex C*) стандарта Ada95, в котором указываются требования для системного программирования.

Спецификация пакета *Ada.Task\_Identification* имеет следующий вид:

|  |
| --- |
| **package** Ada.Task\_Identification **is**  **type** Task\_ID **is private**;  Null\_Task\_ID : **constant** Task\_ID;  **function** "=" (Left, Right : Task\_ID) **return** Boolean;  **function** Image (T : Task\_ID) **return** String;  **function** Current\_Task **return** Task\_ID;  **procedure** Abort\_Task (T : **in out** Task\_ID);  **function** Is\_Terminated(T : Task\_ID) **return** Boolean;  **function** Is\_Callable (T : Task\_ID) **return** Boolean;  **private**  . . . -- Стандартом языка не определено  **end** Ada.Task\_Identification; |

Кроме этого пакета, в приложении *C* стандарта описываются атрибуты, которые могут быть использованы для идентификации задач:

|  |  |  |
| --- | --- | --- |
| T**'*Identity*** | - | где T: любая задача. Возвращает значение типа Task\_ID которое уникально идентифицирует T. |
| E**'*Caller*** | - | где E: имя любого входа задачи. Возвращает значение типа Task\_ID, которое идентифицирует обрабатываемую в текущий момент задачу, обратившуюся к входу задачи E. Использование этого атрибута допустимо только внутри инструкции принятия (для задачи-сервера). |

При использовании этих средств следует учитывать, что по истечении некоторого времени ничто не гарантирует активность задачи или ее присутствие в области видимости.

Кроме перечисленных средств идентификации, при работе с задачами могут быть также использованы следующие атрибуты:

|  |  |  |
| --- | --- | --- |
| T**'*Callable*** | - | Возвращает значение True, если задача T может быть вызвана. |
| T**'*Terminated*** | - | Возвращает True если выполнение задачи T прекращено. |

Следует учесть, что T - это любая задача а результат, возвращаемый этими атрибутами, имеет предопределенный логический тип Standard.Boolean.

Для входов (и семейств входов) задач определен атрибут **'*Count***. Если E - имя любого входа задачи, то E**'*Count*** возвращает значение типа Universal\_Integer, показывающее число обращений к входу E, которые находятся в очереди. Исползование этого атрибута допустимо только внутри тела задачи или внутри программного модуля, который расположен внутри тела задачи.

Следует заметить, что не рекомендуется, чтобы алгоритм обработки имел жесткую зависимость от значений атрибутов задач **'*Callable*** и **'*Terminated***, а также атрибута **'*Count*** для входов (и семейств входов).

**15.2.8 Разделяемые (общие) переменные**

Ада позволяет задачам совместно использовать разделяемые (общие) переменные (объекты, ресурсы), которые располагаются в общей памяти и могут быть необходимы как для организации взаимодействия задач, так и для хранения каких-либо общих данных. Следовательно, необходимо, чтобы такие переменные допускали возможность чтения/записи из различных задач программы.

Поскольку при выполнении программы возможны ситуации, когда несколько задач пытаются получить доступ к одной и той же разделяемой переменной, во избежание конфликтных ситуаций и поддержки общей корректности работы программы необходима синхронизация доступа к разделяемым переменным. Обеспечение синхронизации доступа к разделяемым переменным представляет собой основополагающую проблему многозадачного (параллельного) программирования, и называется взаимным исключением. Решение этой проблемы основывается на том, что в какой-либо момент времени право на доступ к разделяемой переменной предоставляется только одной задаче. Для обеспечения синхронизации доступа используются специальные программные (или аппаратные) средства: семафоры, критические секции, мониторы и т.д.

В стандарте Ada95 для указания разделяемых переменных используются следующие директивы компилятора:

|  |
| --- |
| **pragma** Atomic ( *Local\_Name* );  **pragma** Atomic\_Components ( *Local\_Array\_Name* );  **pragma** Volatile ( *Local\_Name* );  **pragma** Volatile\_Components ( *Local\_Array\_Name* ); |

Здесь *Local\_Name* указывает локальное имя объекта или описание типа, а *Local\_Array\_Name* указывает локальное имя объекта-массива или описание типа-массива.

Директивы компилятора Atomic и Atomic\_Components обеспечивают непрерывность операций чтения/записи для всех указанных в них объектах. Такие объекты называют атомарными (*atomic*), а операции над ними выполняются только последовательно.

Директивы компилятора Volatile и Volatile\_Components обеспечивают выполнение операций чтения/записи для всех указанных в них объектах непосредственно в памяти.

Примеры применения этих директив компилятора могут иметь следующий вид:

|  |
| --- |
| Array\_Size : Positive;  **pragma** Atomic (Array\_Size);  **pragma** Volatile (Array\_Size);  Store\_Array **is array** (1..Array\_Size) **of** Integer;  **pragma** Atomic\_Components (Store\_Array);  **pragma** Volatile\_Components (Store\_Array); |

Разделяемые переменные и перечисленные для них директивы компилятора могут быть использованы для организации:

* взаимодействия задач
* взаимодействия Ада-программы с другими процессами
* управления устройствами из Ада-программ

Следует заметить, что описанные подобным образом разделяемые переменные не должны использоваться для синхронизации взаимодействия задач, для этого следует использовать средства механизма рандеву и/или защищенные объекты.

**15.3 Защищенные модули (*protected units*)**

**15.3.1 Проблемы механизма рандеву**

Несмотря на обилие возможностей, предоставляемых моделью механизма рандеву Ada83, который предусматривает развитый высокоуровневый подход для синхронизации задач, позволяющий избежать многочисленные методологические сложности, вызванные применением низкоуровневых примитивов, подобных семафорам и сигналам, этот механизм обладает некоторыми недостатками.

Представим себе классический пример, когда необходимо обеспечить взаимодействие двух задач, одна из которых выполняет чтение данных из какого-либо устройства, а вторая - осуществляет обработку этих данных. В литературе подобное взаимоотношение задач, как правило, называют "поставщик-потребитель". Как мы уже знаем, механизм рандеву является одновременно как механизмом синхронизации, так и механизмом межзадачного обмена данными. Необходимость передачи данных между двумя задачами, которые обычно выполняются асинхронно, требует приостановки выполнения задач для осуществления "переговоров". Следовательно, для эффективной организации подобного взаимодействия необходима буферизация данных, передаваемых от одной задачи к другой. Обычным решением в этой ситуации является создание промежуточной задачи, которая служит буфером и управляет потоком данных, а значит, всегда доступна для рандеву с обоими задачами, нуждающимися в обмене данными. При этом следует учесть, что рандеву является относительно длительной операцией, и необходимость наличия дополнительной задачи-буфера приводит к неоправданному снижению производительности.

