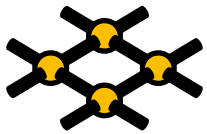
Vláknové programování
část XI

Lukáš Hejmánek, Petr Holub

{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192

2012–05–15

Přehled přednášky

Problém řízení přístupu ke zdrojům

- Aspekty synchronizace požadavků (Bloom, 1979)
  1. typ požadované služby
  2. pořadí požadavků
  3. interní stav přijímajícího vlákna
  4. priorita volajícího
  5. parametry požadavků

Problém řízení přístupu ke zdrojům

- Definice příkladu problému:
  - n zdrojů (vidliček v příborníku)
  - každé vlákno si může požadovat alokaci $1 \ldots m$ zdrojů (vidliček) kde $m \leq n$
- Možné řešení:
  - pollování (zodpovědnost vlákna)
  - rodiny entries (entry families) pro malé m
  - podpora přístupu k `in` parametrům předávaným v rámci volání rendezvous
    - není v Adě podporováno
    - implementováno např. v jazce SR (Synchronizing Resources)
    - problém s efektivitou implementace (bariéra se musí vyhodnocovat při každém řazení vlákna do fronty entry, nikoli jen jednou per entry)
  - `requeue`

Rodiny entries

- Úplná formální signatura entry

```
accept Entry_Name(Family_Index)(P : Parameters) do
-- sequence of statements
exception
-- exception handling part
end Entry_Name;
```

- Rodiny entries
  - např. prioritizace a odlišení volajících vláken

```
task Multiplexer is
   entry Channel(1..3)(X : in Data);
end Multiplexer;
```

Rodiny entries

- Příklad použití s vláknem
  - multiplexer vynucuje pořadí vstupů cyklicky 1, 2, 3

```
1  task body Multiplexer is
   begin
3     loop
         for I in 1..3 loop
5           accept Channel(I)(X : in Data) do
               -- consume input data on channel I
7           end Channel;
         end loop;
9     end loop;
   end Multiplexer;
```

Rodiny entries

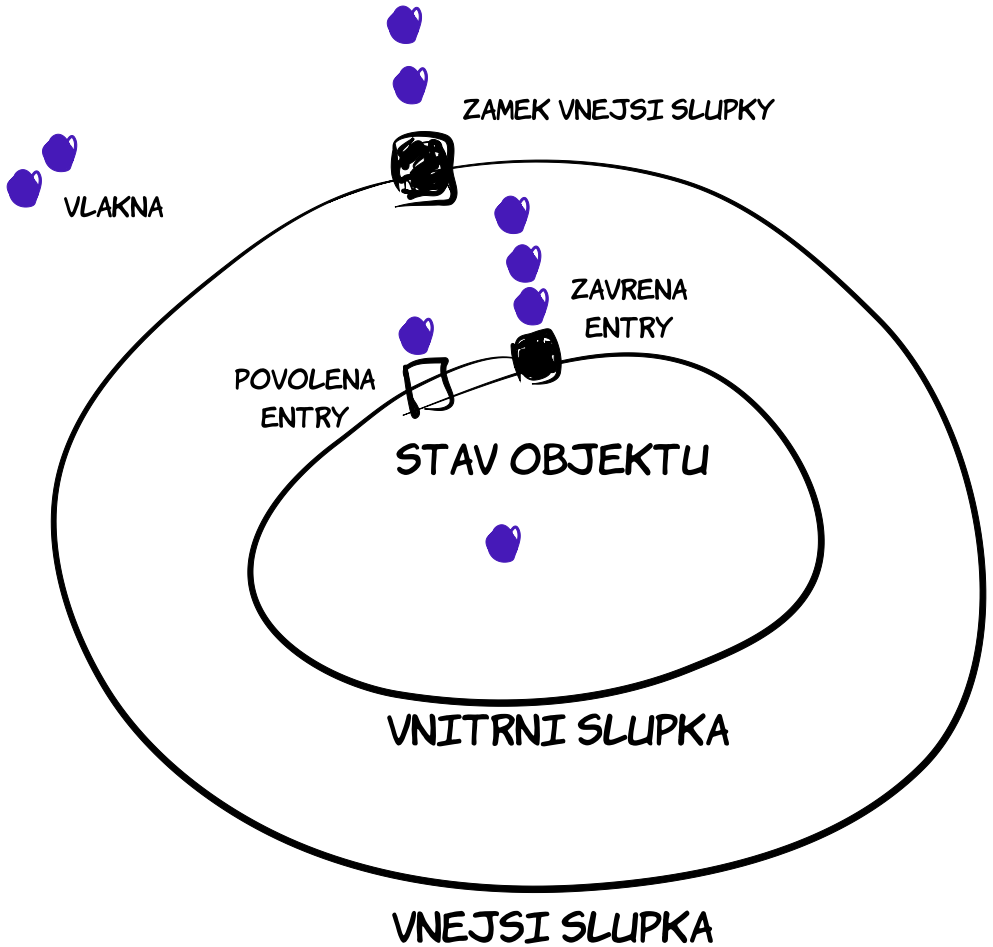
- Příklad řízení zdrojů

```
1  type Request_Range is range 1 .. Max;

3  protected Resource_Controller is
      entry Allocate(Request_Range)(R : out Resource);
5     procedure Release(R : Resource; Amount : Request_Range);
   private
7     Free : Request_Range := Request_Range'Last;
   end Resource_Controller;
9
   protected body Resource_Controller is
11    entry Allocate(for F in Request_Range)(R : out Resource)
         when F <= Free is
13    begin
         Free := Free - F;
15    end Allocate;
      procedure Release(R : Resource; Amount : Request_Range) is
17    begin
         Free := Free + Amount;
19    end Release;
   end Resource_Controller;
```

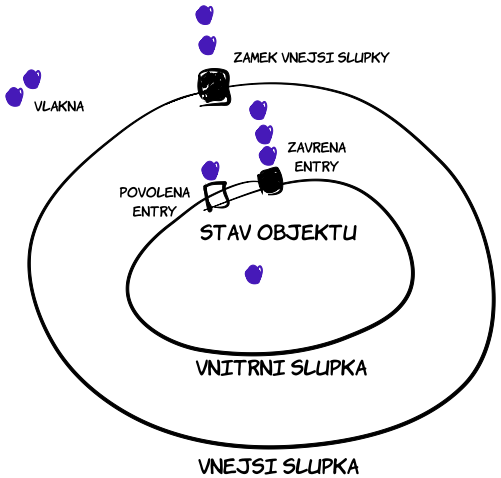
Rodiny entries

- Příklad řízení zdrojů
- Problémy
  - nevhodné pro větší množství alokovatelných zdrojů v jednom požadavku (m)
  - v případě soutěžení je výběr náhodný (prioritu lze nastavovat v rámci Real-Time Systems Annex)

