Výpočty v reálném čase

- Z důvodu snazší orientace v odborné literatuře o RT, která je anglicky, je uvedený text směs angličtiny a češtiny
  - Některé termíny by prostě překladem utrpěli příliš velkou ztrátou přesnosti pro někoho, kdo se ještě v problematice RT dostatečně nevyzná

- V tradičním OS vlákna běží nějakou rychlostí a výpočet se někdy dopočítá
- Sice nám to vyhovuje na řadu úkolů, ale ne na všechny

- Non-Real Time
  - Vlákna nemají stanovenu žádnou dobu, deadline, do kdy musí dokončit výpočet
  - A to ani tehdy, když od nich požadujeme rychlou odezvu
    - Jak napíšete slovo ve Wordu, už aby ho zároveň zkontroloval a odstranil překlepy

- Real Time
  - Dokončení výpočtu ve stanoveném termínu je kritické
  - Termín musí být dodržen bez ohledu na zátěž systému
  - Brzdy v autě, vojenské systémy, podpora života, jaderné zařízení, …
    - Ale i řízení výroby a mobilní telefony

- Hard Real Time
  - Dokončení výpočtu po termínu se považuje za chybu a výsledek za bezcenný – strict deadline
  - Nedodržení termínu může vést k celkovému selhání systému
    - Airbag, řízení motoru (nejenom auto má motor), jaderné zařízení
  - Jsou vyžadovány tam, kde hrozí příliš velké škody v případě selhání systému

- Soft Real Time
  - Překročení termínu se toleruje, systém reaguje zhoršenou kvalitou poskytovaných služeb
    - Vypadne pár snímků
    - Přílet letadla se dozvíte s několika sekundovým zpožděním

- Výkonnost
  - Systémy reálného času nejsou vysoko-výkonnostní výpočetní systémy
    - Nejde o to, vypočítat toho co nejvíc, ale vypočítat to včas
  - Výpočetně náročné úloze se vždy hodí vyšší výkon
  - Pokud je úloha schopná dodržet časové limity, není důvod ke zvyšování výkonu – není toho třeba

- Plánování
  - Buď cyklický plánovač
    - Neosvědčil se díky velké režii při přiřazování priorit úlohám v systému – selhával
  - Prioritní schéma
    - Odstraňuje nedostatky cyklického plánovače
    - RMA

Výběr algoritmu

- Abychom věděli, do kdy algoritmus vrátí výsledek, musíme znát nejhorší možný čas výpočtu
  - Pokud se výsledek spočítá dříve, než je ho třeba, musíme jen zajistit, aby byl dostupný v době, až ho bude třeba

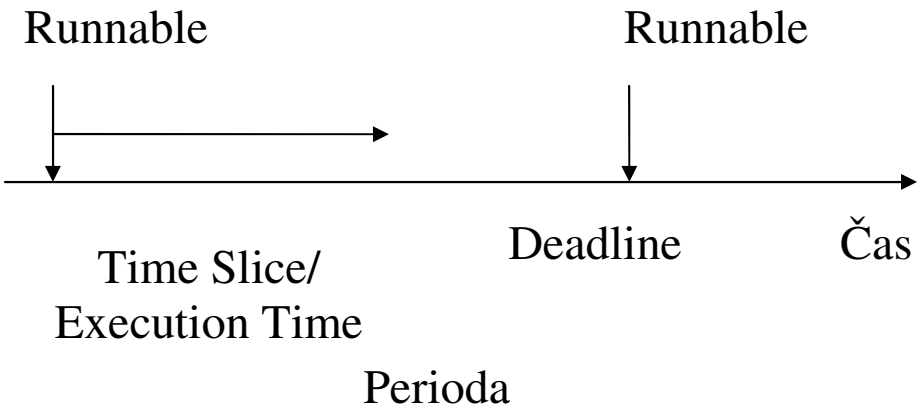
- Například u třídění
  - QuickSort
    - Nejlepší možná složitost je O($N * log\ N$)
    - Nejhorším případem je O (N^2)
  - HeapSort
    - V praxi je o něco pomalejší než QuickSort
    - Ale má zaručenou O($N * log\ N$)
    - Tj. pro RT je, obecně vzato, lepší volba než QuickSort
  - Další kritéria mohou být
    - Spotřeba paměti
    - Možnost a významnost použití paralelizace
      - Např. Bitronic Merge Sort
    - Speciální podmínky
      - Např. Counting a Radix Sort

- Například u evolučních algoritmů
  - Jsou dobré tam, kde vlastně nevíme, jak to vypočítat
  - Ale do kdy nám vrátí požadovaný výsledek?
  - A vrátí nám vůbec použitelný výsledek?
    - Respektive, vrátí ho včas?