Исходя из таких рассуждений, гораздо предпочтительнее выглядит возможность, чтобы задача-производитель (та задача, которая читает данные из устройства) могла просто оставлять данные в какой-либо переменной для задачи-приемника (та задача, которая обрабатывает принятые данные), после чего задача-приемник могла бы так же просто эти данные прочитать. Однако, обычная переменная для этой цели не подходит, поскольку необходимо предотвратить модификацию содержимого переменной двумя одновременно выполняющимися задачами.

Еще одним недостатком рандеву является то, что в некоторых ситуациях может возникнуть инверсия абстракции, приводящая к замкнутой взаимной блокировке задач. Кроме того, с методологической точки зрения, идеология механизма рандеву строго ориентирована на управление и находится вне новейших объектно-ориентированных подходов.

Средством, которое позволяет организовать взаимно исключающий доступ к данным из разных, одновременно выполняющихся задач, являются защищенные модули, которые были введены стандартом Ada95. Характерной особенностью защищенных модулей является обеспечение ими синхронизированного доступа к приватным данным, однако в противоположность задачам, которые являются активными сущностями, защищенные модули - пассивны.

**15.3.2 Защищенные типы и объекты  
          Защищенные подпрограммы**

Защищенные модули (типы и объекты) Ады инкапсулируют данные и позволяют осуществлять доступ к ним с помощью защищенных подпрограмм или защищенных входов. Стандарт языка гарантирует, что в результате выполнения кода таких подпрограмм и входов изменение содержимого данных будет производиться в режиме взаимного исключения без необходимости создания дополнительной задачи.

Защищенный модуль может быть описан как защищенный тип или как одиночный защищенный объект, что аналогично одиночной задаче. В последнем случае предполагается, что защищенный объект имеет соответствующий анонимный тип. Следует учитывать, что защищенный тип является лимитированным и, тем самым, не обладает предопределенными операциями присваивания или сравнения.

Подобно задаче или пакету, защищенный модуль имеет спецификацию и тело. Спецификация описывает протокол доступа к защищенному модулю (интерфейс), и может содержать спецификации процедур, функций и входов защищенного модуля. Тело описывает детали реализации протокола доступа к данным защищенного модуля и, как правило, содержит тела защищенных подпрограмм и входов. Подобно задачам и записям, защищенный модуль может иметь дискриминанты дискретного или ссылочного типа.

В качестве простой иллюстрации рассмотрим пример следующго одиночного защищенного объекта:

|  |
| --- |
| -- спецификация защищенного объекта  **protected** Variable **is**  **function** Read **return** Item;  **procedure** Write(New\_Value : Item);  **private**  Data : Item;  **end** Variable;  -- тело защищенного объекта  **protected body** Variable **is**  **function** Read **return** Item **is**  **begin**  **return** Data;  **end**;  **procedure** Write (New\_Value : Item) **is**  **begin**  Data := New\_Value;  **end**;  **end** Variable; |

Защищенный объект Variable предоставляет управляемый доступ к приватной переменной Data типа Item. Функция Read позволяет читать, а процедура Write - обновлять текущее значение переменной Data.

Защищаемые данные и данные о состоянии объекта должны быть помещены в приватную часть спецификации. Смысл этого заключается в том, что приватная часть не доступна клиенту непосредственно, а наличие всей информации в интерфейсе защищенного объекта необходимо компилятору для эффективного распределения памяти.

Защищенные процедуры предусматривают взаимно исключающий доступ к данным защищенного модуля по чтению и/или записи. Защищенные функции предоставляют одновременный доступ к данным защищенного модуля только по чтению, что подразумевает одновременное выполнение множества вызовов функций. Однако вызовы защищенных процедур и защищенных функций остаются взаимно исключающими. Порядок, в котором разные задачи ожидают выполнения защищенных процедур и защищенных функций, стандартом не определяется. Однако поддержка требований приложения *D* (*Annex D*) стандарта Ada95, в котором указаны требования для систем реального времени, позволяет сделать некоторые предположения о возможном порядке выполнения подпрограмм.

Для обращения к защищенным подпрограммам используется традиционная точечная нотация:

|  |
| --- |
| X := Variable.Read;  . . .  Variable.Write (New\_Value => Y); |

Внутри тела защищенного объекта допускается несколько подпрограмм, при этом реализация Ада-системы будет гарантированно осуществлять вызовы подпрограмм по принципу взаимного исключения (подобно монитору).

**15.3.3 Защищенные входы и барьеры**

По аналогии со входами задач, защищенный модуль может иметь защищенные входы. Действия, выполняемые при вызове защищенного входа, предусматриваются в его теле. Защищенные входы подобны защищенным процедурам в том, что они гарантируют взаимно исключающий доступ к данным защищенного модуля по чтению и/или записи. Однако внутри тела защищенного модуля защищенные входы предохраняются логическим выражением, которое называют барьером, а результат вычисления этого логического выражения должен иметь предопределенный логический тип Standard.Boolean.

Если при вызове защищенного входа значение барьера есть False, то выполнение вызывающей задачи приостанавливается до тех пор, пока значение барьера не станет равным True и внутри защищенного модуля будут отсутствовать активные задачи (задачи, которые выполняют тело какого-либо защищенного входа или какой-либо защищенной подпрограммы). Следовательно, вызов защищенного входа может быть использован для реализации условной синхронизации.

Можно заметить, что существует строгая параллель между инструкцией принятия с охранным условием из тела задачи и телом защищенного входа с барьерным условием для защищенного модуля. При этом затраты времени на проверку барьерного условия для защищенного входа значительно меньше, чем затраты времени на проверку охранного условия инструкции принятия.