„Eggshell" model volání chráněného objektu

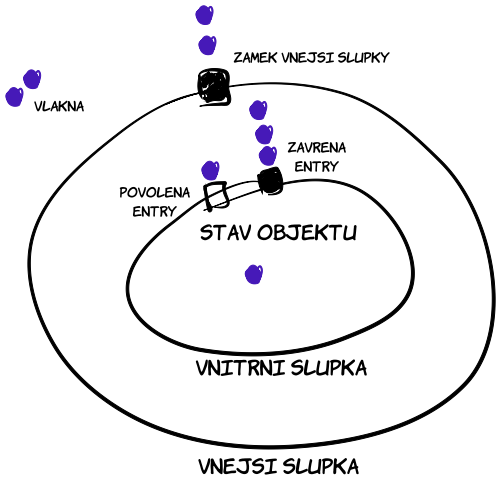


„Eggshell" model volání chráněného objektu



- Přednost entry calls čekajících ve frontě
- Zámek vnější slupky
  - nedovolí vstup do slupky, pokud je jiné vlákno aktivní voláním procedury nebo entry
  - po vstupu se vyhodnocuje asociovaná podmínka (stráž)
  - tento mechanismus zajišťuje bezpečnost použití `'Count`

„Eggshell" model volání chráněného objektu



- Vnitřní slupka
  - po výstupu z každé procedury a entry se vyhodnocují podmínky a eventuálně se propouští volání čekající na vnitřní slupce

requeue

- Jak poslat za dveře někoho, koho už jsme si pustili do místnosti?
- `requeue` umožňuje vysunout aktivní vlákno v chráněném objektu do fronty před vnitřní slupku
  - nové volání musí mít stejnou signaturu
  - implicitně není přerušitelné pomocí `abort`, aby objekt nezůstal v rozpracovaném stavu
  - nové volání může být přerušitelné `requeue with abort`
- `requeue` umí fungovat i napříč více chráněnými objekty/úlohami
  - neobvyklé, používat opatrně

requeue

```
type Request_Range is range 1 .. Max;

protected Resource_Controller is
   entry Allocate(R : out Resource; Amount : Request_Range);
   procedure Release(R : Resource; Amount : Request_Range);
private
   entry Assign(R : out Resource; Amount : Request_Range);
   Free : Request_Range := Request_Range'Last;
   New_Resources_Released : Boolean := False;
   To_Try : Natural := 0;
   ...
end Resource_Controller;

protected body Resource_Controller is
   entry Allocate(R : out Resource; Amount : Request_Range)
      when Free > 0 is
   begin
      if Amount <= Free then
         Free := Free - Amount;
         -- allocate
      else
         requeue Assign;
      end if;
   end Allocate;
```

requeue

```
   entry Assign(R : out Resource; Amount : Request_Range)
2     when New_Resources_Released is
   begin
4     To_Try := To_Try - 1;
      if To_Try = 0 then
6        New_Resources_Released := False;
      end if;
8     if Amount <= Free then
         Free := Free - Amount;
10       -- allocate
      else
12       requeue Assign;
      end if;
14 end Assign;
```

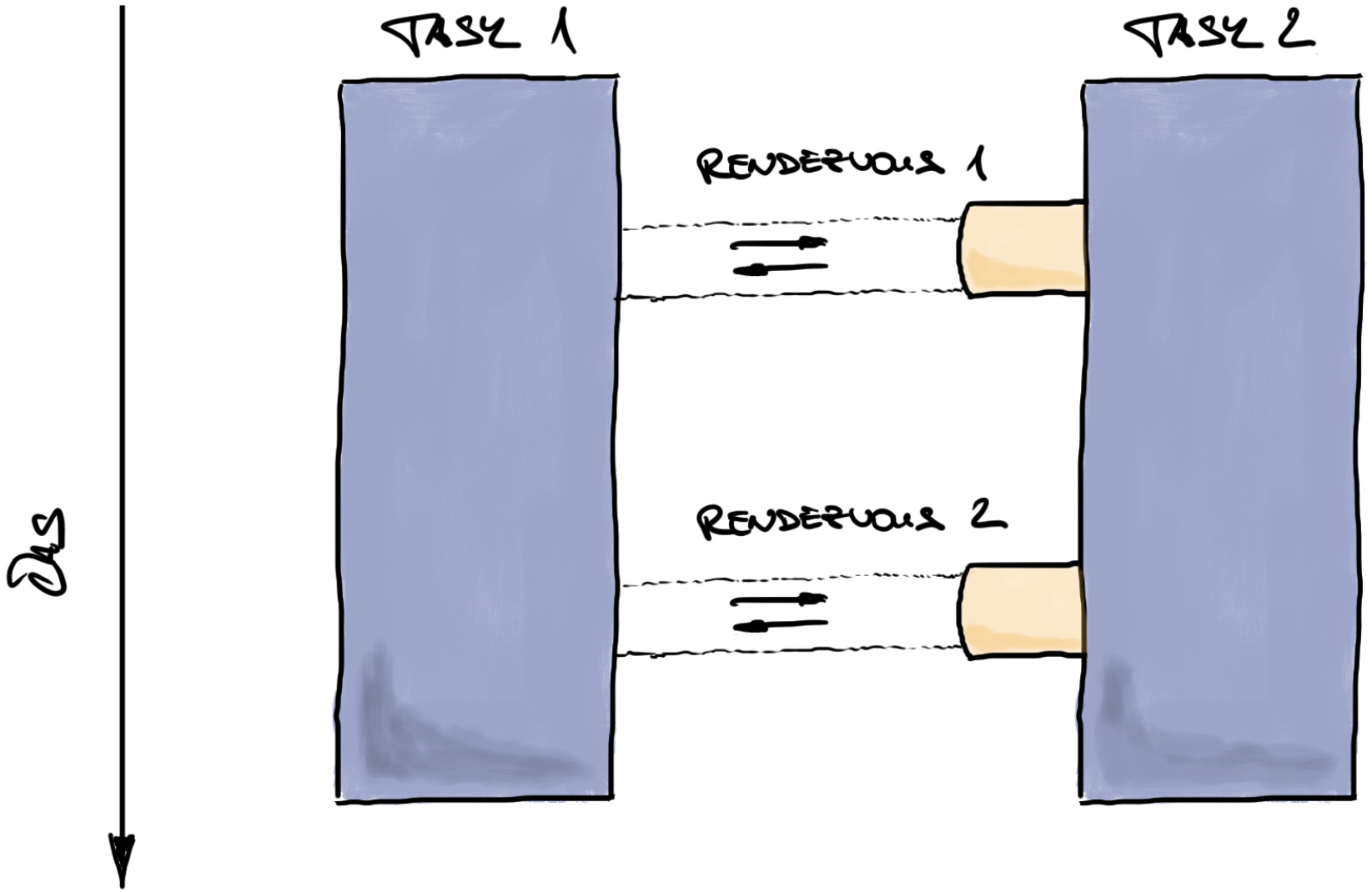
requeue

```
   procedure Release(R : Resource; Amount : Request_Range) is
2  begin
      Free := Free + Amount;
4     -- free resources
      if Assign'Count > 0 then
6        To_Try := Assign'Count;
         New_Resources_Released := True;
8     end if;
   end Release;
10 end Resource_Controller;
```

