Rendez-Vous, vč. select v Adě, a jeho porovnání s monitorem.

Thursday, May 30, 2013 8:33 AM

- Ada je objektově orientovaný jazyk se silnou verifikací typů (nelze implicitně přetypovat datové typy – např. void pointer)
- nepoužívá interpretr
- nepodporuje fibres
- paralelní části výpočtu se označují jako tasky (ne thready)
- mohou být prováděny na jednom procesoru, více procesorech nebo více počítačích, není to však vyjádřeno v kódu programu
- pro synchronizaci tasků se používá asymetrické synchronní rendezvous (zasílání zpráv)

Tasky

Konstrukce *task* představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy.

Deklarace je následující:

```
task jméno is
deklarace jmen komunikačních typů
end jméno;
task body jméno is
lokální deklarace a příkazy
end jméno;
```

- pro ukončení procesu lze použít příkaz *abort jméno;*, ale jeho použití by mělo být výjimečné
- k ukončení tasku se používá příkaz *terminate*

```
Select
    Accept e
Or
    Terminate
End select
```

Rendez-vous

Pro interakci procesů používá ADA principu asymetrického rendezvous, kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje tak elegantní konstrukci monitorů.

- prostředek pro synchronizaci úkolů (tasks)
- dva úkoly spolu komunikují pomocí rendez-vous - Meeting point, entry calls
- task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat

```
task Simple_Task is
entry Start(Num : in Integer);
entry Report(Num : out Integer);
end Simple_Task;
task body Simple_Task is
Local_Num : Integer;
begin
//čeká na vložení čísla – entry call
accept Start(Num : in Integer) do
Local_Num := Num;
end Start;
//normálně pokračuje v běhu
Local_Num := Local_Num * 2;
//čeká na vyzvednutí spočítané hodnoty
```

```
accept Report(Num : out Integer) do
Num := Local_Num;
end Report;
end Simple_Task;
```

- uvedený příklad stačí, pouze pokud potřebujeme jen jedno vlákno běžící podle uvedeného kódu
- průběh dostaveníčka - accept:
  - klient zavolá server
  - server si převezme parametry
  - server provede výpočet, klient spí
  - server předá výsledky

Select - může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) – pokaždé na jiné dle okolností – tj. ne v předem určeném pořadí

```
//Vynutíme si inicializaci a další se
//už pak může vykonávat v libovolném
//pořadí.
accept Init(Item : in Integer) do
    Local_Item := Item;
end Init;
loop
    select
        accept Stop;
                exit;
        or
        when podmínka = > //může i nemusí být
                accept Put(Item : in Integer) do
                    Local_Item := Item;
                end Put;
                Local_Item := Local_Item * 2;
        else
                Put_Line("No entry call at this time");
    end select;
    delay 0.01;
end loop;
```

Protected Objects, Protected Types

- tasky mohou sdílet objekty
- objekt je instance typu – klíčové slovo *type*
- klíčové slovo *protected* zajistí exkluzivní přístup k chráněnému objektu
- jsou tři operace nad chráněnými objekty:
  - **Procedury** – mění stav objektu, aniž by pro to musela být splněna podmínka; překladač se stará, aby měly exkluzivní přístup k objektu
  - **Entry calls** – stejné jako procedury, ale pro vykonání *entry call* je třeba navíc splnit podmínku
  - **Funkce** – pouze vrací stav a nic nemění, a proto nemusí mít exkluzivní přístup k objektu

Porovnání s monitorem

Rendezvous vs. monitors

The previous example is highly reminiscent of the monitor solution. Both provide “structured” approaches to mutual exclusion. Mutual exclusion is implicit in both monitors and rendezvous.

Passive and active objects

Monitors are **passive** objects.
• Only client threads exist.
• The client thread performs the service for itself inside the monitor.
• Server state does not change spontaneously.
Modules containing server threads are **active** objects.
• The server thread acts on behalf of the client thread within the module for the duration of the

rendezvous.
• Server threads may change the module state in between calls from clients.

http://www.engr.mun.ca/~theo/Courses/cp/pub/cp8-rendezvous.pdf - viz příloha

Renezvous v jave ze stranek zcu:
http://www.kiv.zcu.cz/research/groups/dss/download/presentation-2005-03-21.ppt

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> no

Printout

30. května 2013 16:28

Rendezvous

Rendezvous provides synchronization and two-way communication between two threads, a client and a server.

- From the client's point of view the rendezvous is a procedure call (remote or local)
- The client blocks until the server executes an in statement (MPDP) or accept statement (Ada).
- Server blocks until a call is made that it is prepared to accept.
- Once both are ready
  - arguments are copied to parameters
  - code is executed by the server
  - results are returned and "copy out" parameters are copied to arguments.
  - Both threads proceed independently.

