

Vláknové programování

část XII

Lukáš Hejtmánek, Petr Holub

`{xhejtman, hopet}@ics.muni.cz`



Laboratoř pokročilých síťových technologií

PV192
2011-05-12

Přehled přednášky

Vlákna a GUI

Futures & TPE v C++

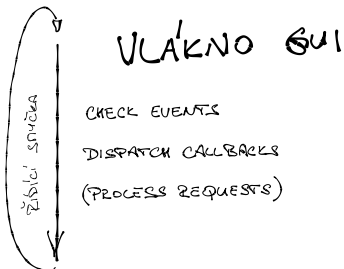
Vlákna a JNI

Systémy real-time

Ostatní

Vlákna a GUI

- GUI jsou obecně řízená událostmi
 - asynchronní vznik události
 - obvykle se jedno vlákno stará o obsluhu událostí – “event loop”



- Problémy událostmi řízeného modelu GUI
 - problém předávání změn z jiných vláken
 - problém s dlouho běžícími úlohami obsluhujícími událost

Vlákna a GUI

- Možnosti synchronizace mezi vlákny a GUI

1. thread-safe/multi-thread GUI

- ◆ nepoužívá se
- ◆ Graham Hamilton: Multithreaded toolkits: A failed dream?

http://weblogs.java.net/blog/kggh/archive/2004/10/multithreaded_t.html

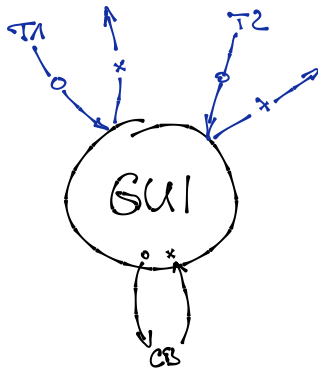
2. explicitní zamykání

- ◆ např. Gtk s použitím Gdk zámků

3. předávání práce GUI vláknu

- ◆ Java SWING, QT, GtkAda Contributions

Explicitní zamykání s GUI



○ LOCK

x UNLOCK

Explicitní zamykání s GUI

- Strategie

1. před startem GUI inicializuji zamykání
2. před startem smyčky obsluhy událostí získám zámek
3. smyčka obsluhy událostí zámek periodicky použít
4. jiné vlákno, pokud chce kreslit, musí získat zámek a po dokončení jej pustit

- Problémy

- ověření, že při všech aktualizacích GUI získávám zámek
- ověření, že po všech aktualizacích GUI použítím zámek
- ověření, že u callbacků *nezískávám a nepouštím* zámky

GtkAda

- Ada binding pro Gtk+, Glib a Gdk
 - prakticky kompletní wrapper z pohledu Gtk+
 - může spolupracovat s tasky
 - 2.14 stabilní, 2.18 v CVS
- Dostupné pro Windows, Linux, MacOS X
- <http://libre.adacore.com/libre/tools/gtkada/>

GtkAda

- Inicializace
- Vytvoření smyčky událostí
 - držení zámku v době startu

```
1  Gtk.Main.Set_Locale;  
   Gdk.Threads.G_Init;  
3  Gdk.Threads.Init;  
   Gtk.Main.Init;  
5  Init_GUI;  
   Gdk.Threads.Enter;  
7  Gtk.Main.Main;  
   Gdk.Threads.Leave;  
9  return;
```


GtkAda

- Modifikace z jiného vlákna
- Zámek pro modifikace

```

2  task body Counter is
   begin
       Main_Loop :
4      loop
           Counter_Monitor.Wait_For_Start;
6          -- because the main thread is waiting for us on termination
           -- we can touch Gtk objects until we signal termination; beware of ogra
8          -- more appropriate is exiting early
           exit Main_Loop when Counter_Monitor.Check_If_Quit;
10         Gdk.Threads.Enter;
           -- Ada95 syntax
12         Gtk.Button.Set_Label (Global_Window.all.Run_Button, -" Stop ");
           Gdk.Threads.Leave;
14         Counter_Loop :
           for I in 0 .. 1000 loop
16             exit Main_Loop when Counter_Monitor.Check_If_Quit;
               if Counter_Monitor.Check_If_Stop then
18                 exit Counter_Loop;
               end if;
20             Gdk.Threads.Enter;
               Gtk.Label.Set_Label (Global_Window.all.Counter_Label, -(Integer' Imag
22             Gdk.Threads.Leave;
               delay 1.0;
24         end loop Counter_Loop;