Хорошим примером решения упоминавшейся ранее проблемы "поставщик-потребитель" служит реализации циклического буфера с помощью защищенного типа:

|  |
| --- |
| -- спецификация защищенного типа  **protected type** Bounded\_Buffer **is**  **entry** Put(X: **in** Item);  **entry** Get(X: **out** Item);  **private**  A: Item\_Array(1 .. Max);  I, J: Integer **range** 1 .. Max := 1;  Count: Integer **range** 0 .. Max := 0;  **end** Bounded\_Buffer;  -- тело защищенного типа  **protected body** Bounded\_Buffer **is**  **entry** Put(X: **in** Item) **when** Count < Max **is**  **begin**  A(I) := X;  I := I **mod** Max + 1; Count := Count + 1;  **end** Put;  **entry** Get(X: **out** Item) **when** Count > 0 **is**  **begin**  X := A(J);  J := J **mod** Max + 1; Count := Count - 1;  **end** Get;  **end** Bounded\_Buffer; |

Здесь предусмотрен циклический буфер, который позволяет сохранить до Max значений типа Item. Доступ обеспечивается с помощью входов Put и Get. Описание объекта (переменной) защищенного типа осуществляется традиционным образом, а для обращения к защищенным входам, как и для обращения к защищенным подпрограммам, используется точечная нотация:

|  |
| --- |
| **declare**  . . .  My\_Buffer: Bounded\_Buffer;  . . .  **begin**  . . .  My\_Buffer.Put(X);  . . .  **end**; |

Заметим, что так же как и в случае вызова входа задачи, вызывающая задача может использовать инструкции отбора для временного или условного вызова защищенного входа. Например:

|  |
| --- |
| . . .  **select**  My\_Buffer.Get(X);  . . . -- список инструкций  **else**  . . . -- список инструкций  **end select**; |

Поведение защищенного типа контролируется барьерами. При вызове входа защищенного объекта выполняется проверка соответствующего барьера. Если значение барьера False, то вызов помещается в очередь, подобно тому, как это происходит при вызове входа задачи. При описании переменной My\_Buffer буфер - пуст, и, таким образом, барьер для входа Put имеет значение True, а для входа Get - False. Следовательно, будет выполняться только вызов Put, а вызов Get будет отправлен в очередь.

В конце выполнения тела входа (или тела процедуры) защищенного объекта производится вычисление значений всех барьеров, у которых есть задачи, ожидающие в очереди, разрешая, таким образом, обработку обращений к входам, которые ранее были помещены в очередь в результате того, что значение барьера было вычислено как False. Таким образом, после завершения обработки первого же вызова Put, если вызов Get уже находится в очереди, то значение барьера для Get будет вычислено заново, и это позволит обслужить ожидающий в очереди вызов Get.

Важно понять, что здесь нет задачи, которая непосредственно ассоциирована с самим буфером. Вычисление барьеров эффективно выполняется системой времени выполнения. Барьеры вычисляются когда вход вызывается впервые и когда происходит что-либо, что может повлиять на состояние барьера ожидающей задачи.

Таким образом, барьеры вычисляются заново только в конце выполнения тела защищенного входа или тела защищенной процедуры, но не в конце выполнения тела защищенной функции, поскольку вызов функции не может повлиять на внутреннее состояние защищенного объекта и, как следствие, не может изменить значения барьеров. Такие правила гарантируют эффективность реализации защищенного объекта.

Следует обратить особое внимание на то, что барьер может ссылаться на значение глобальной переменной. Значение такой переменной может быть изменено независимо от вызова защищенной процедуры или защищенного входа объекта (например, она может быть изменена какой-либо другой задачей или даже в результате вызова защищенной функции), и такие изменения не могут быть обработаны достаточно точно. Поскольку подобные случаи требуют дополнительной внимательности, то рекомендуется воздерживаться от использования глобальных переменных в барьерах.

Необходимо понимать, что защитный механизм барьеров налагается на обычное взаимное исключение защищенной конструкции, предоставляя, таким образом, два различных уровня защиты. В конце защищенного вызова уже находящиеся в очереди обращения к входам (чей барьер теперь имеет значение True) более приоритетны по сравнению с другими защищенными вызовами. С другой стороны, пока защищенный объект занят обработкой текущего вызова (а также любых уже готовых к обработке, но находящихся в очереди вызовов), проверка значения барьера, для вновь поступившего вызова входа, не может быть даже произведена.

Это имеет одно важное следствие: если состояние защищенного объекта изменяется, и существует задача, которая ожидает новое состояние защищенного объекта, то такая задача получает доступ к ресурсу. При этом гарантируется, что состояние ресурса, когда такая задача получает к нему доступ, будет таким же самым, как и в момент принятия решения о предоставлении этой задаче доступа к ресурсу. Таким образом, полностью предотвращаются неудовлетворенные опросы и состязание задач за ресурс.

Основопологающая концепция защищенных объектов подобна мониторам. Они являются пассивными конструкциями с синхронизацией, предусматриваемой системой времени выполнения языка. Однако защищенные объекты, по сравнению с мониторами, обладают большим преимуществом: протокол взаимодействия с защищенными объектами описывается барьерными условиями (в правильности которых достаточно легко убедиться), а не низкоуровневыми и неструктурируемыми сигналами, используемыми в мониторах (как в языке Modula).

Другими словами, защищенные объекты обладают существенными преимуществами высокоуровневых охранных условий модели рандеву, но без дополнительных издержек по производительности, обусловленных наличием дополнительной задачи.

Защищенные модули позволяют осуществлять очень эффективную реализацию различных сигнальных объектов, семафоров и подобных им парадигм:

|  |
| --- |
| -- спецификация защищенного типа  **protected type** Counting\_Semaphore (Start\_Count : Integer := 1) **is**  **entry** Secure;  **procedure** Release;  **function** Count **return** Integer;  **private**  Current\_Count : Integer := Start\_Count;  **end**;  -- тело защищенного типа  **protected body** Counting\_Semaphore **is**  **entry** Secure **when** Current\_Count > 0 **is**  **begin**  Current\_Count := Current\_Count - 1;  **end**;  **procedure** Release **is**  **begin**  Current\_Count := Current\_Count + 1;  **end**;  **function** Count **return** Integer **is**  **begin**  **return** Current\_Count;  **end**;  **end** Counting\_Semaphore; |

Особенностью этого примера является то, что Start\_Count является дискриминантом. При вызове входа Secure опрашивается барьер этого входа. Если результат опроса барьера есть False, то задача ставится в очередь, а ожидание обслуживания становится True.

Следует заметить, что этот пример демонстрирует реализацию общего семафора Дейкстры (*Dijkstra*), где вход Secure и процедура Release соответствуют операциям P и V (*Dutch Passeren* и *Vrijmaken*), а функция Count возвращает текущее значение семафора.

Очевидно, что в очереди обращения к защищенному входу может одновременно находиться несколько задач. Так же как и в случае с очередями к входам задач, очереди к входам защищенных объектов обслуживаются в порядке поступления вызовов (*FIFO - First-In-First-Out*). Однако в случае поддержки требований приложения *D* (*Annex D*) стандарта Ada95, в котором указываются требования для систем реального времени, допускается использование другой дисциплины обслуживания очереди.

Для защищенного типа можно описать семейство защищенных входов; это делается заданием дискретного типа в спецификации входа, подобно описанию семейств входов для задач. Однако, в отличие от задач, нет необходимости предусматривать самостоятельно выделенное тело для входа, принадлежащего такому семейству. Индекс семейства может использовать барьер, ассоциируемый с таким входом (обычно такой индекс используется в качестве индекса массива значений логического типа Standard.Boolean).