Ada: Tasks, Rendezvous

- Koncept CSP: Communicating Sequential Processes
  - Hoare, 1978
  - paralelně běžící sekvenční procesy
  - komunikace: zasílání zpráv
  - synchronizace: synchronní zasílání zpráv
    - odesílatel se zablokuje, dokud příjemce není schopen přijmout zprávu
    - příjemce se zablokuje, dokud není schopen od odesílatele přijmout zprávu

Tasks, Rendezvous



Tasks

- **`task`**
  - lokálně definované
    - běží od začátku rozsahu, v němž jsou definované
  - dynamicky alokované
    - **`access`** typ
    - alokace pomocí **`new`**
    - běží až od alokace
  - pole tasků
- ukončování
  - spontánní
  - **`abort`**

Tasks

```
task T is
end T;

task body T is
begin
   makam;
end T;

task type T_Type is
end T;

task body T_Type is
begin
   loop
      makam;
   end loop;
end T_Type;

Pole_T : array (1..10) of T_Type;

type T_Type_Access is access T_Type;
Dynamicky_T : T_Type_Access;
Dynamicky_T := new T_Type;
```

Parametrizace vláken

- předávání parametrů při vzniku vlákna
- užitečné s typy vláken

```
1 type Monitor_Procedure_Type is access procedure;

3 task type Monitor_Task_Type (Mon_Proc : Monitor_Procedure_Type) is
     entry Run;
5    entry Stop;
     entry Request_Terminate;
7 end Monitor_Task_Type;
```

Parametrizace vláken

```
task body Monitor_Task_Type is
   Finish_Flag : Boolean := False;
   Terminate_Flag : Boolean := False;
begin
   while not Terminate_Flag
   loop
      select
         accept Run;
         while not (Finish_Flag or Terminate_Flag)
         loop
            select
               accept Stop do
                  Finish_Flag := True;
               end Stop;
            else
               Mon_Proc.all;
            end select;
         end loop;
      or
         accept Request_Terminate do
            Terminate_Flag := True;
         end Request_Terminate;
      end select;
      Finish_Flag := False;
   end loop;
end Monitor_Task_Type;
```

Rendezvous

- místa synchronizace – předávání dat
- **`entry`**
  - deklarace rendezvous bodu
  - **`in`**, **`out`**, **`in out`** parametry
- **`accept`**
  - implementace v těle tasku

Tasks, Rendezvous

```
procedure Task1 is

   task Vlakno is
      entry ZadejX (X : in Integer);
      entry PrectiX (X : out Integer);
   end Vlakno;

   task body Vlakno is
      Hodnota : Integer;
   begin
      accept ZadejX (X : in Integer) do
         Hodnota := X;
      end ZadejX;
      Hodnota := Hodnota + 1;
      accept PrectiX (X : out Integer) do
         X := Hodnota;
      end PrectiX;
   end Vlakno;

   Chci_Inkrementovat : Integer;

begin
   Vlakno.ZadejX(Chci_Inkrementovat);
   Vlakno.PrectiX(Chci_Inkrementovat);
end Task1;
```

Rendezvous

- **`select`**
  - výběr z více **`accept`** možností

```
1  task body T is
   begin
3     loop
         select
5           accept Rande1 do
               neco;
7           end Rande1;
         or
9           accept Rande2 do
               neco;
11          end Rande2;
            neco;
13          accept Rande3 do
               neco;
15          end Rande3;
         or
17          terminate;
      end loop;
19 end T;
```

Rendezvous

- **`select`**
  - časovaný výběr

```
1  task body T is
   begin
3     loop
         select
5           accept Rande1 do
               neco;
7           end Rande1;
         or
9           delay 10.0;
            taky_neco;
11       end select;
      end loop;
13 end T;
```

Rendezvous

- **select**
  - časovaný výběr

```
1  task body T is
   begin
3     loop
         select
5           accept Rande1 do
               neco;
7           end Rande1;
         else   --  ekvivalent "or delay 0.0"
9           null;  --  busy waiting
         end select;
11    end loop;
   end T;
```

Rendezvous

- výjimky během rendezvous
  - jsou doručeny jak volajícímu vláknu, tak i vlastnímu vláknu
  - pokud výjimka není ošetřena ve vláknu, je vlákno ukončeno a výjimka se nepropaguje do rodiče (považováno za příliš disruptivní)

```
begin
   select
      accept Volani do
         ...
         raise Chyba;
         ...
      end Volani;
   end select;
exception
   when Chyba =>
      Naprav_Stav;
   when other =>
      Oznam_Uzivateli;
end;
```

Rendezvous

- výjimky během rendezvous
  - jsou doručeny jak volajícímu vláknu, tak i vlastnímu vláknu
  - pokud výjimka není ošetřena ve vláknu, je vlákno ukončeno a výjimka se nepropaguje do rodiče (považováno za příliš disruptivní)

```
--
--

begin
   T.Volani;
exception
   when Chyba =>
      Oznam_Uzivateli;
end;
```

Vnořené Rendezvous

- možnost vícecestné synchronizace

```
procedure Three_Way is
   task User;
   task Device;

   task Controller is
      entry Doio (I : out Integer);
      entry Start;
      entry Completed (K : Integer);
   end Controller;

   task body User is ...;
      -- includes calls to Controller.Doio(...)
   task body Device is
      J : Integer;
      procedure Read (I : out Integer) is ...;
   begin
      loop
         Controller.Start;
         Read(J);
         Controller.Completed(J);
      end loop;
   end Device;
```

Vnořené Rendezvous

- možnost vícecestné synchronizace

```
   task body Controller is
2  begin
      loop
4        accept Doio (I : out Integer) do
            accept Start;
6           accept Completed (K : Integer) do
            I := K;
8           end Completed;
         end Doio;
10    end loop;
   end Controller;
12 begin
   null;
14 end Three_Way;
```

Chráněné entries u vláken

- podobně jako u chráněných objektů s mírně odlišnou syntaxí

```
task body Ukazka is
   Zinicializovano : Boolean := False;
   Hodnota : Data;
begin
   loop
      select
         when Zinicializovano =>
         accept Cti (H : out Data) do
            H := Hodnota;
         end;
      or
         accept Zapis (H : in Data) do
            Hodnota := H;
         end;
         Zinicializovano := True;
      end select;
   end loop;
end Ukazka;
```

  - podmínka se vyhodnocuje při každém průchodu přes `select`
  - pokud není žádná podmínka splněna, je vyhozena výjimka `Program_Error` (možno použít strukturu `select ... else ... end select;`)
  - změna hodnot mezi testem a rendezvous (viz komentář u `'Count`)

Atributy vláken

- Ada 95 Quality and Style Guide, Section 6.2.3
- `'Terminated`
  - bezpečné pouze testování na `True` (po ukončení vlákno nemůže obživnout)
- `'Callable`
  - bezpečné pouze testování na `False` (po ukončení vlákno nemůže obživnout)

Atributy vláken

- Ada 95 Quality and Style Guide, Section 6.2.3
- `'Count`
  - chování vlákna by nemělo záviset na tomto atributu (používat raději jen s chráněnými objekty)

```
select
when Transmit'Count > 0 and Receive'Count = 0 =>
 accept Transmit;
 ...
or
accept Receive;
...
end select;
```

stav `'Count` se může změnit mezi vyhodnocením a následnou akcí (např. volající použil časově omezené volání a mezi testem a `accept` se ukončil)