```

GtkAda

- Modifikace z jiného vlákna
- Zámek pro modifikace

```
2      Gdk.Threads.Enter;
      -- Ada2005 extended syntax
      Global_Window.all.Run_Button.all.Set_Label (-" Run ");
4      Counter_Monitor.Finished;
      Global_Window.all.Run_Button.all.Set_Sensitive (True);
6      Gdk.Threads.Leave;
      end loop Main_Loop;
8      Counter_Monitor.Finished;
      end Counter;
```

GtkAda

- Callbacky

```
2  procedure Quit is
   begin
     Counter_Monitor.Quit;
4   -- BEWARE OF OGRES! May deadlock if lock is not given up!
     Gdk.Threads.Leave;
6   Counter_Monitor.Wait_Until_Finished;
     Gdk.Threads.Enter;
8   Destroy (Global_Window);
     Gtk.Main.Main_Quit;
10  end Quit;

12  procedure On_Quit_Button_Clicked
     (Button : access Gtk.Button.Gtk_Button_Record'Class)
14  is
   begin
16     pragma Unreferenced (Button);
     Quit;
18  end On_Quit_Button_Clicked;
```

GtkAda

- Callbacky

```
1  procedure On_Run_Button_Clicked
2      (Button : access Gtk.Button.Gtk_Button_Record'Class)
3  is
4      begin
5          if Counter_Monitor.Check_If_Running then
6              Button.all.Set_Sensitive (False);
7              Counter_Monitor.Stop;
8          else
9              Counter_Monitor.Start;
10             end if;
11     end On_Run_Button_Clicked;
```

GtkAda

- Callbacky

```
1   Button_Callback.Connect
2     (Global_Window.all.Quit_Button,
3      "clicked",
4      Button_Callback.To_Marshaller (On_Quit_Button_Clicked'Access),
5      False);
6   -- XXX: window delete event should be also handled, but for simplicity
7   --       reasons, it is ommitted.
8
9   Button_Callback.Connect
10    (Global_Window.all.Run_Button,
11     "clicked",
12     Button_Callback.To_Marshaller (On_Run_Button_Clicked'Access),
13     False);
```

GtkAda

- Synchronizace mezi GUI vláknem a počítačím vláknem

```
2  protected Counter_Monitor is
3      procedure Start;
4      entry Wait_For_Start;
5      procedure Stop;
6      function Check_If_Stop return Boolean;
7      procedure Quit;
8      function Check_If_Quit return Boolean;
9      procedure Finished;
10     function Check_If_Running return Boolean;
11     entry Wait_Until_Finished;
12 private
13     Should_Start : Boolean := False;
14     Should_Stop : Boolean := False;
15     Should_Quit : Boolean := False;
16     Is_Running : Boolean := False;
17 end Counter_Monitor;
```

GtkAda

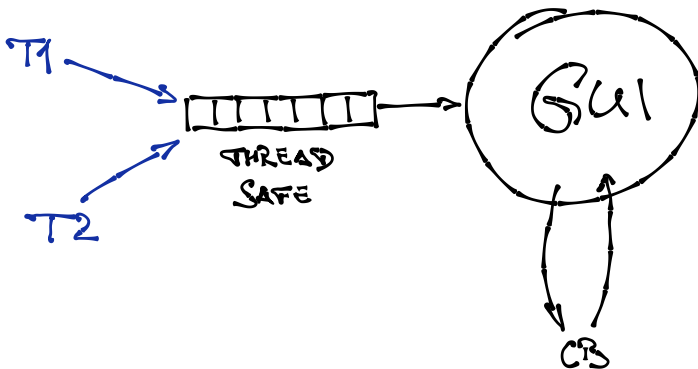
- Synchronizace mezi GUI vláknem a počítacím vláknem

```
1  protected body Counter_Monitor is
2      procedure Start is
3          begin
4              Should_Start := True;
5          end Start;
6
7      entry Wait_For_Start
8          when Should_Start or Should_Quit is
9          begin
10             Should_Start := False;
11             Should_Stop := False;
12             -- don't touch Should_Quit, so that it's persistent
13             Is_Running := True;
14         end Wait_For_Start;
```