RMA

- Rate Monotonic Analysis
- Plánovací algoritmus systému reálného času – Rate Monotonic Scheduling
- Co bychom intuitivně nazvali vláknem se označuje jako úloha (task)
- Přiřazuje priority jednotlivým úlohám tak, aby stihly dokončit výpočet v termínu
    - Možným výsledkem je, že se zjistí, že to úloha nemůže stihnout

- Periodická úloha
    - Je opakovaně/periodicky runnable v pevně daných intervalech
        - Respektive pro korektní činnost systému musí být (hard real time) runnable
            - Měla by být – soft real time
        - Při korektní činnosti systému je spuštěna
            - Tj. systém neselhal – hard real time
            - Nedošlo ke zpoždění – soft real time
    - Perioda
        - Časové rozmezí, kdy je úloha runnable
    - Deadline – začátek další periody



- Priorita
  - Je odvozena od délky periody
  - Čím kratší perioda, tím
    - Higher request rate – tj. úloha běží častěji (s větší frekvencí)
    - => Větší priorita úlohy
  - Ratio grid – rozdělit interval <min_period, max_period> aby byl poměr mezi sousedy stejný: 1ms, 2ms, 4ms, 8ms…
  - Ideálně tolik úrovní priorit, kolik je třeba
    - Není vždy možné

- Plánování
  - Lui a Layland dokázali, že uvedený způsob přiřazování priorit je optimální v tom smyslu, že je-li množina úloh naplánovatelná tak, aby žádná z nich nepřekročila deadline, pak je naplánovatelná s rate-monotonic plánováním

  - V některých systémech mohou zajišťovat kritické úkoly úlohy s dlouhou periodou
    - tj. s nízkou prioritou, díky které by nebyly dostatečně často naplánovány
    - řešením je rozdělit takovou úlohu do několika menších s kratší periodou

- Systém je naplánovatelný tehdy
  - C_i/T_i je využití procesoru i-tou úlohou z n úloh
  - Lui a Layland dokázali, že

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(\sqrt[n]{2} - 1\right)$$

- U – využití procesoru
- n – počet úloh
- C_i – výpočetní čas úlohy
- T_i – perioda úlohy

$$\lim_{n \to \infty} n\left(\sqrt[n]{2} - 1\right) = \ln 2 \approx 0{,}693147...$$

- Udržíme-li zatížení procesoru n úlohami pod 70%, pak je možné naplánovat všechny úlohy tak, aby dodržely své deadlines
- To neznamená, že není možné najít takové plánování, které bude mít stejnou vlastnost při větším zatížení systému
- Zbývajících 30% mohou být non-real time threads

- Původní RMA měla řadu omezení, jejichž dodržení garantovalo, že daný systém bude naplánovatelný a deadlines dodržitelné
- Postupem výzkumu se ukázalo, že původní omezení byla příliš striktní a že bylo možné slevit z jejich nároků a přesto dosáhnout naplánovatelný systém s dodržením deadlines

- Zatížení 70% můžeme použít jako rychlý test, zda je systém naplánovatelný

- Pokud to s ním nevychází, pak lze použít Response Time Test

  - Pro každou úlohu rekurzivně vypočítáme R_k
    - Jako u každé iterační metody počítáme tak dlouho, dokud je rozdíl posledních dvou výsledků mimo nějaký rozsah

