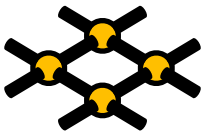
Vláknové programování
část X

Lukáš Hejmánek, Petr Holub

{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192

2011–04–28

Přehled přednášky

Paměťový model Javy

Ada

Erlang

(příklady povětšinou převzaty z JCiP, Goetz)

```
long promenna = 10000000L;
```

Paměťový model Javy

- *happens-before*
  - částečné uspořádání

The rules for *happens-before* are:

Program order rule. Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Monitor lock rule. An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock.[3]

Volatile variable rule. A write to a volatile field *happens-before* every subsequent read of that same field.[4]

Thread start rule. A call to `Thread.start` on a thread *happens-before* every action in the started thread.

Tabulka převzata z JCiP, Goetz

Paměťový model Javy

Thread termination rule. Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`.

Interruption rule. A thread calling `interrupt` on another thread *happens-before* the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).

Finalizer rule. The end of a constructor for an object *happens-before* the start of the finalizer for that object.

Transitivity. If *A happens-before B*, and *B happens-before C*, then *A happens-before C*.

Tabulka převzata z JCiP, Goetz

Paměťový model Javy

- Piggybacking
  - spojení happens-before pravidla s jiným pravidlem, obvykle monitorem nebo `volatile`
  - raději nepoužívat
  - příklad: `http://kickjava.com/src/java/util/concurrent/FutureTask.java.htm`
    - postaveno na `tryReleaseShared` happens-before `tryAcquireShared`
    - kombinace volatilní proměnné `runner`, do které `tryReleaseShared` zapisuje s program order

Paměťový model Javy

```
        void innerSet(V v) {
2           for (;;) {
                int s = getState();
4               if (ranOrCancelled(s))
                    return;
6               if (compareAndSetState(s, RAN))
                    break;
8           }
            result = v;
10          releaseShared(0);
            done();
12      }
```

```
        V innerGet() throws InterruptedException JavaDoc, ExecutionException Jav
2           acquireSharedInterruptibly(0);
            if (getState() == CANCELLED)
4               throw new CancellationException JavaDoc();
            if (exception != null)
6               throw new ExecutionException JavaDoc(exception);
            return result;
8       }
```

Paměťový model Javy

```
1         protected int tryAcquireShared(int ignore) {
              return innerIsDone()? 1 : -1;
3         }

5         /**
           * Implements AQS base release to always signal after setting
7          * final done status by nulling runner thread.
           */
9         protected boolean tryReleaseShared(int ignore) {
              runner = null;
11            return true;
          }
13
          boolean innerIsCancelled() {
15            return getState() == CANCELLED;
          }
17
          boolean innerIsDone() {
19            return ranOrCancelled(getState()) && runner == null;
          }
```

Paměťový model Javy

- líná inicializace

```
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }

    static class Resource {
    }
}
```

Paměťový model Javy

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }

    static class Resource {
    }
}
```

- líná inicializace thread-safe

Paměťový model Javy

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() {
        return resource;
    }

    static class Resource {
    }
}
```

- „dychtivá“ inicializace
- využívá skutečnosti, že statické inicializátory jsou vždy dokončeny před použitím třídy

Paměťový model Javy

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceFactory.ResourceHolder.resource;
    }

    static class Resource {
    }
}
```

- idiom líné inicializace s použitím holder class
- využívá líné inicializace tříd

Paměťový model Javy

```
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
```

- Double Checked Locking anti-pattern
- pomíjí možnost, že `resource` je v nedefinovaném stavu
- od Java 5.0 možno spravit použitím `volatile`
- nepoužívat
  - ani v C/C++!

Přehled přednášky

(některé příklady z Burns & Wellings: Concurrent and Real-Time Programming in Ada – CRTPA)

Atomické a volatilní proměnné

- `pragma Atomic ();`
  - zajišťuje atomické aktualizace proměnných

```
Prirozene_Cislo : Natural;
pragma Atomic (Prirozene_Cislo);
```

- `pragma Atomic_Components ();`
  - zajišťuje atomické aktualizace součástí složeného typu `record`

```
type Byte is range 0 .. 255;
for Byte'Size use 8;
type Moje_Struktura is
record
        B1 : Byte;
        B2 : Byte;
        B3 : Byte;
        B4 : Byte;
end record;
pragma Atomic_Components (Moje_Struktura);
```

Atomické a volatilní proměnné

- `pragma Volatile ();`
  - upozornění pro kompilátor, že se hodnoty proměnných mohou neočekávaně měnit
  - zejména kompilátor musí zamezit optimalizacím, které by mohly interferovat (zakazuje cachování na čtení i zápis)