Gtk

`http://www.gnu.org/software/guile-gnome/docs/gdk/html/Threads.html`

Asynchronní předávání do GUI



Asynchronní předávání do GUI

- Strategie
 1. vytvoříme smyčku obsluhy událostí GUI, která volá přímo jednotlivé callbacky
 2. vlákno, které chce aktualizovat GUI předá práci GUI smyčce
 3. do vlákna zpracovávajícího události můžeme zasílat blokujícím (`Request`) nebo neblokujícím (`Send`) způsobem
- Problémy
 - ověření, že při všech aktualizacích GUI používáme předání práce GUI vláknu

GtkAda a Gtk.Main.Router

- GtkAda Contributions:
http://www.dmitry-kazakov.de/ada/gtkada_contributions.htm
- Inicializace
- Vytvoření smyčky událostí
 - není třeba podpora vláken z Gdk

```
2  Gtk.Main.Set_Locale;  
   Gtk.Main.Init;  
   Gtk.Main.Router.Init;  
4  Init_GUI;  
   Gtk.Main.Main;  
6  return;
```

GtkAda a Gtk.Main.Router

- Zasílání z vlákna o smyčky událostí

```
1  task body Counter is
2  begin
3      Main_Loop :
4      loop
5          Counter_Monitor.Wait_For_Start;
6          -- if we don't exit here when appropriate, the application would
7          -- deadlock: the GUI task callback is waiting for us while we're
8          -- synchronously calling modification of the GUI
9          exit Main_Loop when Counter_Monitor.Check_If_Quit;
10         Gtk.Main.Router.Request (+Update_GUI_Button_Start'Access);
11         Counter_Loop :
12         for I in 0 .. 1000 loop
13             exit Main_Loop when Counter_Monitor.Check_If_Quit;
14             if Counter_Monitor.Check_If_Stop then
15                 exit Counter_Loop;
16             end if;
```

GtkAda a Gtk.Main.Router

- Zasílání z vlákna o smyčky událostí

```
2      declare
3          Label : aliased Unbounded_String := To_Unbounded_String (Integer'
4      begin
5          Update_GUI_Label_Callback.Request (Update_GUI_Label'Access, Label
6          end;
7          delay 1.0;
8      end loop Counter_Loop;
9      declare
10         NR : aliased Null_Record;
11     begin
12         -- this goes asynchronously
13         Update_GUI_Button_End_Handler.Send (Update_GUI_Button_End'Access, NR
14     end;
15     Counter_Monitor.Finished;
16 end loop Main_Loop;
17 Counter_Monitor.Finished;
18 end Counter;
```

GtkAda a Gtk.Main.Router

- Registrace volání
 - generika
 - komplikované kvůli typům

```
1  procedure Update_GUI_Button_Start is
2  begin
3      Gtk.Button.Set_Label (Global_Window.all.Run_Button, -" Stop ");
4  end Update_GUI_Button_Start;
5
6  type Null_Record is null record;
7  procedure Update_GUI_Button_End (NR : in out Null_Record) is
8      pragma Unreferenced (NR);
9  begin
10     Global_Window.all.Run_Button.all.Set_Label (-" Run ");
11     Global_Window.all.Run_Button.all.Set_Sensitive (True);
12 end Update_GUI_Button_End;
13
14 procedure Update_GUI_Label (Label_Access : access Unbounded_String) is
15 begin
16     Gtk.Label.Set_Label (Global_Window.all.Counter_Label, -To_String (Label_Access));
17 end Update_GUI_Label;
```

GtkAda a Gtk.Main.Router

- Registrace volání
 - generika
 - komplikované kvůli typům

```
1  -- this is ugly
2  type Local_Callback is access procedure;
3  function "+" is
4      new Ada.Unchecked_Conversion (Local_Callback, Gtk.Main.Router.Gtk_Callback);
5
6  -- this is better
7  package Update_GUI_Label_Callback is new Gtk.Main.Router.Generic_Callback_Request;
8
9  package Update_GUI_Button_End_Handler is new Gtk.Main.Router.Generic_Message_Handler;
```

