Gem #92: Архетипы кода программирования в реальном времени - Часть 2. Простая Спорадическая Задача

Автор: Marco Panunzio, University of Padua

Краткое содержание: В ходе программирования систем, работающих в реальном времени, код, обеспечивающий параллелизм и собственно работу в реальном времени часто следует явно или неявно определённым образцам. Такие образцы можно собрать в типовые архетипы и применять в случае необходимости. В этой серии публикаций будет описан набор таких архетипов, которые значительно облегчают разработку систем реального времени.

Введение.

В предыдущем Gem #89 в этой серии мы ввели ключевые понятия, лежащие в основе наших архетипов кода, и описали простейший из наших архетипов, который реализует циклическую задачу. В этом Gem #92 мы показываем, как реализовать одинаково простую спорадическую задачу. В следующем Gem в этой серии мы удалимся от этого уровня простоты, чтобы реализовать полный архетип, который преодолевает некоторые из ограничений, присущих этим первоначальным решениям.

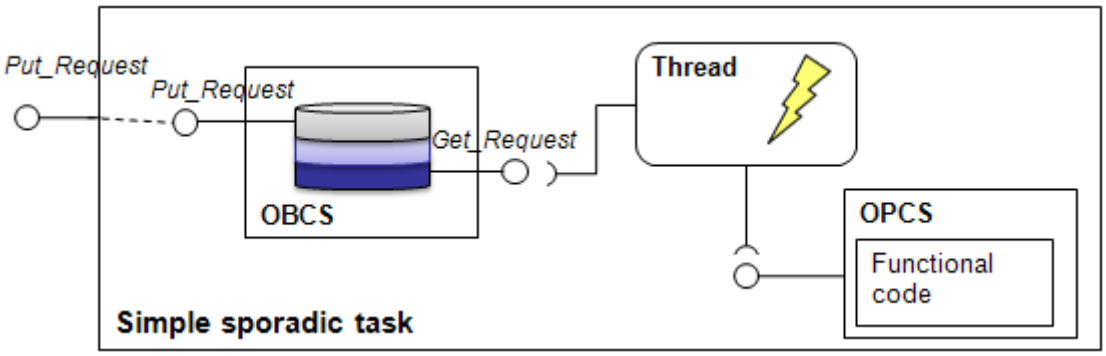
Давайте начнём…

Простая Спорадическая Задача

Спорадическая задача – это задача, любая из двух последующих активаций которой всегда разделяется не менее чем на минимальный гарантированный промежуток времени. Это минимальное разделение обычно называется минимальным временем прибытия (minimum inter-arrival time MIAT). В этом исходном архетипе задача выполняет одну операцию при каждой активации, и она делает это в ответ на запрос, выданный внешним клиентом.

Подобно циклической задаче, наша спорадическая задача состоит из: (i) защищённого объекта, совместно используемого с внешним миром, который внешние клиенты вызывают для отправки своих запросов на выполнение. В предыдущем Gem мы называли этот ресурс OBCS; и (ii) поток управления, который ожидает входящие запросы, извлекает первый из них из защищённого объекта и выполняет спорадическую операцию, которая соответствует определённой операции, как предусмотрено OPCS.

Структура задачи, код которой мы собираемся показать, показана на рисунке ниже.



К счастью, Язык Ada хорошо оборудован для реализации этой структуры, и мы реализуем это в нашем архетипе, используя защищённый объект в соответствии с политикой Ceiling_Locking для (i) и тип задачи для (ii).

На данный момент мы иллюстрируем базовую версию архетипа для спорадической задачи. В объяснении мы также проиллюстрируем некоторые из его ограничений, которые не будут присутствовать в более сложной версии архетипа.

Ниже приводится Спецификация простой спорадической задачи:

```
with System;

generic
   with procedure Sporadic_Operation;
   Ceiling : System.Priority;
   OBCS_Size : Integer;
package Simple_Sporadic_Task is

   procedure Put_Request;

   task type Thread_T
     (Thread_Priority : System.Priority; Interval : Integer)
   is
      pragma Priority (Thread_Priority);
   end Thread_T;

end Simple_Sporadic_Task;
```

Спецификация определяет (то же что и для циклической задачи, которую мы представили в предыдущем Gem) Тип задачи внутри универсального пакета. При создании экземпляра пакета мы указываем спорадическую операцию для задачи, приоритет потолка для защищённого объекта OBCS и размер очереди запросов OBCS.

Кроме того, мы создаём процедуру Put_Request, которая используется клиентами для отправки запроса в спорадическую задачу.

Тело пакета отличается:

```
with System_Time;
with System_Types;
with Ada.Real_Time;

package body Simple_Sporadic_Task is

   Protocol : System_Types.Simple_Sporadic_OBCS (Ceiling, OBCS_Size);

   procedure Put_Request is
   begin
      Protocol.Put_Request;
   end Put_Request;

   task body Thread_T is

      use Ada.Real_Time;
      Next_Time : Time := System_Time.System_Start_Time +
        System_Time.Task_Activation_Delay;
      MIAT : Time_Span := Milliseconds (Interval);
      Release : Time;

   begin
```

```
      loop
         delay until Next_Time;
         Protocol.Get_Request (Release);
         Next_Time := Release + MIAT;
         Sporadic_Operation;
      end loop;
   end Thread_T;

end Simple_Sporadic_Task;
```

Сравнивая это тело с телом циклической задачи, появляются два основных различия: (i) наличие OBCS; и (ii) слегка изменённая структура цикла.