**15.3.4 Особенности программирования защищенных входов и подпрограмм**

При программировании действий, выполняемых в телах защищенных входов и подпрограмм, следует учитывать, что время выполнения кода внутри защищенного объекта должно быть настолько кратким, насколько это возможно. Это вызвано тем, что пока выполняется этот код, выполнение других задач, которые пытаются получить доступ к данным защищенного объекта, будет задержано. Ада не позволяет принудительно ограничивать максимальную продолжительность выполнения кода во время защищенных действий, хотя и пытается убедиться в том, что задача не будет заблокирована в состоянии бесконечного ожидания доступа к защищенной процедуре или функции. Потенциальными причинами возникновения подобных ситуаций могут быть попытки выполнения (внутри защищенных действий):

* инструкции отбора (**select**)
* инструкции принятия (**accept**)
* инструкции вызова входа
* инструкции задержки выполнения
* создание или активация задачи

Перечисленные действия называют потенциально блокирующими. Кроме того, обращение к любому потенциально блокирующему действию является также потенциально блокирующим.

Напомним, что при выполнении вызова защищенного входа в процессе обработки вызова защищенной процедуры или защищенного входа осуществляется проверка барьера. Если барьер закрыт (условие барьера имеет значение False), то вызов ставится в очередь. После завершения выполнения тела защищенной процедуры или защищенного входа значения всех барьеров вычисляются заново и, возможно, происходит выполнение тела входа. Вычисление значения барьера для входа и постановка вызова входа в очередь являются защищенными операциями ассоциированного с ними защищенного объекта, и они могут быть названы защищенными действиями.

Любое исключение, возбужденное в процессе вычисления значения барьера для входа, приводит к возбуждению исключения *Program\_Error* во всех задачах, которые в текущий момент находятся в очереди ожидания обслуживания вызова входа защищенного объекта.

**15.3.5 Атрибуты входов защищенных объектов**

Входы защищенных объектов имеют атрибуты, назначение которых подобно назначению атрибутов для входов задач:

|  |  |  |
| --- | --- | --- |
| E**'*Caller*** | - | Возвращает значение типа Task\_ID, которое идентифицирует обрабатываемую в текущий момент задачу, обратившуюся на вход защищенного объекта E. Использование этого атрибута допустимо только внутри тела входа (для защищенного объекта). |
| E**'*Count*** | - | Возвращает значение типа Universal\_Integer, показывающее число обращений на входе E, которые находятся в очереди. |

Здесь, подразумевается, что E - это имя любого входа защищенного объекта. Так же как и в случае задач, при использовании этих средств следует учитывать, что по истечении некоторого времени ничто не гарантирует активность задачи или ее присутствие в области видимости.

**15.4 Перенаправление requeue**

**15.4.1 Проблема предпочтительного управления**

В процессе создания некоторого сложного сервера или планировщика для определения правил очередности предоставления сервиса, предусмотренного телом защищенного входа или телом инструкции принятия, достаточно часто бывает необходимо полагаться на текущие значения различных управляющих элементов. Такие управляющие элементы могут быть локальными для сервера и зависимыми только от внутреннего состояния сервера, они также могут являться атрибутами клиента или управляющей задачи, либо они могут быть глобальными для всей системы в целом. Кроме того, состояние управляющих элементов может изменяться с момента вызова входа (защищенного модуля или задачи) до момента начала обработки обращения к входу.

В самых простых случаях, когда управляющие элементы известны вызывающему клиенту, не изменяются с момента вызова и имеют сравнительно небольшой дискретный диапазон значений, использование возможностей предоставляемых семействами точек входа Ada83, может оказаться вполне достаточным. Однако в тех ситуациях, когда для построения логики обслуживания какого-либо сервиса, необходимо использование внутри сервера предпочтительного управления (*preference control*), такие ограничения, как правило, не соблюдаются.

Примером сервера, который нуждается в использовании предпочтительного управления, может служить сервер, управляющий распределением каких-либо ресурсов. После вызова со стороны клиента подобный сервер должен начать обработку запроса только в том случае, если этот запрос может быть удовлетворен немедленно. В противном случае запрос от клиента должен быть отправлен в очередь для более позднего обслуживания.

Идея альтернативного решения заключается в том, что клиент должен сначала опросить состояние сервера, а затем инициировать вызов с параметрами, которые соответствуют текущему состоянию сервера (предположительно, используя определенный индекс в семействе входов). Однако недостатком такого решения является потенциальная возможность возникновения состязания задач за ресурс, поскольку не гарантируется атомарность операций. Иными словами, между двумя обращениями клиента может произойти обслуживание запроса от другого клиента, что приведет к изменению внутреннего состояния сервера. Таким образом, при последующем обращении клиента может быть утеряна достоверность параметров, значения которых основаны на предыдущем запросе.

В других случаях логика работы сервера нуждается в том, чтобы обслуживание какого-либо сервиса было разделено на два (или более) этапа, и чтобы задача-клиент, запрашивающая сервис, была приостановлена после выполнения первого этапа до тех пор, пока какие-либо условия не позволят выполнить последующий этап обработки. В таком случае необходимы последовательные вызовы двух (или более) входов, но попытки запрограммировать это в Ada83, как правило, затруднительны, поскольку в этом случае также теряется атомарность операций и требуется обеспечить видимость внутреннего протокола работы сервера, что нежелательно.

**15.4.2 Инструкция перенаправления очереди requeue**

С целью поддержки построения необходимых управляющих алгоритмов, позволяющих разделить обработку какого-либо сервиса на несколько (два и более) этапов и для организации предпочтительного управления, стандарт Ada95 ввел единственную и простую инструкцию перенаправления очереди **requeue**.

Инструкция перенаправления очереди используется для завершения инструкции принятия рандеву (**accept**) или тела защищенного входа при необходимости перенаправления соответствующего вызова клиента от одного входа задачи или защищенного модуля к другому входу (или даже к тому же самому входу). Общий вид этой инструкции следующий:

|  |
| --- |
| **requeue** *имя\_входа* *[* **with abort** *]* ; |

Для того чтобы сервер получил доступ к параметрам вызова, нет необходимости возобновлять обработку вызова и требовать от клиента инициирование вызова на другм входе, основываясь на результатах первого вызова. С помощью инструкции перенаправления **requeue**, можно просто переслать вызов клиента в очередь другого входа.

В случае вызова защищенного входа исключительный доступ обеспечивается на протяжении периода проверки параметров и выполнения перенаправления. В случае тела инструкции принятия рандеву (**accept**), задача-сервер управляет своим собственным состоянием, и в данном случае атомарность также обеспечивается, так как она может отвергнуть принятие любого промежуточного вызова.