Java SWING

- Multiplatformní GUI v Javě
- SWING single-thread rule
 - všechny prvky mohou být vytvářeny, měněny a dotazovány pouze z vlákna obsluhujícího události
 - `SwingUtilities.isEventDispatchThread` – kontrola, zda jsme ve vlákně obsluhující události
 - `SwingUtilities.invokeLater` – předávání `Runnable` do vlákna obsluhujícího události
 - `SwingUtilities.invokeAndWait` – předávání `Runnable` do vlákna obsluhujícího události a zablokuje se do dokončení akce
 - callbacky se řeší pomocí akcí `action listener` z vlákna obsluhujícího události
 - dlouho běžící callbacky je možno odštípnout do nového vlákna (přímo nebo přes `Executory`)

Java SWING

- Inicializace

```
1 public class JavaGUI {  
3     public static void main(String[] args) {  
5         CounterDialog dialog = new CounterDialog();  
6         dialog.setSize(400,300);  
7         dialog.setVisible(true);  
8     }  
9 }
```

Java SWING

- Předávání vláknu událostí SWINGu

```
2  @Override
   public void run() {
4     for (int i = 0; i <= 1000; i++) {
6         if (shouldShutdown.get()) {
           break;
       }
       final String labelText = String.valueOf(i);
       javax.swing.SwingUtilities.invokeLater(new Runnable() {
8           public void run() {
10              counterLabel.setText(labelText);
           }
12          });
       try {
14           Thread.sleep(1000);
       } catch (InterruptedException ignored) {
16          }
       }
18     runButton.setText("Run");
       runButton.setEnabled(true);
20 }
```

Java SWING

- Callbacky

```
1      buttonRun.addActionListener(new ActionListener() {  
2          public void actionPerformed(ActionEvent e) {  
3              onRun();  
4          }  
5      });
```

```
1      private void onRun() {  
2          if (!isRunning.get()) {  
3              buttonRun.setText("Stop");  
4              isRunning.set(true);  
5              if (counterThread != null) {  
6                  try {  
7                      counterThread.join();  
8                  } catch (InterruptedException ignored) {  
9                      }  
10             }  
11             counterThread = new CounterThread(counterLabel, buttonRun);  
12             counterThread.start();  
13         } else {  
14             buttonRun.setEnabled(false);  
15             counterThread.requestShutdown();  
16             counterThread.interrupt();  
17             isRunning.set(false);  
18         }  
19     }
```

QT

<http://doc.trolltech.com/4.7/threads.html>

- rozlišování reentrant vs. thread-safe
- `QThread` reprezentuje vlákno
- `QObject` je reentrantní, stejně jako řada odvozených tříd
- avšak odvozené GUI třídy `QWidget` reentrantní nejsou!
 - použití pouze z hlavního vlákna
- per-thread event loop
 - pro třídy jako `QTimer`, `QTcpSocket` – ne pro GUI

OpenGL

- Průmyslový standard specifikující multiplatformní rozhraní pro tvorbu aplikací počítačové grafiky
- Existuje v řadě verzí od 1.0 po poslední 4.0
- Aplikace využívající OpenGL je schopna ±běžet na různém HW
 - OpenGL podporuje různá rozšíření, která spolu se změnami verzí zhoršují portabilitu
 - Aplikace musí umět detekovat co daná platforma nabízí
- Rasterizaci objektů provádí obvykle HW, existují ale i SW rasterizátory (Mesa)

OpenGL pipeline

- OpenGL funkce jsou asynchronní
- OpenGL je stavová
- Návrh scény provádíme:

```
1 glBegin( GL_POLYGON );      /* Begin issuing a polygon */
2 glColor3f( 0, 1, 0 );      /* Set the current color to green */
3 glVertex3f( -1, -1, 0 );   /* Issue a vertex */
4 glVertex3f( -1, 1, 0 );   /* Issue a vertex */
5 glVertex3f( 1, 1, 0 );    /* Issue a vertex */
6 glVertex3f( 1, -1, 0 );   /* Issue a vertex */
7 glEnd();                  /* Finish issuing the polygon */
```