  - u chráněných objektů: každá práce s frontou je chráněná (viz eggshell model)

Chráněné entries s timeoutem

- Nelze implementovat pomocí chráněných typů, jsou třeba vlákna

```
task body Ukazka is
   Zinicializovano : Boolean := False;
   Hodnota : Data;
begin
   loop
      select
         when Zinicializovano =>
         accept Cti (H : out Data) do
            H := Hodnota;
         end;
      or
         delay 1.0;
         --  neco
      end select;
   end loop;
end Ukazka;
```

Vynucené ukončování vláken

- Alternativní příklad

```
select
   T.Ukonci;
or
   delay 180*Seconds;
   abort T;
end select;
```

nebo pokud nevěříme, že se úloha po `T.Ukonci` ukončí

```
select
   T.Ukonci;
   delay 60*Seconds;
or
   delay 180*Seconds;
end select;
abort T;
```

  - pozor na nebezpečí použití `abort`

    `pragma Restrictions (No_Abort_Statements);`
  - ani druhé řešení není blbuvzdorné (pokud se `T.Ukonci` může zakousnout v rámci bloku `accept`, použití `requeue`)

Asynchronous Transfer of Control

- Potřeba *rychle* reagovat na asynchronní události
  - reakce na chyby (např. výpadek HW, kvůli němuž se akce nikdy nedokončí)
  - změny režimů v důsledku (neočekávaných) událostí
  - dosažení co nejlepšího výsledku v případě iterativního přerušitelného výpočtu
  - přerušení uživatelem

Asynchronous Transfer of Control

- Struktura

```
select
   --  triggering_statement
   delay 5.0;
   --  post_trigger_part
   Put_Line ("Tudy cesta nevede!");
then abort
   --  abortable_part
   Prevelevelmidlouhe_Volani;
end select;
```

  - pokud **abortable_part** doběhne dříve než **triggering_statement**, pokusí se ukončit **triggering_statement**
  - pokud **triggering_statement** doběhne dřív než **abortable_part**, je **abortable_part** ukončena a provede se část **post_trigger_part**
  - **triggering_statement** – v Ada 95 delay/entry, v Ada 2005 i procedury
  - **abortable_part** nemusí být implementována jako samostatný task

Asynchronous Transfer of Control

- Příklad: iterativní dlouhý výpočet, chceme nejlepší odhad v době, kdy jej potřebujeme (J. Barnes, Ada 95)
  - chráněný objekt na předávání posledního výsledku

```
1 begin Result is
     procedure Set_Estimate(X : in Data);
3    function Get_Estimate return Data;
  private
5    Est : Data;
  end;
```

  - signalizační objekt (např. uživatel chce výsledek)

```
  begin Trigger is
2    entry Wait;
     --  when Flag
4    procedure Signal;
  private
6    Flag : Boolean := False;
  end;
```

Asynchronous Transfer of Control

- Příklad: iterativní dlouhý výpočet, chceme nejlepší odhad v době, kdy jej potřebujeme (J. Barnes, Ada 95)
  - použití

```
1 select
     Trigger.Wait;
3 then abort
     Computation;
5 end select;
```

```
1 Trigger.Signal;
  E := Result.Get_Estimate;
```

  - oddělení logiky výpočtu od jeho ukončování – výpočet nemusí zjišťovat, kdy má končit

Asynchronous Transfer of Control

- Výjimky při ATC
  - pokud se odehraje jen jedna výjimka (v jedné z částí), je možno ji zachytit
  - pokud se odehrají dvě výjimky současně v řídící i přerušitelné části, je výjimka z přerušitelné části ztracena

Plánování a spouštění vláken

- Task dispatching
  - proces výběru vlákna, které má běžet na daném procesoru
- Dispatching points
  - místa v kódu, kde dochází k přepínání
  - vždy:
    1. blokování na volání (rendezvous, chráněný objekt)
    2. ukončování úlohy
  - další místa jsou definována annexem pro specifické politiky
- Politika plánování se nastavuje per partition
  - partition – dělení aplikace podle Distributed Annex
- Nedeterminismus v Adě je zapříčiněn
  - plánováním a prokládáním běhu vláken
  - výběrem varianty ve výrazech `select`
  - chování chráněných objektů

Dynamické priority

- Base priority – základní priorita přiřazená vláknu
  - lze měnit pomocí **`Ada.Dynamic_Priorities`**

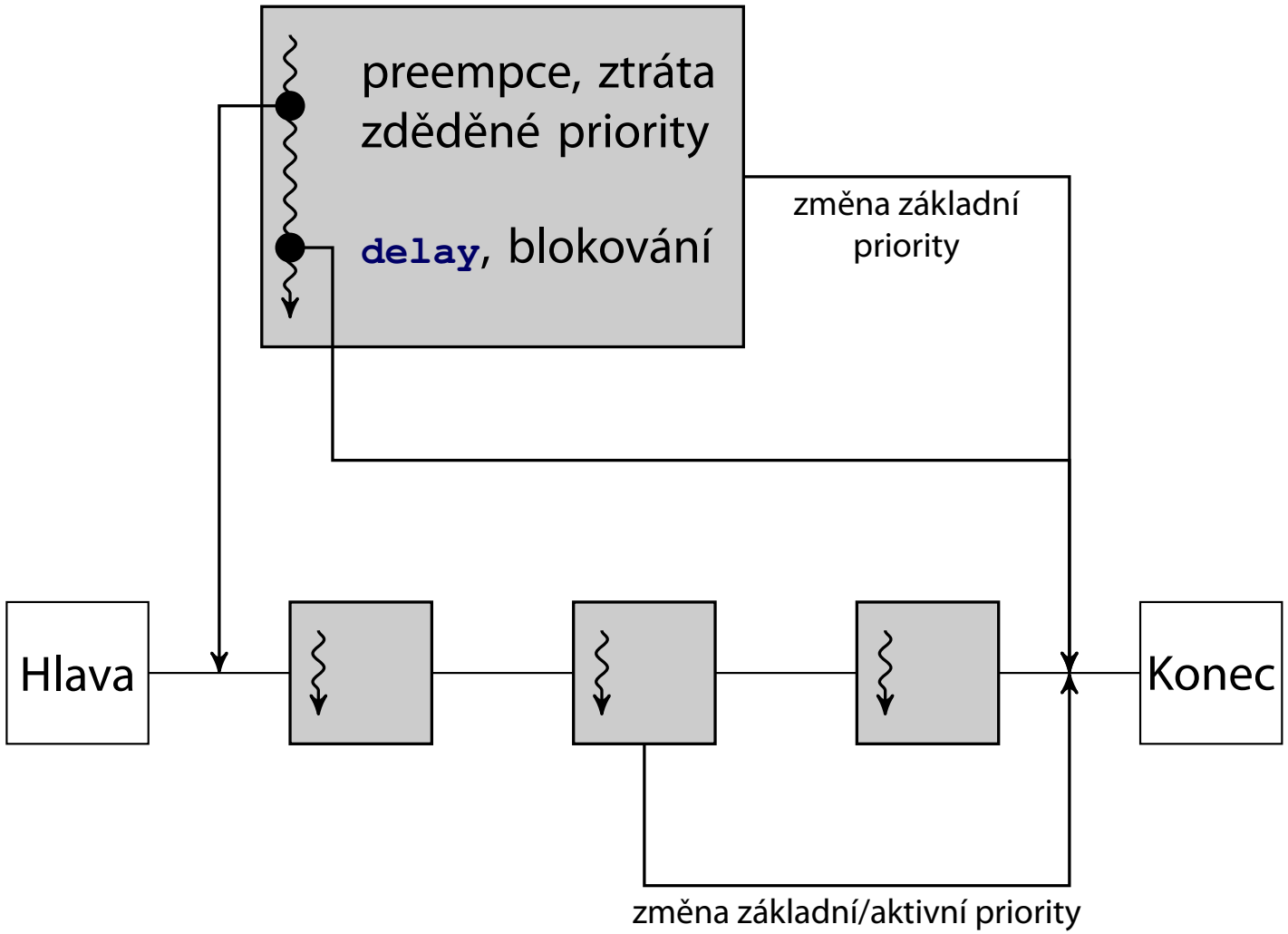
```
package Ada.Dynamic_Priorities is

   procedure Set_Priority
     (Priority : System.Any_Priority;
      T        : 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;
```