Как и в циклической задаче, спорадическая задача входит в бесконечный цикл и приостанавливается до общесистемного времени начала. После этого: (i) он вызывает запрос к задаче `Get_Request (Time)` OBCS; (ii) после обработки запроса (из которой, как мы покажем ниже, она получает временную метку, когда релиз действительно произошёл), задача выполняет `Sporadic_Operation` (одиночный, на данный момент), указанный при создании своего общего пакета; (iii) он вычисляет следующее раннее время выпуска (`Next_Time`), чтобы соблюдать минимальное разделение между последующими активациями. Поэтому на следующей итерации цикла задача выдаёт запрос на абсолютную приостановку до этого времени и, таким образом, не будет проверять запросы OBCS для выполнения, пока не истечёт требуемое минимальное разделение.

В заключение, когда вызывается процедура `Put_Request`, она просто выполняет простое косвенное обращение к процедуре OBCS с тем же именем. Чтобы оценить это, мы должны взглянуть на OBCS, который действует как агент синхронизации для задачи.

Спецификация OBCS следующая:

```
with System;
with Ada.Real_Time; use Ada.Real_Time;

package System_Types is

   protected type Simple_Sporadic_OBCS (C : System.Priority; Size : Integer) is
      pragma Priority(C);
      procedure Put_Request;
      entry Get_Request (Release_Time : out Time);
   private
      Max_Pending : Integer := Size;
      START_Pending : Integer := 0;
      Barrier : Boolean := False;
   end Simple_Sporadic_OBCS;

end System_Types;
```

Спецификация OBCS объявляет процедуру `Put_Request.` Эта процедура используется для размещения запросов в своей очереди. OBCS также объявляет и защищённый запрос к задаче `Get_Request (Time)`, который используется потоком для выборки запросов. В закрытой части объявления атрибут `Max_Pending` используется для задания максимального числа ожидающих запросов, которое может содержать OBCS (очевидно, не больше его размера); атрибут `START_Pending` указывает фактическое число ожидающих запросов; наконец, логический барьер используется для управления защитой `Get_Request(Time).`

```
package body System_Types is

   protected body Simple_Sporadic_OBCS is
```

```
      procedure Update_Barrier is
      begin
         Barrier := Start_Pending > 0;
      end Update_Barrier;

      procedure Put_Request is
      begin
         if Start_Pending < Max_Pending then
            Start_Pending := Start_Pending + 1;
         end if;
         Update_Barrier;
      end Put_Request;

      entry Get_Request (Release_Time : out Time) when Barrier is
      begin
         Release_Time := Ada.Real_Time.Clock;
         Start_Pending := Start_Pending - 1;
         Update_Barrier;
      end Get_Request;

   end Simple_Sporadic_OBCS;

end System_Types;
```

Тело OBCS довольно легко понять. При вызове процедуры `Put_Request` количество ожидающих запросов (`START_Pending`) увеличивается, до тех пор, пока максимальное число запросов не достигнуто. В случае достижения максимума новый запрос просто игнорируется.

Запись `Get_Request (Time)` используется задачей для проверки OBCS для ожидающих запросов. В случае, когда есть запросы и охраняющий задачу `Barrier` открыт то тогда задача: (i) сохраняет отметку времени выполнения запроса (которая условно совпадает с запуском задачи), и которая позже используется для расчёта времени следующего запуска задачи; и (ii) уменьшает количество ожидающих запросов.

В конце `Put_Request` и `Get_Request` значение защиты `Barrier` обновляется с помощью `Update_Barrier`. Если ожидающих запросов больше нет, для параметра `Barrier` устанавливается значение `false`. Защита откладывает запуск задачи, вызов задачи блокируется, пока не будет обработан новый запрос к задаче.

Проверка для очереди запроса, что очередь не является пустой, непосредственно не используется в качестве защитного выражения для запроса, чтобы соответствовать ограничению профиля Ravenscar, который требует, чтобы защита были простыми Boolean условиями и таким образом имели детерминированную оценку. OBCS имеет единственную точку обработки запроса, поскольку профиль Ravenscar это требует, и единственная задача, которая может быть поставлена в очередь на нем, является задачей, к которой принадлежит OBCS, таким образом гарантируя полное соответствие профилю Ravenscar.

В то время как предлагаемая структура достигает нашей цели создания спорадической задачи, мы сразу замечаем два потенциальных недостатка: `Sporadic_Operation` не имеет параметров, и протокол синхронизации очень, возможно, слишком прост для удовлетворения реальных потребностей системы.

Что касается первой проблемы, клиенты спорадической задачи просто переключаются на новые разновидности задачи, но не могут, например, передавать данные задаче как параметры запроса. Создание нетривиального шаблона типа сотрудничества производитель-потребитель с этой структурой задач невозможно, поскольку задача-потребитель (наша спорадическая задача) не может получить какие-либо данные для обработки.

Это связано с тем, что OBCS, в этой версии это простой счётчик ожидающих запросов.

В следующих Gems, этой серии, мы покажем, как поддерживать спорадические операции с параметрами. Начнём реализовывать более сложные политики очередей для запросов на выполнение.

Связанный со статьёй текст программы

Attached Files отсутствуют

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Сведения об авторе отсутствуют

Last Updated: 10/13/2017
Posted on: 10/11/2010

Обсуждение…