- Problematické při použití vláken

OpenGL kontext

- Kontext OpenGL není zachycen OpenGL specifikací, je tedy implementačně závislý
- Kontext uchovává stav a další informace pro rasterizér
- Existují rozšíření OpenGL pro manipulaci s kontexty
 - WGL – `wglCreateContext`, `wglMakeCurrent`, `wglDeleteContext`
 - GLX – `glXCreateNewContext`, `glXMakeCurrent`, `glXDestroyContext`
 - Pozor na použití `glXDestroyContext` – OpenGL je asynchronní!
- Pouze jeden kontext může být aktivní v rámci 1 vlákna
- Kontext je často uložen v TLS

OpenGL kontext a vlákna

- S implicitním kontextem
 - OpenGL na MacOS dovoluje přístup k GL funkcím z více vláken
 - Windows ohlásí resource busy
 - Linux (s Nvidia drivery) končí segmentation fault
 - Obecně je u vláken s implicitním kontextem problém, že kontext má právě master vlákno
 - Nelze triviálně předat kontext jinému vláknu (zejména u GLX)
- Různá vlákna si mohou vytvořit svůj kontext
 - Nemohou pak ale kreslit jednu společnou scénu přímo (lze přes nepřímý rendering do bufferu)
 - U WGL to často znamená, že každé vlákno kreslí do jiného okna

OpenGL kontext a vlákna

- Změna kontextu na jiné než master vlákno
 - Při použití `libSDL` triviálně tak, že zavoláme `SDL_init()` z ne-master vlákne, o zbytek se postará `libSDL`
 - Implicitní kontext vytváří GLX rozšíření Xserveru samo o sobě
 - Jediná cesta pro GLX je reload GLX z vlákna, které má kreslit
 - Návod lze najít např. ve zdrojových kódech `libSDL`
`src/video/x11/SDL_x11gl.c`

Futures & TPE v C++

- Futures nabízí knihovna `libboost`
<http://www.boost.org>
 - Nabízí ± stejná primitiva jako `pthread`
 - Navíc nabízí některé pokročilejší nástroje pro vlákna

```
1 #include <boost/thread.hpp>
2 #include <iostream>
3
4 using namespace std;
5
6 void hello ()
7 {
8     cout << "Pocitam." << endl;
9 }
10
11 int main(int argc, char* argv[])
12 {
13     boost::thread thrd(&hello);
14     thrd.join();
15     return 0;
16 }
```

Futures & TPE v C++

- Futures

```
1 #include <boost/thread.hpp>
2 #include <iostream>
3 #include <string>
4
5 using namespace std;
6
7 string hello()
8 {
9     cout << "FT: Pocitam." << endl;
10    sleep(5);
11    cout << "FT: Vypocet hotov." << endl;
12    return "12345";
13 }
14
15 int main(int argc, char* argv[])
16 {
17     boost::packaged_task<string> pt(hello);
18     boost::unique_future<string> fi=pt.get_future();
19
20     boost::thread task(boost::move(pt));
21     cout << "Master: vytvoreno vlakno." << endl;
22     cout << "Master: Cekam na vysledek." << endl;
23     fi.wait();
24     cout << "Mam vysledek: " << fi.get() << endl;
25     return 0;
26 }
```

Futures & TPE v C++

- Samotná knihovna `libboost` nemá threadpools
- <http://threadpool.sourceforge.net/> implementuje threadpools v C++ nad `libboost`

Futures & TPE v C++

Anybody who tells me that STL and especially Boost are stable and portable is just so full of bullshit that it's not even funny.

– *Linus Torvalds*

Java Native Interfaces

- Volání nativních metod z Javy
- Struktura JNI volání

```
2  /* C */
3  JNIEXPORT void JNICALL Java_ClassName_MethodName
4      (JNIEnv *env, jobject obj, jstring javaString)
5  {
6      //ziskani nativniho retezce z javaString
7      const char *nativeString = (*env)->GetStringUTFChars(env, javaString, 0);
8      ...
9      //nezapomenout uvolnit!
10     (*env)->ReleaseStringUTFChars(env, javaString, nativeString);
11 }
12
13 // C++
14 JNIEXPORT void JNICALL Java_ClassName_MethodName
15     (JNIEnv *env, jobject obj, jstring javaString)
16 {
17     //ziskani nativniho retezce z javaString
18     const char *nativeString = env->GetStringUTFChars(javaString, 0);
19     ...
20     //nezapomenout uvolnit!
21     env->ReleaseStringUTFChars(javaString, nativeString);
22 }
```