Page 2

30. května 2013 16:28

Example

```
module TicketServer
        op getNext returns int ;
body
        process daemon {
                int next := 0 ;
                while( true) {
                        in getNext() returns val ->
                                val := next ;
                        ni
                        next := next + 1 ;} }
        end TicketServer
```

Note that mutual exclusion is implicit, since the server thread can be handling but 1 request at a time.

Choice

The server thread can offer a choice of requests that it will accept and can specify the conditions under which it will accept a choice.

Page 3

30. května 2013 16:28

```
module Bounded_buffer
        op deposit(char data);
        op fetch(result char data);
body

        process Buffer {
                char buf[n]; # buffer
                int front = 0; # first full slot
                int count = 0; # number of full slots

                while (true) {
                        in deposit(data) and count < n ->
                                buf[(front+count)%n] = data ;
                                count := count+1 ;
                        [] fetch(data) and count > 0 ->
                                data := buf[front] ;
                                front := (front+1)% n ;
                                count := count - 1 ;
                        ni } }
end Bounded_buffer
```

Page 4

30. května 2013 16:28

Rendezvous vs. monitors

The previous example is highly reminiscent of the monitor solution.

Both provide “structured” approaches to mutual exclusion.

Mutual exclusion is implicit in both monitors and rendezvous.

Passive and active objects

Monitors are passive objects.

- Only client threads exist.
- The client thread performs the service for itself inside the monitor.
- Server state does not change spontaneously.

Modules containing server threads are active objects.

- The server thread acts on behalf of the client thread within the module for the duration of the rendezvous.
- Server threads may change the module state in between calls from clients.

Page 5

30. května 2013 16:28

Data state vs. control state

With active objects, control state as well as data state can regulate operations that can proceed.

```
module Buffer
        op deposit( char data) ;
        op fetch( result char data) ;
body
        char buffer ;

        process daemon {
             while (true) {
                  in deposit(data) -> buffer := data ; ni
                  in fetch(data) -> data := buffer ; ni } }
end Buffer
```

- Acceptance of deposit and fetch strictly alternate.
- 2 states are represented by the program counter.
- Use of control state is sometimes clearer than use of data state.

Page 6

30. května 2013 16:28

Wait/signal vs. nested "in"

In monitors: a wait can happen at any point service is suspended until it can resume later.

With rendezvous: once a service has started it can only be suspended by a nested "in". Consider:

```
module Barrier {
        op done ;
body

        process daemon {
            while (true) {
                in done() -> in done() -> skip ni
                ni ; } }
end Sync2
```

How would you write an N process barrier?

Page 7

30. května 2013 16:28

Rendezvous vs. (Remote) Procedure Call

Rendezvous provides synchronization and mutual exclusion.

- RPC is handled by a new thread. Mutual exclusion must be made explicit.
- Rendezvous is handled by a single thread. Hence implicit mutual exclusion.

Rendezvous vs. Synchronous Message Passing

As with synchronous message passing the client (sender) is delayed until the server (receiver) is ready to accept the communication.

In a degenerate form

```
in MessType( param ) -> local := param ni
```

rendezvous is a synchronous receive. We abbreviate the above by

```
receive MessType( local ) ;
```

Rendezvous adds the ability for the server to

Page 8

30. května 2013 16:28

- delay the client until further processing is done and
- to send information back to client after that processing.

Rendezvous in Ada

The rendezvous is strongly associated with Ada since

- Ada introduced the concept (early ’80s)
- No other major language has supported it.

In Ada

- in is called accept.
- Operations are called entries.
- Choice requires use of select statement.

Conditional acceptance can not depend on parameter values.

Page 9

30. května 2013 16:28

An Ada Example

```
loop
        select when count < n =>
                accept deposit( data : in char ) do
                        buf((front+count) mod n) := data ;
                end deposit ;
                count := count+1 ;
        or when count > 0 =>
                accept fetch( data : out char ) do
                        data := buf(front) ;
                end fetch ;
                front := (front+1) mod n ;
                count := count - 1 ;
        end select ;
end loop ;
```

Page 10

30. května 2013 16:28

Non-blocking servers and clients (Ada)

The server thread can opt not to block.

```
loop
        select
                accept calibration( v : real ) do
                        scale := v ;
                end calibration
        else
                null ; - - do nothing
        end select
        sensorOut := scale * sensorIn ;
end loop
```

Likewise, the client can make a conditional call depending on whether the server is currently prepared to accept it.

```
select
        Queue.deposit( packet ) ;
else droppedPacketCount := droppedPacketCount + 1 ;
end select
```

Page 11

30. května 2013 16:28

Time outs

The server thread may time-out if it blocks too long.

A watch-dog thread

```
loop
        select
                accept AllIsWell ;
        or delay 10.0 ;
                RaiseAlarm ;
        end select
end loop
```

Likewise, the client may time out if service is not sufficiently fast

```
select Queue.deposit( packet ) ;
or delay 20.0 ;
        droppedPacketCount := droppedPacketCount + 1 ;
end select
```