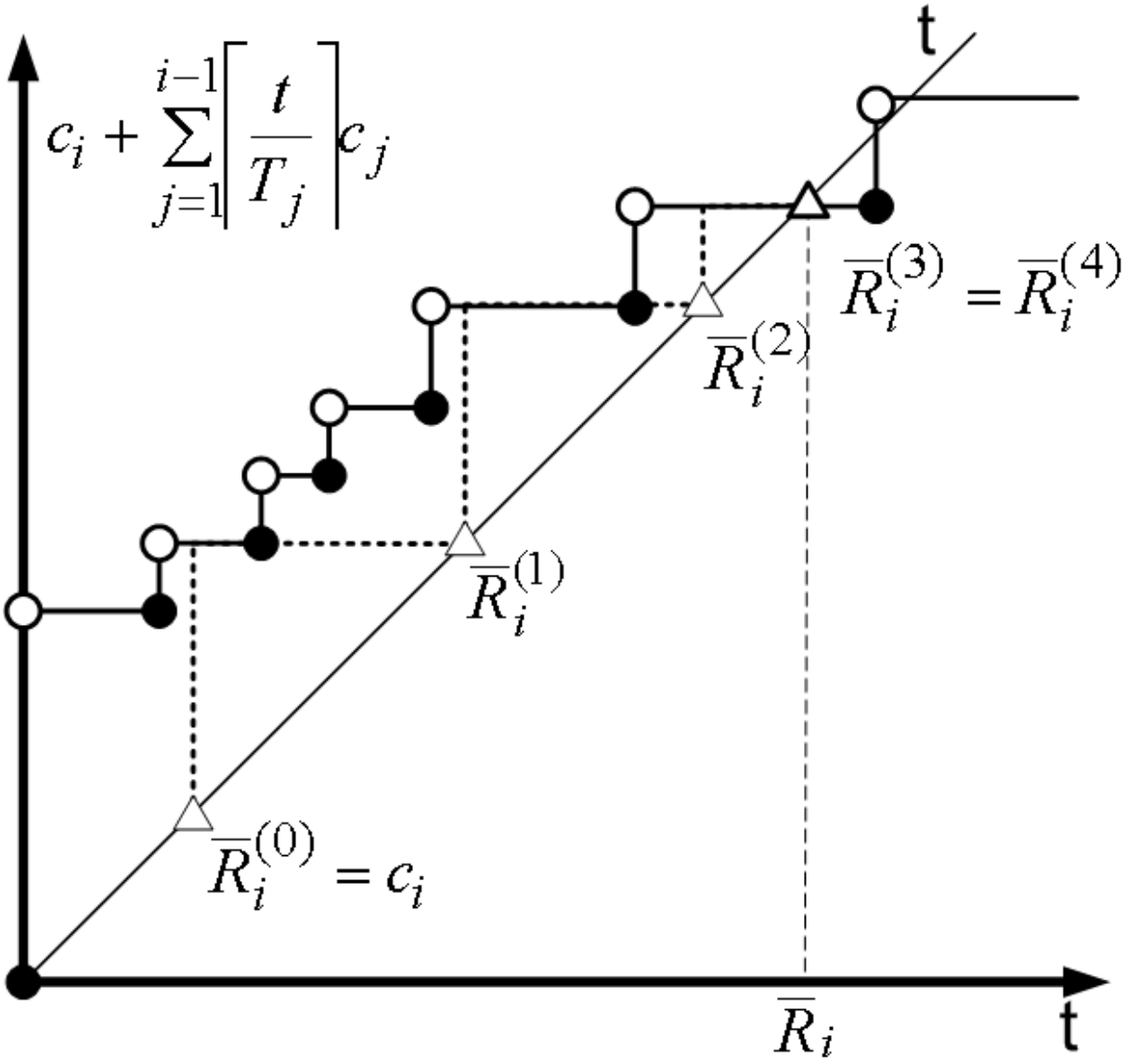
$$R_0 = \sum_{j=1}^{i} C_j$$

    - Úlohy jsou seřazené podle priorit
      - Tj. provede se součet všech úloh s vyšší prioritou – kratší výpočetní čas znamená kratší periodu

$$R_{k+1} = C_i + \sum_{j=1}^{i-1} C_j \times ceil\left(\frac{R_k}{T_j}\right)$$

    - Algoritmus funguje tak, že se snaží navýšit výpočetní čas úlohy na základě času, který zaberou úlohy s vyšší prioritou
    - Iterace končí v okamžiku, kdy se výpočetní čas úlohy přestane navyšovat

  - jestliže R_k<deadline úlohy, pak úlohu lze naplánovat
    - R_k – response time