- Active priority – skutečná priorita v daném okamžiku (ovlivněná děděním)
- Dynamic ceiling priority – lze měnit pomocí atributu **`'Priority`**
  - potenciál vyhazování **`Program_Error`** výjimek (bounded error) – pořadí: snížení ceiling priority musí být až po snížení priority vlákna

Preemptive Fixed Priority Dispatching

- `pragma Task_Dispatching_Policy(FIFO_Within_Priorities);`
- Priorita
  - definuje fronty, z nichž se odebírají úlohy v případě výběru
  - vybírá se ze začátku nejprioritnější neprázdné fronty
  - systém musí podporovat:
    - minimálně 30 úrovní `System.Priority`
    - minimálně 1 úroveň `System.Interrupt Priority`
- Dispatching points specificky pro Preemptive Fixed Priority Dispatching
  - kdykoli se objeví spustitelné (runnable) vlákno s vyšší prioritou $\Longrightarrow$ preemtpivita
  - kdykoli se v kódu objeví `delay`, který už vypršel
    - `delay 0.0;`
- Podpora i před Ada 2005
  - velmi dobře prostudované chování: ~ 30 let výzkumu a používání

Preemptive Fixed Priority Dispatching



Round-Robin Dispatching

- `pragma Task_Dispatching_Policy(Round_Robin_Within_Priorities);`
- Běh vlákna je omezen *kvantem*
- Oproti Preemptive Fixed Priority Dispatching přidává task dispatching do kódu kdykoli dojde k využití kvanta vláknem (execution time budget = quantum)
  - přerušené vlákno je zařazeno na konec fronty své priority
- Mapuje poměrně dobře na SCHED_RR politiku POSIXu
  - musí se ošetřit, aby kvantum neexpirovalo během aktivace a rendezvous

Round-Robin Dispatching

```
package Ada.Dispatching.Round_Robin is

   pragma Unimplemented_Unit;

   Default_Quantum : constant Ada.Real_Time.Time_Span :=
                       Ada.Real_Time.Milliseconds (10);

   procedure Set_Quantum
     (Pri     : System.Priority;
      Quantum : Ada.Real_Time.Time_Span);

   procedure Set_Quantum
     (Low, High : System.Priority;
      Quantum   : Ada.Real_Time.Time_Span);

   function Actual_Quantum
     (Pri : System.Priority) return Ada.Real_Time.Time_Span;

   function Is_Round_Robin (Pri : System.Priority) return Boolean;

end Ada.Dispatching.Round_Robin;
```

Non-Preemptive Fixed Priority Dispatching

- `pragma Task_Dispatching_Policy (Non_Preemptive_FIFO_Within_Priorities);`
- Vlákno nemůže být přerušeno vláknem vyšší priority kdykoli.
  - může však být přerušeno vláknem ošetřujícím interrupt (vč. časovače), ale pak je řízení vráceno původnímu vláknu, i když je k dispozici vlákno s vyšší prioritou

Non-Preemptive Fixed Priority Dispatching

- Lze udělat kooperativní preempci `delay 0.0`
  - jazyk definuje tzv. bounded error pokud se z akce chráněného objektu volá potenciálně blokující operace
    - `select`
    - `accept`
    - entry call
    - `delay`
    - vytváření nebo aktivace vlákna (task)
    - volání podprogramu, jehož tělo obsahuje potenciálně blokující operaci
  - bounded error – specifikace jazyka vyjmenovává seznam následků chyby, mimálně obsahuje vyhození výjímky `Program_Error`
  - výjimku tvoří chráněné objekty implementující interrupt handlery – lze snadno identifikovat, protože používají `pragma Interrupt Handler` a/nebo `pragma Attach Handler`

$\Longrightarrow$ vlákno se nemůže uspat/přerušit uvnitř chráněného objektu

$\Longrightarrow$ není třeba dělat ceiling priority

$\Longrightarrow$ jednodušší implementace

Earliest Deadline First Dispatching

- `pragma Task_Dispatching_Policy(EDF_Across_Priorities);`
  musí být použito dohromady s
  `pragma Locking_Policy(Ceiling_Locking)`
- Je-li systém naplánovatelný, pak jej lze naplánovat pomocí EDF.
- Každá úloha ma přiřazený termín dokončení – deadline
  - ve frontě jsou úlohy seřazeny podle termínu dokončení
  - z fronty se odebírá úloha s nejbližším termínem
- Kombinace s prioritami
  - více front, odebírá se z nejprioritnější neprázdné
  - aktivní priorita vlákna/úlohy není už přímo odvozena od základní priority

Earliest Deadline First Dispatching

- Komplikovanější koncept aktivní priority
  - pokud nějaké vlákno B pracuje v chráněném objektu (tedy s ceiling prioriton P) a vlákno A má dřívější termín, je vlákno A zařazeno do fronty s prioritou větší než P (existuje-li taková fronta)
  - pokud nikdo nepracuje s chráněnými objekty, je vlákno A zařazeno do fronty s prioritou `Priority'First`
  - vlákno A podědí aktivní prioritu fronty
- Dispatching points pro vlákno A při použití EDF
  - změna termínu A
  - zkrácení termínu pro úlohu B ve frontě s prioritou A, pokud nový termín B je nastane dříve jako termín A
  - pokud se objeví úloha ve frontě s prioritou vyšší než A
- Problém s implementovatelností na běžných OS (poznámka ve specifikaci balíku v GNATu)
  - implementováno např. pro MARTE OS (**`http://marte.unican.es/`**)