Инструкция перенаправления **requeue** предназначена для обработки двух основных ситуаций:

* После начала выполнения инструкции принятия **accept** или тела защищенного входа, может быть определено, что запрос не может быть удовлетворен немедленно. Взамен, существует необходимость перенаправлять вызывающего клиента до тех пор, пока его запрос/вызов не сможет быть обработан.
* В качестве альтернативы, часть запроса клиента может быть обработана немедленно, но могут существовать дополнительные шаги, которые должны быть выполнены несколько позже.

В обоих случаях, инструкция принятия **accept** или тело защищенного входа, нуждается в передаче управления. Таким образом, могут быть обработаны запросы от других клиентов или выполнена какая-либо другая обработка. Инструкция перенаправления **requeue** позволяет разделить обработку оригинального запроса/вызова на два (и более) этапа.

Чтобы продемонстрировать логику работы и использования этой инструкции, рассмотрим пример широковещательного сигнала (или события). Задачи приостанавливаются для ожидания некоторого события, а после того как оно происходит, все приостановленные задачи продолжают свое выполнение, а событие очищается. Сложность состоит в том, как предотвратить от попадания в ожидание задачи которые вызвали операцию ожидания после того как событие произошло, но до того как событие очищено. Другими словами, мы должны очистить событие только после того как новые задачи продолжат свое выполнение. Нам позволяет запрограммировать такое предпочтительное управление инструкция перенаправления **requeue**:

|  |
| --- |
| **protected** Event **is**  **entry** Wait;  **entry** Signal;  **private**  **entry** Reset;  Occurred: Boolean := False;  **end** Event;  **protected body** Event **is**  **entry** Wait **when** Occurred **is**  **begin**  **null**; -- пустое тело!  **end** Wait;  **entry** Signal **when** True **is** -- барьер всегда True  **begin**  **if** Wait**'*Count*** > 0 **then**  Occurred := True;  **requeue** Reset;  **end if**;  **end** Signal;  **entry** Reset **when** Wait**'*Count*** = 0 **is**  **begin**  Occurred := False;  **end** Reset;  **end** Event; |

Задачи указывают на то, что они желают ждать событие, выполняя вызов:

|  |
| --- |
| Event.Wait; |

а возникновение события индицируется тем, что какая-либо задача выполняет вызов:

|  |
| --- |
| Event.Signal; |

после чего, все приостановленные в ожидании события задачи продолжат свое выполнение и событие очистится. Таким образом, последующие вызовы входа Wait работают корректно.

Логическая переменная Occurred обычно имеет значение False. Она имеет значение True только тогда, когда задачи возобновляют свое выполнение после приостановки в ожидании события. Вход Wait существует, но фактически не имеет тела. Таким образом, вызывающие задачи приостанавливают свое выполнение в его очереди, ожидая того, что переменная Occurred получит значение True.

Особый интерес представляет вход Signal. Значение его барьера постоянно и всегда равно True, таким образом, он всегда открыт для обработки. Если нет задач ожидающих событие (нет задач вызвавших вход Wait), то его вызов просто ничего не делает. С другой стороны, при наличии задач ожидающих событие, он должен позволить им продолжить выполнение, причем так, чтобы ни одна новая задача не попала в очередь ожидания события, после чего, он должен сбросить флаг, чтобы восстановить управление. Он выполняет это перенаправляя себя на вход Reset (с помощью инструкции перенаправления **requeue**) после установки флага Occurred в значение True, для индикации появления события.

Семантика инструкции перенаправления **requeue** подобна тому, что описано при рассмотрении алгоритма работы Signal. Однако, помните, что в конце обработки тела входа или защищенной процедуры осуществляется повторное вычисление состояний тех барьерных условий в очереди которых находятся задачи, приостановленные в ожидании обслуживания. В этом случае, действительно существуют задачи находящиеся в очереди входа Wait, и существует задача в очереди входа Reset (та задача которая перед эти вызвала вход Signal). Барьер для Wait теперь имеет значение True, а барьер для Reset, естественно, False, поскольку очередь задач на входе Wait не пуста. Таким образом, ожидающая задача способна выполнить тело входа Wait (которое фактически ничего не делает), после чего опять осуществляется повторное вычисление значений барьеров. Этот процесс повторяется до тех пор, пока все приостановленные в ожидании задачи не возобновят свое выполнение и значение барьера для Reset не получит значение True. Оригинальная задача, которая вызвала сигнал, теперь выполняет тело входа Reset, сбрасывая флаг Occurred в значение False, возвращая всю систему в исходное состояние еще раз. Теперь, защищенный объект (как единое целое) полностью освобожден, поскольку нет ни одной ожидающей задачи, ни на одном барьере.

Следует обратить внимание на то, что если какие-либо задачи пытаются вызвать Wait или Signal, когда происходит обработка всего выше описанного процесса, то эти задачи будут заблокированы, поскольку защищенный объект, как единое целое, будет находиться в занятом состоянии. Это иллюстрирует два уровня защиты и является смысловой основой отсутствия возможности появления состязания задач за ресурс.

Другим следствием двухуровневой защиты является то, что все будет работать надежно даже в случае таких сложностей как временные и условные вызовы и принудительное завершение (*abort*). Можно заметить, что это несколько противоположно использованию атрибута **'*Count*** для входов задач, который не может быть использован при временных вызовах.

Следует сделать небольшое замечание о том, что вход Reset описан в приватной части защищенного объекта, и, следовательно, не может быть вызван извне. Ada95 позволяет задачам также иметь приватную часть, содержащую приватные входы.

Следует заметить, что показанный выше пример был приведен только в качестве демонстрации. Внимательный читатель должен заметить, что проверка условия внутри Signal не является строго необходимой. Без этой проверки, вызвавшая задача будет просто всегда осуществлять перенаправление **requeue** и немедленно продолжать вторую часть обработки, при отсутствии ожидающих задач. Однако это условие делает описание более ясным и чистым. Еще более внимательный читатель может заметить, что мы можем запрограммировать этот пример на Ada95 вовсе без использования инструкции перенаправления **requeue**.

Как видно из общего вида инструкции перенаправления **requeue**, она может быть указана с принудительным завершением "**with abort**". В Ada83, после начала рандеву, вызывающий клиент не обладал возможностью отменить свой запрос (таймаут мог быть получен только если запрос не был принят в обработку задачей-сервером). Такое поведение весьма обосновано. После того как сервер начал обработку запроса, он находится в неустойчивом состоянии, и асинхронное удаление вызывающего клиента может нарушить внутренние структуры данных сервера. В дополнение, при наличии параметров переданных по ссылке, сервер, принимающий вызов, должен быть способен получить доступ к данным вызывающего клиента (таким как данные в стеке). При исчезновении вызывающего клиента, это может привести к "висячим" ссылкам и последующему краху работы программы.