Vlákna a JNI

- Ukazatel na `JNIEnv` je platný pouze z vlákna, jemuž je přiřazen
 - nelze předávat mezi vlákny
 - stejný při opakovaných voláních v témže vlákne
- Lokální reference nesmí opustit vlákno
 - lokální reference jsou platné pouze v rámci daného volání
 - ◆ nelze se je uschovávat ve `static` proměnných
 - převést na globální, pokud je třeba (`NewGlobalRef`)
 - globální reference vylučují objekt z garbage collection
 - existují slabé lokální reference (`NewWeakGlobalRef`)
 - ◆ umožňují garbage collection odkazovaného objektu
 - ◆ potřeba kvůli class unloading
 - ◆ musí se kontrolovat, zda odkazovaný objekt existuje (`IsSameObject` s `NULL` parametrem)

Vlákna a JNI

```
2  static jclass stringClass = NULL;
3  ...
4  if (stringClass == NULL) {
5      jclass localRefCls =
6          (*env)->FindClass(env, "java/lang/String");
7      if (localRefCls == NULL) {
8          return NULL; /* exception thrown */
9      }
10     /* Create a global reference */
11     stringClass = (*env)->NewGlobalRef(env, localRefCls);
12
13     /* The local reference is no longer useful */
14     (*env)->DeleteLocalRef(env, localRefCls);
15
16     /* Is the global reference created successfully? */
17     if (stringClass == NULL) {
18         return NULL; /* out of memory exception thrown */
19     }
20 }
21 ...
22 // potreba explicitne mazat
23 if (terminate) {
24     (*env)->DeleteGlobalRef(env, stringClass);
25 }
```


Vlákna a JNI

- Použití monitorů
 - vždy monitor uvolnit

```
1 synchronized (obj) {  
2     ... // synchronized block  
3 }
```

```
1 if ((*env)->MonitorEnter(env, obj) != JNI_OK) ...;  
2 ...  
3 if ((*env)->ExceptionOccurred(env)) {  
4     ... /* exception handling */  
5     /* remember to call MonitorExit here */  
6     if ((*env)->MonitorExit(env, obj) != JNI_OK) ...;  
7 }  
8 ... /* Normal execution path. */  
9 if ((*env)->MonitorExit(env, obj) != JNI_OK) ...;
```

Zdroj: JNI Book

Vlákna a JNI

- Wait/Notify
 - generické volání metod (`GetMethodID`, `CallVoidMethod`)
 - potřeba držet monitor

```
1  /* precomputed method IDs */
2  static jmethodID MID_Object_wait;
3  static jmethodID MID_Object_notify;
4  static jmethodID MID_Object_notifyAll;
5
6  void
7  JNU_MonitorWait(JNIEnv *env, jobject object, jlong timeout) {
8      (*env)->CallVoidMethod(env, object, MID_Object_wait, timeout);
9  }
10
11 void
12 JNU_MonitorNotify(JNIEnv *env, jobject object) {
13     (*env)->CallVoidMethod(env, object, MID_Object_notify);
14 }
15
16 void
17 JNU_MonitorNotifyAll(JNIEnv *env, jobject object) {
18     (*env)->CallVoidMethod(env, object, MID_Object_notifyAll);
19 }
```

Vlákna a JNI

- Explicitní získání `JNIEnv` pro stávající vlákno
 - např. callback volaný OS
 - odkaz na `JavaVM` lze předávat mezi voláními a vlákny
 - získání odkazu např. `JNI_GetCreatedJavaVMs` nebo `GetJavaVM`

```
1  JavaVM *jvm; /* already set */  
3  f() {  
    JNIEnv *env;  
5   (*jvm)->AttachCurrentThread(jvm, (void **)&env, NULL);  
    ... /* use env */  
7  }
```