Earliest Deadline First Dispatching

```
package Ada.Dispatching.EDF is

   subtype Deadline is Ada.Real_Time.Time;

   Default_Deadline : constant Deadline := Ada.Real_Time.Time_Last;

   procedure Set_Deadline
      (D : Deadline;
       T : Ada.Task_Identification.Task_Id :=
             Ada.Task_Identification.Current_Task);

   procedure Delay_Until_And_Set_Deadline
      (Delay_Until_Time : Ada.Real_Time.Time;
       Deadline_Offset  : Ada.Real_Time.Time_Span);

   function Get_Deadline
      (T : Ada.Task_Identification.Task_Id :=
             Ada.Task_Identification.Current_Task)
       return Deadline;

end Ada.Dispatching.EDF;
```

Srovnání metod

- FIFO
  - dobře předpověditelné chování
- EDF
  - nejlepší využití zdrojů
  - pokud jsou dané termíny splnitelné, EDF je dokáže naplánovat
- Round-robin
  - férové rozdělené zdrojů

Řazení do front chráněných objektů

- Problém blokování prioritních vláken méně duležitými při čekání na chráněném objektu.
- Problém, pokud je současně otevřených (povolených) více variant v rámci `select ... accept ...`
  - jazyk nespecifikuje pořadí
- `pragma Queuing_Policy(Priority_Queuing);`
- Uspořádání z pohledu volání jednoho entry: priority + FIFO
  - vybírá se z neprázdné fronty s nejvyšší prioritou
  - mezi vlákny stejné priority se vybírá FIFO podle pořadí volání
- Uspořádání z pohledu soutěže mezi různými entries a/nebo otevřenými cestami `select`: textové pořadí + family entry index
  - pokud je otevřeno více volání a volající mají stejné priority, volí se podle uspořádání (textu) v definici (týká se i pokud ve stejnou dobu vyexpirují `select ... delay ...`) – kvůli jednoduchosti a zajištění determinismu
  - v případě soutěže mezi voláními stejné family entry má nižší index prioritu

Zpoždění vzbuzení

- Vzbuzení po použití `delay` a `delay until` je nejdříve se specifikovaném okamžiku, ale může nastat i později
  - zpoždění se označuje jako *lateness*
- Použití `delay` a `delay until` má nějakou režii i v případě, že se fakticky nečeká (tj. parametry vyustí v `delay 0.0`)
  - režie se promítá do kódu v případě specifikace dispatching points pomocí `delay 0.0`

Zpoždění vzbuzení

- Příklad pro GNATforLEON **`http://polaris.dit.upm.es/~ork/download/opm-2.1.0.pdf`**
- *The implementation shall document the following metrics* (only those metrics that are significant in the context of the Ravenscar profile are cited)*:*
  - *An upper bound on the execution time, in processor clock cycles, of a* **`delay until`** *statement whose requested value of the delay expression is less than or equal to the value of* **`Real Time.Clock`** *at the time of executing the statement.*
    The measured value is equal to 740 processor clock cycles.

Zpoždění vzbuzení

- Příklad pro GNATforLEON **`http://polaris.dit.upm.es/~ork/download/opm-2.1.0.pdf`**
- *The implementation shall document the following metrics* (only those metrics that are significant in the context of the Ravenscar profile are cited)*:*
  - *An upper bound on the lateness of a* **`delay until`** *statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a* **`delay until`** *statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement.*

    The following measurements have been performed:

    - *One task + background task* The delay until lateness upper bound for a call to a delay until statement is 8051 clock cycles (161 μs), using a 50 MHz system clock. This lateness occurs when the time of the delay until coincides with a second boundary. It must be noted that the clock interrupt occurs every second in the kernel tested.
      If the time of the delay until statement does not coincide with a clock interrupt, the lateness upper bound for the execution of a delay until statement is 7061 clock cycles (141.2 μs).

Zpoždění vzbuzení

- Příklad pro GNATforLEON **`http://polaris.dit.upm.es/~ork/download/opm-2.1.0.pdf`**
- *The implementation shall document the following metrics* (only those metrics that are significant in the context of the Ravenscar profile are cited)*:*
  - *An upper bound on the lateness of a* **`delay until`** *statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a* **`delay until`** *statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement.*

    The following measurements have been performed:

    - *N tasks + background task*
      The lateness of delay until for N tasks is $294825.84+7\times N$ μs when the time of the delay until coincides with a clock interrupt. Otherwise, it is $201.8 + 7\ N\times$ μs.

Hodiny reálného času

- **`Ada.Real_Time`** – monotónní hodiny s vysokým rozlišením
  - **`Time`** – vyjádření časového okamžiku
  - **`Time_Span`** – vyjádření rozsahu trvání/intevalu
  - srovnání (minimálních) požadavků na hodiny v Adě

	`Calendar`	**`Real_Time`**
rozsah času	500 let	50 let
rozsah intervalu	1 den	$\pm$ 1 hodina
přesnost	20 ms	20 μs

Rozsah **`Real_Time`** může být menší na platformách se slovem kratším jak 32 b.

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

begin
   T_One : Time := Clock;
   TS : Time_Span :=  To_Time_Span(1.0);
   T_Two : Time := T_One + TS;
   delay until T_Two;
   delay To_Duration (TS);
end;
```

Časovače událostí

- **`Ada.Real_Time.Timing_Events`**
- Využití pro událostmi řízené programování bez vláken a komplikace kódu pomocí **`delay`**

```
package Ada.Real_Time.Timing_Events is

   type Timing_Event is tagged limited private;

   type Timing_Event_Handler
     is access protected procedure (Event : in out Timing_Event);

   procedure Set_Handler
     (Event   : in out Timing_Event;
      At_Time : Time;
      Handler : Timing_Event_Handler);

   procedure Set_Handler
     (Event   : in out Timing_Event;
      In_Time : Time_Span;
      Handler : Timing_Event_Handler);

   function Current_Handler
     (Event : Timing_Event) return Timing_Event_Handler;

   procedure Cancel_Handler
     (Event     : in out Timing_Event;
      Cancelled : out Boolean);

   function Time_Of_Event (Event : Timing_Event) return Time;
```

Časovače událostí

- **`Timing_Event`**
  - tagovaný typ – možnost rozšíření o vlastní data
  - privátní ne-abstraktní typ – nepotřebuje run-time dispatching
- **`Timing_Event_Handler`**
  - obdoba obsluhy přerušení
  - ukazatel na **`access protected procedure`**
- **`Set_Handler`**
  - varianta s absolutním časem
  - varianta s relativním časem
  - **`null`** místo ukazatele na proceduru vymaže časovač (ekvivalent **`Cancel_Handler`**)
  - opakované volání přepisuje budoucí událost
- Spuštění události
  - co nejdříve poté, co uplyne specifikovaný čas
  - z důvodů efektivity se obvykle pověsí na přerušení hodin ≤ 10 ms (obdobně jako ošetřování **`delay`** ve vláknech)
- Nevýhoda: kód běží s prioritou **`Interrupt_Priority`**
  - pro složitější a déle běžící úlohy je lépe používat vlákna
  - míněno jako lightweight mechanismus pro omezené platformy

Časovače událostí