Однако, в некоторых случаях, откладывание аннулирования вызова невозможно. В частности, когда значение таймаута необходимо использовать для контроля за общим временем обращения, которое также включает время непосредственной обработки вызова, а не только время ожидания принятия вызова. В дополнение к асинхронной передаче управления Ada95, подобная ситуация может возникнуть когда вызывающий клиент "прерван" и должен изменить последовательность своего выполнения как можно быстрее.

Поскольку не существует единственно возможного наилучшего решения сразу для всех приложений, и поскольку отсутствует легкий обходной путь, то использование инструкции перенаправления (**requeue**) с принудительным завершением (**with abort**) предоставляет программисту возможность выбора для его приложения наиболее подходящего механизма. В основном, когда допускается аннулирование вызова в процессе перенаправления, сервер будет сохранять состояние своих внутренних структур данных перед началом выполнения инструкции перенаправления с принудительным завершением, после чего, если вызывающий клиент будет удален из следующей очереди, то сервер сможет нормально продолжить свою работу. Когда это не возможно, или когда не требуется аннулирование вызова в процессе перенаправления, будет достаточно использования простого перенаправления, и вызывающий клиент будет удерживаться до полного завершения обработки запроса.

Инструкция перенаправления **requeue** позволяет перенаправить вызывающего клиента в очередь того же самого или какого-либо другого входа. При этом, вызывающий клиент не обязан заботиться о таком перенаправлении и даже о фактическом количестве необходимых для удовлетворения его запроса этапов обработки, которые вообще могут быть не видимыми извне защищенного типа или типа задачи. Использование такого эффекта, предоставляет гибкие возможности построения сервера со сложной внутренней архитектурой и простым, однозначным для клиента, интерфейсом.

В процессе выполнения перенаправления, не происходит никаких переопределений параметров. Вместо этого, значения параметров прямо переносятся в новый вызов. Если был предусмотрен новый список параметров, то он может включать ссылки на данные, которые локальны для инструкции принятия **accept** или тела защищенного входа. Это может вызвать некоторые трудности, поскольку выполнение инструкции принятия **accept** или тела защищенного входа будет завершено в результате выполнения инструкции перенаправления **requeue** и локальные переменные будут, таким образом, деаллоцированы (*deallocated*). Необходимо соответствие используемых подтипов между вновь вызываемым целевым входом (если он имеет какие-либо параметры) и текущим входом. Это позволяет использовать то же самое представление для нового множества параметров, когда они передаются по значению (*by-copy*) или по ссылке (*by-reference*), а также исключить необходимость размещения (*allocate*) нового пространства для хранения параметров. Необходимо отметить, что при выполнении перенаправления, кроме передачи тех же самых параметров, существует еще только одна возможность - не передавать никаких параметров вовсе.

В заключение, как общий итог обсуждения инструкции перенаправления **requeue** и логики ее использования, сделаем следующие выводы:

* Инструкция перенаправления **requeue** допустима внутри тела защищенного входа или внутри инструкции принятия рандеву **accept**. Целевой вход (допустим тот же самый вход) может быть в той же самой задаче или том же защищенном объекте, или другой задаче или другом защищенном объекте. Возможно использование любых перечисленных комбинаций.
* Любые фактические параметры оригинального вызова передаются к новому входу. Следовательно, новый вход должен иметь такой же самый профиль параметров или не иметь никаких параметров.

**15.5 Цикл жизни задачи**

До сих пор, во всех примерах, мы подразумевали только то, что создание задач осуществляется на этапе элаборации, и никак не рассматривали, что происходит с задачами в процессе их существования. Однако, при построении сложных многозадачных программ, необходимо понимание из чего состоит цикл жизни задачи, то есть, как происходит создание, активация и завершение одной отдельно взятой задачи.

**15.5.1 Создание задачи**

Напомним, что тип задачи может рассматриваться как шаблон для создания реальных объектов-задач. Типы и объекты задач могут быть описаны в любой описательной части, включая тело задачи. Для любого типа задачи, спецификация и тело должны быть описаны вместе, в одном и том же модуле (тело задачи, как правило, помещается в конце описательной части).

Объект задачи может быть создан в процессе элаборации описания какого-либо объекта, расположенного где-либо в описательной части, или в процессе обработки аллокатора (выражение в виде "**new** ..."). Все задачи, которые созданы в результате элаборации описаний объектов, расположенных в одной описательной части (включая внутренние компоненты описываемых объектов), активируются вместе. Подобным образом, все задачи созданные в процессе обработки одиночного аллокатора, также активируются вместе.

Выполнение объекта-задачи имеет три основные фазы:

1. ***Активация*** - элаборация описательной части тела задачи, если она есть (локальные переменные для тела задачи создаются и инициализируются в процессе активации задачи). *Активатор* идентифицирует задачу, которая создала и активировала задачу.
2. ***Нормальное выполнение*** - выполнение инструкций, видимых внутри тела задачи.
3. ***Завершение*** - выполнение любого кода завершения (*finalization*), ассоциированного с любым объектом в описательной части задачи.

Вновь созданная задача находится в ***неактивированом*** (*unactivated*) состоянии. Затем, система времени выполнения осуществляет ассоциирование с этой задачей потока управления (*thread of control*). Если элаборация задачи терпит неудачу, то задача сразу переходит в ***прекращенное*** (*terminated*) состояние. В противном случае, задача переходит в ***работоспособное*** (*runnable*) состояние и начинает выполнять код инструкций тела задачи. Если этот код выполняет некоторые операции, которые блокируют выполнение задачи (рандеву, защищенные операции, задержки выполнения...), то задача переходит в ***приостановленное*** (*sleep*) состояние, а затем возвращается обратно в ***работоспособное*** состояние. Когда задача выполняет альтернативу завершения (*terminate alternative*) или нормальным образом завершает свое выполнение, она переходит в ***прекращенное*** (*terminated*) состояние.

Задача индицирует свою готовность к началу завершения выполнением инструкции **end**. Кроме того, задача может начать процесс своего завершения в результате необработанного исключения, в результате выполнения инструкции отбора **select** с альтернативой завершения или в случае выполнения инструкции принудительного завершения **abort**. Задача, которая окончила свою работу называется ***завершенной*** (*completed*) или ***прекращенной*** (*terminated*), в зависимости от наличия активных задач которые от нее зависят.

Каждая задача имеет ***владельца*** (*master*), которым может являться задача, подпрограмма, инструкция блока или инструкция принятия рандеву **accept**, содержащая описание объекта-задачи (или, в некоторых случаях, ссылочного типа который ссылается на тип-задачи). Говорят, что задача ***зависит*** от своего владельца.

