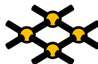
Vláknové programování
část XIII

Lukáš Hejmánek, Petr Holub

{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2011–05–19

Přehled přednášky

Systémy real-time
- Plánování a spouštění vláken
- Řazení do front
- Časovače a události
- Komunikace mezi vlákny

Omezující profily

Závěrečný projekt

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**

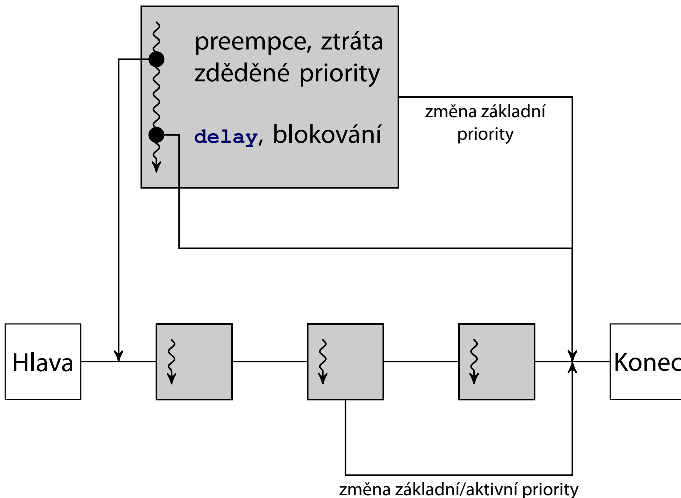
```
package Ada.Dynamic_Priorities is

   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 $\leq 10\,\mathrm{ms}$ (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
  - využítí 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

Závěrečný projekt

- Smyslem je navrhnout a implementovat paralelismus do existujícího rozsáhlejšího technického kódu.
- Odevzdávání:
  - minimálně 3 dny před zkouškou
  - odevzdat implementaci
  - odevzdat 1-2 strany dlouhou zprávu o řešení projektu

Závěrečný projekt: Java

Vláknová paralelizace řešiče systémů s omezujícími podmínkami: Choco

- `http://www.emn.fr/z-info/choco-solver/`
- existuje distribuovaná verze DisCHOCO
  `http://www.lirmm.fr/coconut/dischoco/`
- Redouane Ezzahir Christian Bessiere Mustapha Belaissaoui and El Houssine Bouyakhf: DisChoco: A platform for distributed constraint programming, DCR'07 Proceedings, 16–27.
  `http://liawww.epfl.ch/Publications/Archive/DCR07Proceedings.pdf`

Závěrečný projekt: C/C++

Vláknová paralelizace nástroje pro interpolaci snímků pomocí detekce pohybu: yuvmotionfps

- `http://jcornet.free.fr/linux/yuvmotionfps.html`