Zdroj: JNI Book

Vlákna a JNI

- Mapování vláknového modelu OS a JVM
 - záleží, zda pro danou platformu JVM podporuje nativní vlákna
 - závisí od dané platformy i od daného JVM
- Můžeme implementovat v Javě afinitu k CPU
 - závisí od dané platformy i od daného JVM
 - `sched_setaffinity(gettid(), ...)`
 - v praxi může pro danou platformu dobře fungovat

Systémy real-time (RT)

- Požadavky na RT systémy
 - reakce programu v definovaném termínu (deadline)
 - *hard real-time*: systém musí vždy reagovat do deadline
 - *real real-time*: hard-RT s velmi krátkou reakční dobou (řízení střel)
 - *soft real-time*: systém může občas zmeškat deadline
 - *firm real-time*: soft-RT, kdy opožděně poskytovanou službu už nemá smysl poskytovat
- Interakce s OS
 - real-time OS
 - embedded bez OS
- Paralelismus
 - většinou se pracuje se sekvenčními programy
 - co když paralelismus potřebujeme?

Ada: vlákna a RT systémy

- Zajímavé úrovně jazyka
 - zero footprint (ZFP): bez vláken (např. jednodušší certifikace pro DO-178B Level A)
 - Ravenscar profile: omezené interakce vláken (lze lépe dělat statickou analýzu kódu)
 - plnohodnotný přístup k vláknům
- Ada Annex D: Real-Time Systems
 - podpora monotónních hodin s vysokým rozlišením
 - politiky spouštění vláken (task dispatching)
 - priority vláken vč. dynamických priorit
 - politiky řazení do front entries
 - (preemptivní abort)
 - synchronní řízení vláken
 - asynchronní řízení vláken
 - omezení vláken (`pragma Restrictions`)
- Ada Annex H: High-Integrity Systems, Ada 2005 Annex M: Summary of Documentation Requirements, jazyk SPARK

Plánování vláken

- Plánování
 - nalezení takového pořadí vykonávání úloh, aby byly splněny termíny
 - analýza plánovatelnosti
- Typické úlohy v RT systémech
 - periodické – opakované v pravidelných intervalech (`delay until ...;`)
 - sporadické – reakce na události (typicky čekající pomocí blokování na chráněném objektu)

Preemptive Fixed Priority Dispatching

- Podpora i před Ada 2005
- Task dispatching
 - proces výběru vlákna, které má běžet na daném procesoru
- Dispatching points
 - místa v kódu, kde dochází k přepínání
 - vždy: blokování na volání, ukončování úlohy
 - další místa jsou definována annexem pro specifické politiky (např. `delay 0.0;` pro nepreemptivní plánování)
- `pragma Task_Dispatching_Policy(FIFO_Within_Priorities);`
 - nastavuje se per partition (Distributed Annex)

Preemptive Fixed Priority Dispatching

- 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 \implies preempivita
 - kdykoli se v kódu objeví `delay 0.0;`

Priority ceiling locking

- Strop priority (priority ceiling)
 - *Immediate Ceiling Priority Protocol (ICPP)* neboli *Ceiling Priority Emulation* neboli *Priority Protect Protocol*
 - maximální priorita, s níž mohou být daná úloha či chráněný objekt volány
 - při vstupu do volání úloha získá úloha stropní prioritu
 - po ukončení volání se úloha vrátí ke své původní prioritě
 - jsou-li fakticky zavolány s vyšší prioritou než je strop, je to považováno za chybu programu
- `pragma Locking_Policy(Ceiling_Locking);`

Priority ceiling locking

- Změna chování H/M/L/P systému:
 1. spustí se úloha L a vstoupí do P; tím získá prioritu (nejméně) H
 2. M je nachystáno, ale nemůže přerušit L protože má nižší okamžitou prioritu
 3. H je nachystáno, ale nespustí se, protože nemá vyšší okamžitou prioritu než L
 4. L opustí P a je vráceno na původní prioritu
 5. H je spuštěno

Ostatní

- Odevzdávání „projektu“
 - Odevzdavarny/Projekt
 - UCO-Jmeno-Prijmeni.{zip,tgz,...}
- Organizace zkoušky
 - diskuse řešení projektu
 - minimálně 2 obecné otázky
 - minimálně 1 specializační (Java/C/Ada)