```
Buffer_Zarizeni : Integer;
pragma Volatile (Buffer_Zarizeni);
```

- `pragma Volatile_Components ();`
  - totéž pro komponenty složeného typu `record`

Simpsonův algoritmus

- Kompilátor může mít omezení na maximální délku atomické proměnné
  - pokud je (nesplnitelný) požadavek na větší atomickou proměnnou, musí ho kompilátor odmítnout
- Algoritmus pro větší proměnné: Simpson'90[1]
  - dva sloty, každý o dvou bankách
  - do jednoho slotu se zapisuje (round-robin do bank)
  - ze druhého slotu se čte (poslední zapsaná hodnota)
  - atomické nastavování indexů slotů a bank
  - volatilní zápisy do bank/slotů

```
generic
   type Data is private;
   Initial_Value : Data;
package Simpsons_Algorithm is
   procedure Write(Item : Data); -- non-blocking
   procedure Read (Item : out Data); -- non-blocking
end Simpsons_Algorithm;
```

[1] H. Simpson, 'Four-Slot Fully Asynchronous Communication Mechanism', IEE Proceedings, 137 (Pt.E.1), 17–30 (January 1990). Implementace z CRTPA.

Simpsonův algoritmus

```
1  package body Simpsons_Algorithm is
      type Slot is (First, Second);
3     Four_Slot : array (Slot, Slot) of Data :=
         (First => (Initial_Value,Initial_Value),
5        Second => (Initial_Value,Initial_Value));
      pragma Volatile(Four_Slot);
7
      Next_Slot : array(Slot) of Slot := (First, First);
9     pragma Volatile(Next_Slot);

11    Latest : Slot := First;
      pragma Atomic(Latest);
13
      Reading : Slot := First;
15    pragma Atomic(Reading);
```

Simpsonův algoritmus

```
   procedure Write(Item : Data) is
2     Pair, Index : Slot;
   begin
4     if Reading = First then
      Pair := Second;
6     else
      Pair := First;
8     end if;
      if Latest = First then
10    Index := Second;
      else
12    Index := First;
      end if;
14    Four_Slot(Pair, Index) := Item;
      Next_Slot(Pair) := Index;
16    Latest := Pair;
   end Write;
```

Simpsonův algoritmus

```
   procedure Read(Item : out Data) is
2     Pair, Index : Slot;
   begin
4     Pair := Latest;
      Reading := Pair;
6     Index := Next_Slot(Pair);
      Item := Four_Slot(Pair, Index);
8  end Read;

10 end Simpsons_Algorithm;
```

Protected Types – Monitory

- Implementace monitorů
  - funkce – nemohou měnit data
    procedury – mohou měnit data
    entry – strážený vstup, mohou měnit data
  - efektivní paralelizace: podobne `ReadWriteLock`u v Javě
    - funkce mohou přistupovat paralelně
    - procedury a entries musí pracovat exkluzivně

```
1 protected type Muj_Typ is
     procedure Nastav_hodnotu (n : Integer);
3    procedure Odnastav_hodnotu;
     function Zjisti_hodnotu return Integer;
5    entry Pockej_na_nastaveni (n : Integer);
  private
7    Hodnota : Integer;
     Nastaveno : Boolean := False;
9 end Muj_Typ;
```

Protected Types – monitory

```
10 protected body Muj_Typ is
      procedure Nastav_hodnotu (n : Integer) is
12    begin
         Hodnota := n;
14       Nastaveno := True;
      end Nastav_hodnotu;
16
      procedure Odnastav_hodnotu is
18    begin
         Nastaveno := False;
20    end Odnastav_hodnotu;

22    function Zjisti_hodnotu return Integer is
      begin
24       return Hodnota;
      end Zjisti_hodnotu;
26
      entry Pockej_na_nastaveni
28    when Nastaveno is
      begin
30       null;
      end Pockej_na_nastaveni;
32 end Muj_Typ;
```

Guarded Entries

- chránění dle privátního stavu

```
protected type Chraneno_Stavem is
   entry Vstup;
private
   I : Integer;
end Chraneno_Stavem;

protected body Chraneno_Stavem is
   entry Vstup when I > 0 is
   begin
      null;
   end Vstup;
end Chraneno_Stavem;
```

  - používat pouze privátní proměnné
  - např. implementace mutexů a semaforů

Guarded Entries

- Chránění dle atributů

```
1  protected type Chraneno_Stavem is
       entry Vstup;
3  private
       I : Integer;
5  end Chraneno_Stavem;