```
protected Watchdog is
   pragma Interrupt_Priority (Interrupt_Priority'Last);

   entry Alarm_Control;
   procedure Call_In;

private
   procedure Timer(Event : in out Timing_Event);
   Alarm : Boolean := False;
end Watchdog;

Fifty_Mil_Event : aliased Timing_Event;
TS : Time_Span := Milliseconds(50);
Set_Handler(Fifty_Mil_Event, TS, Timer'Access);
```

- Aplikace musí volat `Call_In` nejpozději 1× za 50 ms
- `Alarm_Control` umožňuje reagovat na vzniknuvší alarm

Zdroj: Burns & Wellings: Concurrent and Real-Time Programming in Ada

Časovače událostí

```
protected body Watchdog is
   entry Alarm_Control when Alarm is
   begin
      Alarm := False;
   end Alarm_Control;

   procedure Timer(Event : in out Timing_Event) is
   begin
      Alarm := True;
      -- Note no use is made of the parameter in this example
   end Timer;

   procedure Call_in is
   begin
      Set_Handler(Fifty_Mil_Event, TS, Timer'Access);
      -- This call to Set_Handler cancels the previous call
   end Call_in;
end Watchdog;
```

Zdroj: Burns & Wellings: Concurrent and Real-Time Programming in Ada

Odlehčená komunikace mezi vlákny

- Definice efektivnějších komunikačních nástrojů než jsou rendezvous
  - nízkoúrovňovější nástroje umožňuje efektivnější implementaci
  - problém Ady 83: vysokoúrovňová abstrakce (rendezvous) se musela používat i pro implementaci nízkoúroňových primitiv (typu semaforů) $\Longrightarrow$ inverze abstrakce
  - řešení v Adě 95:
    - chráněné objekty
    - synchronní řízení vláken
    - asynchronní řízení vláken

Odlehčená komunikace mezi vlákny

- Synchronní komunikace mezi vlákny

```
package Ada.Synchronous_Task_Control is

   type Suspension_Object is limited private;
   procedure Set_True (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);
   function Current_State (S : Suspension_Object) return Boolean;
   procedure Suspend_Until_True (S : in out Suspension_Object);
```

  - ekvivalent `wait/notify`
  - `Set_True`, `Set_False`, `Current_Stat` jsou vzájemně atomické a neblokující
  - `Suspend_Until_True` překlopí suspension object zpět na `False`

Odlehčená komunikace mezi vlákny

- Asynchronní komunikace mezi vlákny

```
package Ada.Asynchronous_Task_Control is

   pragma Unimplemented_Unit;
   procedure Hold (T : Ada.Task_Identification.Task_Id);
   procedure Continue (T : Ada.Task_Identification.Task_Id);
   function Is_Held (T : Ada.Task_Identification.Task_Id) return Boolean;

end Ada.Asynchronous_Task_Control;
```

  - umožňuje zasuspendovat jiné vlákno – potenciálně nebezpečné
  - koncept idle task priority
  - suspendování se provádí pomocí snížení priority pod idle task priority
    - volání `Hold` na vlákno řízené EDF jej dočasně vyloučí z EDF
    - dispatching points odpovídají plánovači, kterým jsou vlákna v daném okamžiku řízena
    - řeší problém, aby se vlákno nezasuspednovalo uvnitř chráněného objektu (nejsou v něm dispatching points)
    - pokud je zavolán `accept` zasuspendovaného vlákna, je vykonán, protože podědí prioritu volajícího
    - pokud je vlákno blokováno uvnitř chráněného objektu v čekání na otevření stráže entry, je uvolněno, pokud je se stráž otevře a vlákno je jediné ve frontě

Možnosti omezení

- **`pragma Restrictions`** – kontrolované před během programu

  No_Task_Hierarchy All (non-environment) tasks only depend directly on the environment task.

  No_Nested_Finalization Objects with controlled parts, and access types that designate such objects, are declared only at library level.

  No_Abort_Statement There are no abort statements.

  No_Terminate_Alternatives There are no select statements with terminate alternatives.

  No_Task_Allocators There are no allocators for task types or types containing task subcomponents.

  No_Implicit_Heap_Allocation There are no operations that implicitly require heap storage allocation to be performed by the implementation. For example, the concatenation of two strings usually requires space to be allocated on the heap to contain the result.

Možnosti omezení

- **`pragma Restrictions`** – kontrolované před během programu

  No_Dynamic_Priorities There is no use of dynamic priorities.

  No_Dynamic_Attachments There are no calls to any of the operations defined in package Interrupts, e.g. Attach Handler.

  No_Local_Protected_Objects Protected objects are only declared at the library level.

  No_Local_Timing_Events Timing events are only declared at the library level.

  No_Protected_Type_Allocators There are no allocators for protected types or types containing protected subcomponents.

  No_Relative_Delay There are no relative delay statements (i.e. delay).

  No_Requeue_Statements There are no requeue statements.

  No_Select_Statements There are no select statements.

  No_Specific_Termination_Handlers There are no calls to the specific handler routines in the task termination package.

  Simple_Barriers The boolean expression in an entry barrier is either a static boolean expression or a boolean component of the enclosing protected object (e.g. a simple boolean variable).

Možnosti omezení

- **`pragma Restrictions`** – nedefinované místo kontroly

  Max_Select_Alternatives Specifies the maximum number of alternatives in a select statement.

  Max_Task_Entries Specifies the maximum number of entries per task. The maximum number of entries for each task type (including those with entry families) must be determinable at compile-time. A value of zero indicates that no rendezvous is possible.

  Max_Protected_Entries Specifies the maximum number of entries per protected type. The maximum number of entries for each protected type (including those with entry families) must be determinable at compile-time.

Možnosti omezení

- **`pragma Restrictions`** – kontrola za běhu

  No_Task_Termination All tasks are non-terminating. It is implementation defined what happens if a task terminates – but any fall-back handler must be executed as the first task terminates.

  Max_Storage_At_Blocking Specifies the maximum portion (in storage elements) of a task's storage size that can be retained by a blocked task. If a check fails, Storage Error is raised at the point where the respective construct is elaborated.

  Max_Asynchronous_Select_Nesting Specifies the maximum dynamic nesting level of asynchronous select statements. A value of zero prevents the use of any such statement. If a check fails, Storage Error is raised as above.

  Max_Tasks Specifies the maximum number of tasks, excluding the environment task, that are allowed to exist over the lifetime of a partition. A zero value prevents tasks from being created. If a check fails, Storage Error is raised as above.

  Max_Entry_Queue_Length This defines the maximum number of calls queued on an entry. Violation will cause Program Error to be raised at the point of call.

Ravenscar

- **`pragma Profile(Ravenscar);`**

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions(
   No_Abort_Statements,
   No_Dynamic_Attachment,
   No_Dynamic_Priorities,
   No_Implicit_Heap_Allocations,
   No_Local_Protected_Objects,
   No_Local_Timing_Events,
   No_Protected_Type_Allocators,
   No_Relative_Delay,
   No_Requeue_Statements,
   No_Select_Statements,
   No_Specific_Termination_Handlers,
   No_Task_Allocators,
   No_Task_Hierarchy,
   No_Task_Termination,
   Simple_Barriers,
   Max_Entry_Queue_Length => 1,
   Max_Protected_Entries => 1,
   Max_Task_Entries => 0,
   No_Dependence => Ada.Asynchronous_Task_Control,
   No_Dependence => Ada.Calendar,
   No_Dependence => Ada.Execution_Time.Group_Budget,
   No_Dependence => Ada.Execution_Time.Timers,
   No_Dependence => Ada.Task_Attributes);
```