Задачу, которая осуществляет выполнение владельца, называют ***ведущей*** или ***родительской*** задачей (*parent task*). Таким образом, для каждой задачи существует родительская задача от которой она также зависит.

Приняты следующие правила:

* Если задача была описана как объект, то ее родительская задача - это та задача, которая содержит описание объекта задачи.
* Если задача была описана как часть аллокатора **new**, то ее родительская задача - это та задача, которая содержит соответствующее описание ссылочного типа.

Когда родительская задача создает новую, ***дочернюю*** задачу (*child task*), ее выполнение приостанавливается на время активации ее дочерней задачи (немедленно, если дочерняя задача создана аллокатором, или после завершения процесса элаборации соответствующей описательной части). Как только все дочерние задачи завершают свою активацию, родительская задача и все ее дочерние задачи продолжают свое выполнение независимо. Если задача, в процессе своей активации, создает другую задачу, то, перед тем как продолжить свое выполнение, она также должна ждать пока активируется ее дочерняя задача.

Существует концептуальная задача, называемая ***задачей окружения*** (*environment task*), которая ответственна за элаборацию всей программы. В общем случае, задача окружения - это поток управления операционной системы, который осуществляет инициализацию системы времени выполнения и выполняет головную подпрограмму Ады. Перед вызовом головной подпрограммы Ада-программы, задача окружения осуществляет элаборацию всех библиотечных модулей, указанных в спецификаторе **with** головной подпрограммы. В результате процесса элаборации происходит создание и активация задач уровня библиотеки до вызова головной подпрограммы Ада-программы.

**15.5.2 Активация задачи**

Для процесса активации задач приняты следующие правла:

1. Для статических задач, активация начинается немедленно после окончания процесса элаборации описательной части в которой эти задачи описаны.
2. Первая инструкция, следующая за описательной частью, не будет выполняться до тех пор, пока все созданные задачи не окончат свою активацию.
3. Задаче, перед выполнением своего тела, не требуется ожидать активацию других задач, которые созданны одновременно с ней.
4. Задача может попытаться осуществить взаимодействие с другой задачей, которая уже создана, но еще не активирована. При этом, выполнение вызывающей задачи будет задержано до тех пор, пока не произойдет взаимодействие.
5. Если объект-задачи описан в спецификации пакета, то он начнет свое выполнение после окончания процесса элаборации описательной части тела пакета.
6. Динамические задачи активируются немедленно после завершения обработки создавшего их аллокатора.
7. Задача, которая выполнила инструкцию, ответственную за создание задач, будет заблокирована до тех пор, пока созданные задачи не окончат свою активацию.
8. Если в процессе элаборации описательной части возбуждено исключение, то любая задача, созданная в процессе элаборации становится ***прекращенной*** и никогда не активируется. В виду того, что на этом этапе задача сама по себе не может обрабатывать исключения, модель языка требует, чтобы такую ситуацию обрабатывала родительская задача или владелец задачи, поскольку будет возбуждено предопределенное исключение *Tasking\_Error*.
   * В случае динамического создания задач, возбуждение исключения осуществляется после инструкции которая содержит аллокатор. Однако, если вызов аллокатора находится в описательной части (как часть инициализации объекта), то описательная часть, в результате неудачи, не выполняется, а возбуждение исключения происходит в объемлющем блоке (или вызвавшей подпрограмме).
   * В случае статического создания задачи, возбуждение исключения осуществляется перед выполнением первой исполняемой инструкции, которая следует сразу после описательной части. Это исключение возбуждается после того как осуществлена активация всех созданных задач, вне зависимости от успешности их активации, и оно, в большинстве случаев, вызывается однократно.
9. Атрибут задачи **'*Callable*** возвращает значение True только в случае, когда указанная задача не находится в состоянии ***завершенная*** или ***прекращенная***. Любая задача находящаяся в ***ненормальном*** (*abnormal*) состоянии является принудительно завершенной задачей. Атрибут задачи **'*Terminated*** возвращает значение True когда указанная задача находится в ***прекращенном*** состоянии.

**15.5.3 Завершение задачи**

Перед выходом, ***владелец*** (*master*) осуществляет выполнение конструкций, которые включают очистку (*finalization*) локальных объектов после их использования (и после ожидания всех локальных задач). Каждая задача зависит от одного владельца, или более:

* Если задача была создана в результате обработки аллокатора ссылочного типа, то она зависит от каждого владельца, который включает элаборацию описаний самого отдаленного предка указанного ссылочного типа.
* Если задача была создана в результате элаборации описания объекта, то она зависит от каждого владельца, который включает эту элаборацию.

Кроме того, если задача зависит от указанного владельца, то считается, что она также зависит от задачи, которая выполняет этого владельца, и (рекурсивно) от любого владельца такой задачи.

В первую очередь, зависимая задача дожидается очистки (*finalization*) владельца. Затем, осуществляется очистка каждого объекта с таким же уровнем вложенности как и у очищаемого владельца, если этот объект был успешно проинициализирован и до сих пор существует. Примечательно, что любой объект, уровень вложенности которого глубже чем у владельца, уже не должен существовать, поскольку он должен был очиститься внутренним владельцем. Таким образом, после выхода из владельца, не очищенными остаются только те объекты, уровень вложенности которых меньше чем у владельца.

**15.6 Прерывания**

Ада позволяет программисту ассоциировать определяемые пользователем обработчики прерываний с некоторыми прерываниями системы. Хотя обработчик прерываний может быть защищенной процедурой или входом задачи, ассоциация с входом задачи, в настоящее время, считается устаревшим средством языка. Таким образом, внимание будет сфокусировано на определяемых пользователем обработчиках прерываний на основе защищенных процедур.

Следует заметить, что в зависимости от решаемой задачи, прерывания Ады могут ассоциироваться с внешними событиями, которые могут являться как непосредственными аппаратными прерываниями, так и определенными событиями используемой операционной системы. В последнем случае, такие события, вне Ада-системы, могут носить название сигналов.