7  protected body Chraneno_Stavem is
       entry Vstup when Vstup'Count > 4 is
9      begin
           null;
11     end Vstup;
   end Chraneno_Stavem;
```

  - možnost použití atributů
  - atribut `E'Count` vrátí počet zablokovaných vláken na vstupu do `entry E`
  - např. implementace bariér
  - funguje bezpečně pouze u chráněných objektů, nikoli tasků

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

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

Rodiny entries

- Příklad řízení zdrojů

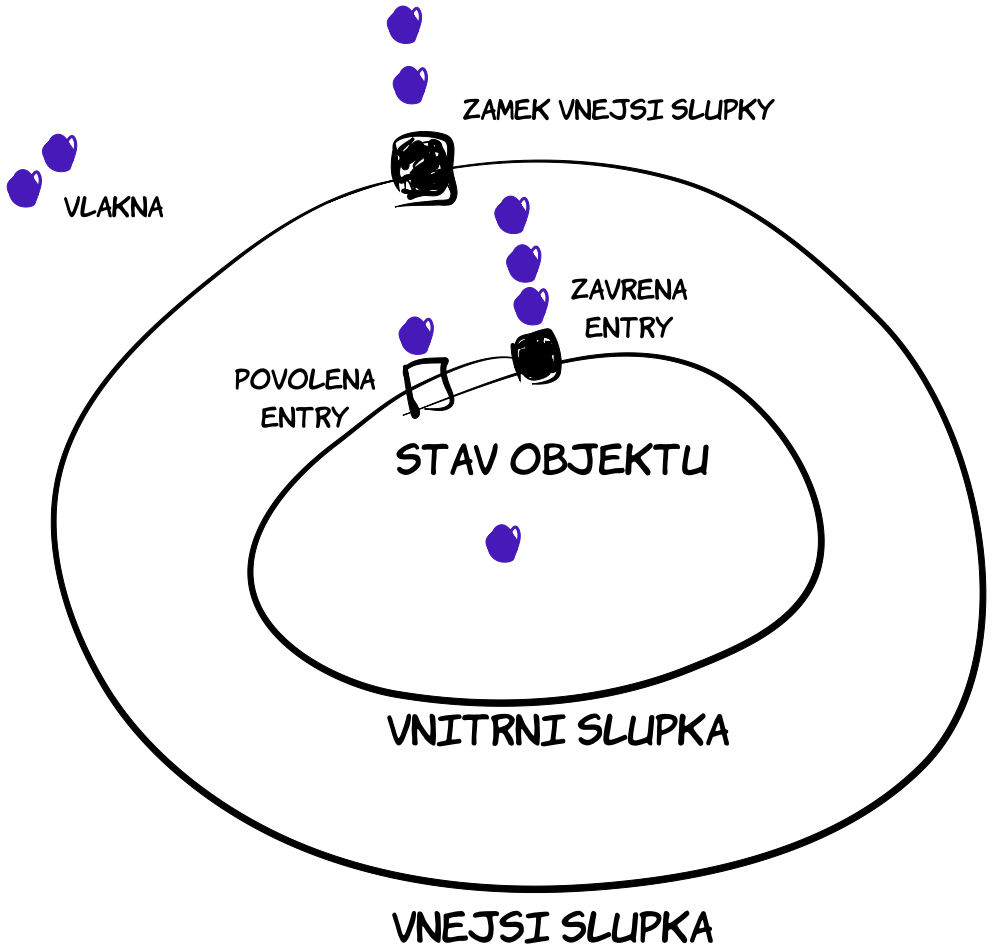
```
1  type Request_Range is range 1 .. Max;

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;
```

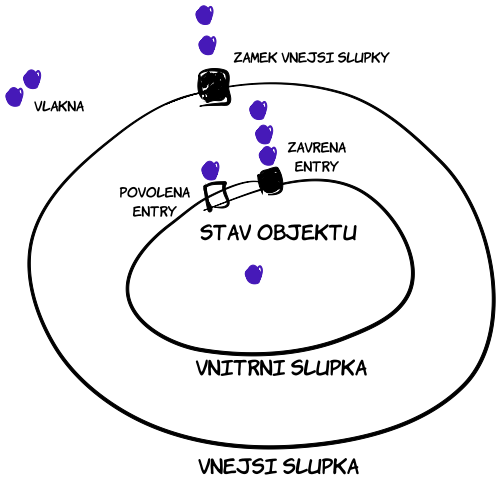
Rodiny entries

- 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

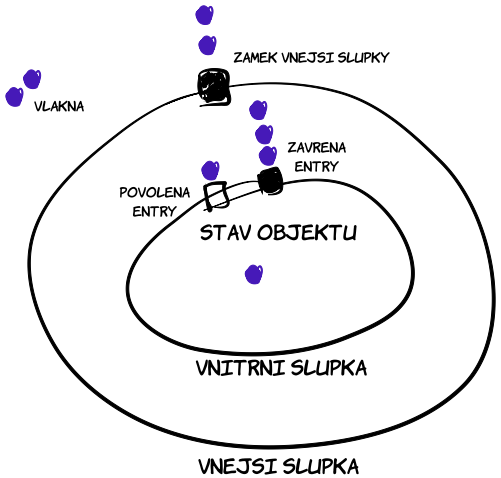


„Eggshell" model volání chráněného objektu



- 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;
```