Ravenscar

- Nesmí být hierarchie vláken
  - vlákna se musí být deklarována na úrovni knihoven, nikoli z hlavního vlákna
- Zákaz rendezvous
  - vlákna se musí synchronizovat přes chráněné objekty
- Předávání dat
  - atomické proměnné
  - chráněné objekty
  - využití suspension objects

```
Ada.Synchronous_Task_Control.Suspend_Until_True(S);
Ada.Synchronous_Task_Control.Set_True(Periodic.S);
```

- Omezení na nejvýše jedno entry per chráněný objekt/typ
  - kombinace protected type a suspension object
- Pouze jedno vlákno smí čekat ve vnitřní slupce entry
  - separátní entries pro různá vlákna

Ravenscar – příklady

- Periodická úloha

```
task type Periodicka (Prio : System.Priority; Cyklus : Positive) is
   pragma Priority (Prio);
end Periodicka;

task body Periodicka is
   Dalsi_Cas : Ada.Real_Time.Time;
   Interval : constant Ada.Real_Time.Time_Span :=
      Ada.Real_Time.Microseconds(Cyklus);
begin
   Dalsi_Cas := Ada.Real_Time.Clock + Interval;
   loop
      --  neco
      delay until Dalsi_Cas;
      Dalsi_Cas := Dalsi_Cas + Interval;
   end loop;
end Periodicka;
```

Ravenscar – příklady

- Transformace více entries – 2 vlákna sbírají informace ze dvou sond, jedno vlákno je čte

```
protected Probe_Protector is
   entry Write (D : in Data_Type; Probe_ID : in Natural)
      when not Data_Ready;
   entry Read (DA : out Data_Type_Array)
      when Data_Ready;
end Probe_Protector;
```

  - více entries per chráněný objekt
  - ne-jednoduchá podmínka ve stráži
    - lze spravit snadno druhým příznakem s opačným významem
  - potenciálně 2 vlákna čekající ve `Write` entry

Zdroj: M. Ben-Ari, Ada for Software Engieneers, 2nd ed. for Ada 2005

Ravenscar – příklady

- Ravenscar verze
  - použijeme kombinaci chráněného objektu a suspension objektu pro každou sondu

```
protected Probe_Protector_Ravenscar is
   type Data_Type_Array is array(0..1) of Data_Type;
   type SO_Type is
   array(Data_Type_Array'Range) of Ada.Synchronous_Task_Control.Suspension_Object;

   procedure Write (D : in Data_Type; Probe_ID : in Natural);
   entry Read (DA : out Data_Type_Array)
      when Data_Ready;
end Probe_Protector_Ravenscar;

protected body Probe_Protector_Ravenscar is
   ...
   entry Read (Data : out Data_Type_Array)
      when Data_Ready is
   begin
      DA := ...
      for I in SO_Type'Range loop
         Ada.Synchronous_Task_Control.Set_True(S(I));
      end loop;
   end Read;
   ...
end Probe_Protector_Ravenscar;
```

Zdroj: M. Ben-Ari, Ada for Software Engieneers, 2nd ed. for Ada 2005

Ravenscar – příklady

- Ravenscar verze
  - zapisující vlákna se zastavují na suspension objektu

```
task body Probe_Collector (ID: Probe_ID) is
begin
   Ada.Synchronous_Task_Control.Set_True(S(ID));
   loop
      Ada.Synchronous_Task_Control.Suspend_Until_True(S(ID));
      delay until Next;
      ...
      Probe_Protector_Ravenscar.Write(D, ID);
      Next := Next + Interval;
   end loop;
end Probe_Collector;
```

Zdroj: M. Ben-Ari, Ada for Software Engieneers, 2nd ed. for Ada 2005

Přehled přednášky

Erlang: distribuované programování

- Erlang
  - funkcionální programovací jazyk pro paralelní a distribuované programování
  - An Erlang Course
    `http://www.erlang.org/course/course.html`
  - **`http://www.erlang.org/doc/reference_manual/processes.html`**
  - skutečně použitelný: např. ejabberd, zpracování transakcí u Goldman-Sachs
- Paralelismus na konceptu CSP
  - komunikující procesy
  - asynchronní zasílání zpráv
  - každý proces má svůj „mailbox“

Erlang: distribuované programování

- vytvoření procesu

```
spawn(Modul, Exportovana_fce, Seznam_argumentu)
```

- předávání zpráv

```
PID_procesu ! zprava
receive zprava -> udelej_neco
```

- registrace procesů

```
register(nejaky_atom, PID)
whereis(registrovane_jmeno)
```

Erlang: distribuované programování

```
-module (priklad).
-compile(export_all).

klient(Pid) ->
   Pid ! {self(),pozadavek,ping},
   receive
      {Pid,odpoved,Odpoved} ->
         io:format("dorazila odpoved ~p~n",[Odpoved]),
         Pid ! exit
   end,
   io:format("klient skoncil~n").

server() ->
   receive
      {Od,pozadavek,Pozadavek} ->
         io:format("obdrzel jsem pozadavek ~p od ~p~n",
            [Pozadavek, Od]),
         sleep(1000),
         Od ! {self(),odpoved,pong},
         server();
      _ ->
         io:format("server skoncil~n")
   end.
```

Erlang: distribuované programování

```
sleep(Time) ->
   receive
   after Time -> void
  end.

spust() ->
   Pid = spawn(fun server/0),
   spawn(fun() -> klient(Pid) end),
   io:format("spusteni server i klient~n", []).
```

```
-bash-2.05b$ erl
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0] [hipe]

Eshell V5.5.2  (abort with ^G)
1> c(priklad).
{ok,priklad}
2> priklad:spust().
spusteni server i klient
obdrzel jsem pozadavek ping od <0.37.0>
ok
dorazila odpoved pong
klient skoncil
server skoncil
```

Erlang: distribuované programování

- Svázené (linked) procesy

```
link(Pid) -> true
spawn_link(Modul, Exportovana_fce, Seznam_argumentu)
unlink(Id) -> true
```

  - pokud umře proces, pošle o tom zprávu všem, kteří jej mají „nalinkovaný“

```
{'EXIT', FromPid, Reason}
```

- Monitorování procesů

```
erlang:monitor(process, registrovane_jmeno)
```

  - alternativa k použití linků
  - při ukončení procesu zašle zprávu

```
{'DOWN', Ref, process, Pid2, Reason}
```

  - vícenásobné volání vytvoří více monitorů

- Zachytávání Exit signálů

```
process_flag(trap_exit,true)
```

  - při exitu vygeneruje zprávu jako u linků