**15.6.1 Модель прерываний Ады**

Стандарт описывает следующую модель прерывания:

* Прерывание представляет класс событий, которые детектируются оборудованием или системным программным обеспечением.
* ***Появление*** (*occurence*) прерывания состоит из ***генерации*** (*generation*) и ***доставки*** (*delivery*).
* Генерация прерывания - это событие в оборудовании или системе, которое делает прерывание доступным для программы.
* Доставка прерывания - это действия, которые вызывают часть программы (называемую ***обработчиком*** прерывания) в ответ на появление прерывания. Прерывание называют ***ожидающим обслуживание*** (*pending*), когда оно находится между генерацией и доставкой прерывания. Вызов обработчика происходит только после доставки каждого прерывания.
* Пока происходит обработка прерывания, последующие прерывания от того же источника ***заблокированы***, то есть, предотвращается генерация всех последующих прерываний. Будут ли утеряны заблокированные прерывания, обычно, зависит от устройства.
* Некоторые прерывания ***зарезервированы***. Программист не может определить обработчик для зарезервированного прерывания. Обычно, зарезервированные прерывания непосредственно обрабатываются библиотекой времени выполнения Ады (например, прерывание от часов, которое используется для реализации инструкций задержки).
* Каждое не зарезервированное прерывание имеет обработчик по умолчанию, который устанавливается библиотекой времени выполнения Ады.

**15.6.2 Защищенные процедуры обработки прерываний**

Ада предусматривает два стиля установки обработчиков прерываний: ***вложенный*** (*nested*) и ***не-вложенный*** (*non-nested*). При ***вложенном*** стиле, обработчик прерываний в защищенном объекте устанавливается неявно в момент появления данного защищенного объекта, и после того как защищенный объект прекращает существовать, происходит неявное восстановление старого обработчика прерывания. При ***не-вложенном*** стиле, обработчик прерываний устанавливается явным вызовом процедуры, и старый обработчик прерывания может быть восстановлен только явно.

Существует два способа определения устанавливаемого обработчика прерывания вложенного стиля, в зависимости от используемой в защищенном описании директивы компилятора.

В первом случае, устанавливаемый с использованием вложенного стиля обработчик прерывания, идентифицируется последующей директивой компилятора Attach\_Handler, которая появляется в защищенном описании и имеет следующий вид:

|  |
| --- |
| **pragma** Attach\_Handler (Handler, Interrupt); |

Здесь, Handler указывает имя защищенной процедуры без параметров, которая описана в защищенном описании, а Interrupt является выражением типа Interrupt\_ID. Защищенное описание должно выполняться на уровне библиотеки, то есть, оно не может быть вложено в тело подпрограммы, тело задачи или инструкцию блока. Однако, если защищенное описание описывает защищенный тип, то индивидуальный защищенный объект этого типа может быть размещен в этих местах. Динамическое размещение с помощью аллокатора **new** обладает большей гибкостью: аллокация защищенного объекта с обработчиком прерывания устанавливает обработчик прерывания, ассоциированный с этим защищенным объектом, а деаллокация защищенного объекта восстанавливает ранее установленный обработчик. Выражение типа Interrupt\_ID не обязано быть статическим. В частности, его значение может зависить от дискриминанта защищенного типа. Например:

|  |
| --- |
| **package** Nested\_Handler\_Example **is**  **protected type** Device\_Interface  (Int\_ID : Ada.Interrupts.Interrupt\_ID) **is**  **procedure** Handler;  **pragma** Attach\_Handler (Handler, Int\_ID);  **end** Nested\_Handler\_Example; |

Во втором случае, устанавливаемый с использованием вложенного стиля обработчик прерывания, идентифицируется последующей директивой компилятора Interrupt\_Handler, которая появляется в защищенном описании и имеет следующий вид:

|  |
| --- |
| **pragma** Interrupt\_Handler (Handler, Interrupt); |

Здесь, также как и в первом случае, Handler указывает имя защищенной процедуры без параметров, которая описана в защищенном описании, а Interrupt является выражением типа Interrupt\_ID. Как и в случае использования директивы Attach\_Handler, защищенное описание не может быть вложено в тело подпрограммы, тело задачи или инструкцию блока. Кроме того, эта директива компилятора имеет одно дополнительное ограничение: если защищенная процедура описана для защищенного типа, то объекты этого типа также не могут быть расположены в этих местах. Следовательно, они должны создаваться динамически, с помощью **new**.

**15.6.3 Пакет *Ada.Interrupts***

***Не-вложенная*** установка и удаление обработчиков прерываний полагается на дополнительные средства стандартного пакета *Ada.Interrupts* спецификация которого имеет следующий вид:

|  |
| --- |
| **package** Ada.Interrupts **is**  **type** Interrupt\_ID **is** *Определяется\_Реализацией*;  **type** Parameterless\_Handler **is access protected procedure**;  **function** Is\_Reserved (Interrupt : Interrupt\_ID)  **return** Boolean;  **function** Is\_Attached (Interrupt : Interrupt\_ID)  **return** Boolean;  **function** Current\_Handler (Interrupt : Interrupt\_ID)  **return** Parameterless\_Handler;  **procedure** Attach\_Handler  (New\_Handler : **in** Parameterless\_Handler;  Interrupt : **in** Interrupt\_ID);  **procedure** Exchange\_Handler  (Old\_Handler : **in out** Parameterless\_Handler;  New\_Handler : **in** Parameterless\_Handler;  Interrupt : **in**  Interrupt\_ID);  **procedure** Detach\_Handler  (Interrupt : **in** Interrupt\_ID);  **function** Reference (Interrupt : Interrupt\_ID)  **return** System.Address;  **private**  . . . -- стандартом не определено  **end** Ada.Interrupts; |

Процедура Attach\_Handler используется для установки соответствующего обработчика прерывания, переопределяя любой существующий обработчик (включая обработчик пользователя). Если параметр New\_Handler - **null**, то осуществляется восстановление обработчика по умолчанию. Если параметр New\_Handler указывает защищенную процедуру для которой не была применена директива компилятора Interrupt\_Handler, то возбуждается исключение *Programm\_Error*.

**15.5.4 Приоритеты**

Для установки приоритета защищенного объекта может быть использована директива компилятора Interrupt\_Priority, которая имеет следующий вид:

|  |
| --- |
| **pragma** Interrupt\_Priority ( *expression* ); |

Отсутствие выражения *expression* воспринимается как установка максимального системного приоритета (Interrupt\_Priority**'*Last***). Пока обрабатываются операции этого защищенного объекта, прерывания с равным или низшим приоритетом будут заблокированы. Во избежание возникновения ситуации инверсии приоритетов, любая задача, вызывающая операции этого защищенного объекта, должна устанавливать свой приоритет в приоритет этого защищенного объекта на время выполнения операции, отражая срочность завершения операции. Благодаря этому прерывания становятся не блокируемыми. Любой обработчик прерывания обрабатывается с приоритетом своего защищенного объекта, который может быть выше чем приоритет прерывания, если этот же обработчик защищенного объекта обрабатывает более одного вида прерывания. В дополнение к этому, для динамического изменения приоритера, может быть использована процедура Set\_Priority, расположенная в пакете *Ada.Dynamic\_Priorities*.