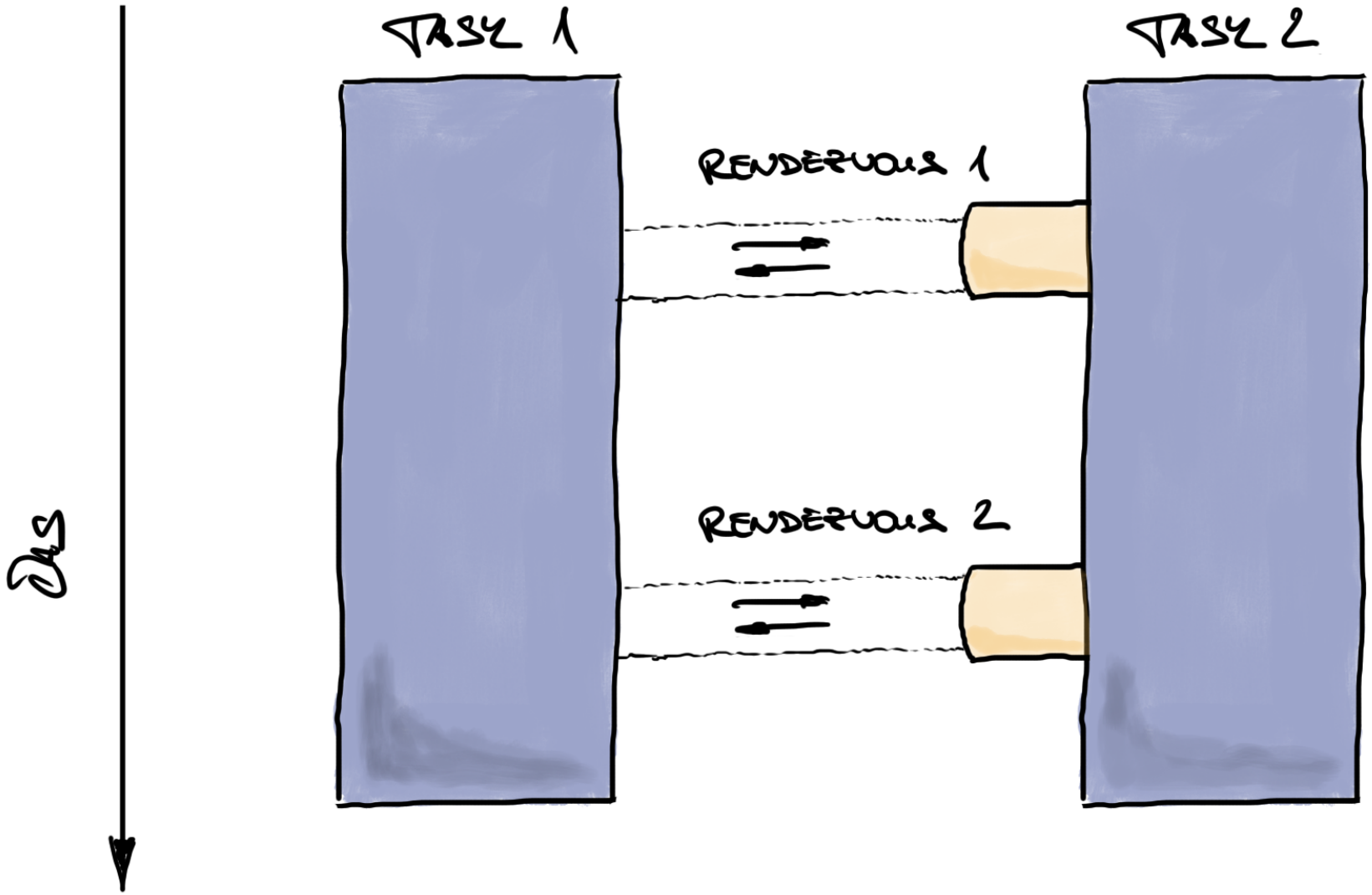
requeue

```
   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
   begin
      loop
         accept Doio (I : out Integer) do
            accept Start;
            accept Completed (K : Integer) do
            I := K;
            end Completed;
         end Doio;
      end loop;
   end Controller;
begin
   null;
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

Programování v reálném čase

- Real-Time and Distributed Systems Annex
  - (dynamické) priority vláken při volání entries
  - monotónní hodiny s vysokou přesností
  - restrikce tasků pro speciální případy (Ravenscar profile)
  - preemptivní `abort`
  - mnoho dalšího

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