- Manuel Coutinho, José Rufino, Carlos Almeida, "Response Time Analysis of Asynchronous Periodic and Sporadic Tasks Sheduled by a Fixed Priority Preemptive Algorithm," ecrts, pp. 156-167, 2008 Euromicro Conference on Real-Time Systems, 2008

- Aperiodické a sporadické úlohy
  - Zpracování událostí
    - např. hw přerušení
  - Inicializace
  - Zotavení se z chyby systému
  - Příkaz/požadavek operátora

  - Soft deadline
    - Změna parametrů radaru
    - Dá se tolerovat zpoždění v reakci na příkaz operátora
    - Co nejmenší zpoždění je žádoucí, ale nikdy na úkor úloh s hard deadline
    - Aperiodická úloha
      - Dokonce vůbec nemusí mít deadline

  - Hard deadline
    - Pokyn pilota
    - Při některých manévrech může být zpoždění smrtelné
    - Sporadická úloha

  - Plánování
    - V systému je několik serverů, které vykonávají aperiodické/sporadické úlohy
    - Servery jsou periodické úlohy a jsou plánovány podle pravidel RMA
    - Sporadické úlohy jsou nejprve setříděny podle EDF
      - Earliest Deadline First

- Synchronizace – kritická sekce
  - Nebezpečí
    - Úloha s nízkou prioritou může zablokovat úlohu s vyšší prioritou
    - Původně se proto uvažovala omezující podmínka, že se úlohy nemohou vzájemně blokovat – např. mutexem

    - Úloha s nízkou prioritou je v kritické sekci
    - Úloha s vysokou prioritou chce také do kritické sekce, ale je zablokována
    - Úloha se střední prioritou přeruší vykonávání úlohy s nízkou prioritou a tak zablokuje úlohu s vysokou prioritou
      - Nestíhá se deadline, selhání systému

  - Inverze priorit
    - Z úlohy s nízkou prioritou, která drží zámek, se dočasně stane úloha s vysokou prioritou
    - Úloha, která má normálně vysokou prioritou, dokončí výpočet o něco později
      - Většinou se to stihne
      - Ale ne vždy a může to vést k závažným problémům
        - Např. úloha s vysokou prioritou může být kontrolní a pokud včas nezpracuje informace, může současný stav vyhodnotit jako selhání a resetovat celý systém
          - Mars Pathfinder – díky inverzi priorit úlohy nedokončily výpočet v očekávaném pořadí a výsledkem byl reset systému

- Zákaz všech přerušení
  - Tj. i hodin
  - Jedna úloha je absolutním pánem systému
  - Nicméně, kritická sekce musí být krátká, jinak systém selže i tak
  - Používá se v malých systémech
  - Nehodí se pro general-purpose

- Dědičnost priorit
  - Úloha s nižší prioritou vykonává kritickou sekci
  - Úloha s vyšší prioritou se pokusí o vstup do kritické sekce
  - Úloha v kritické sekci dostane prioritu čekající úlohy s nejvyšší prioritou
  - => žádná úloha se střední prioritou nezablokuje úlohu s nejvyšší prioritou
  - Stejně ale dědičnost priorit negarantuje,
    - Že nedojde k uvíznutí,
    - Ani že nevznikne řetěz postupně blokovaných úloh

- Priority Ceiling
  - Ví se, jaké zdroje chráněné kritickými sekcemi budou úlohy požadovat

  - Plánování
    - Každá úloha běží s přidělenou prioritou, dokud nechce vstoupit do kritické sekce
    - Úloha dostane nejvyšší prioritu ze všech úloh, které kdy budou chtít vstoupit do stejné kritické sekce
      - Viz znalostní podmínka

  - Alokace – úloha uzamkne zámek kritické sekce, jak si o něj požádá

  - Nedochází k uvíznutí
    - V okamžiku, kdy je v kritické sekci, neexistuje žádná jiná úloha, která by ji mohla přerušit

  - Stále však může dojít k vyhladovění úlohy s vyšší prioritou úlohou s nižší prioritou
    - Příliš dlouho se v kritické sekci vykonávající úloha s nízkou prioritou zdrží úlohu s vyšší prioritou natolik, že ta nestihne deadline

- Synchronizace – zprávy
  - Možnost synchronizace, kdy nedojde k uvíznutí díky zámku kritické sekce
  - Mohou být generovány rychleji než zpracovány
  - Zpráva může obsahovat synchronizační informace

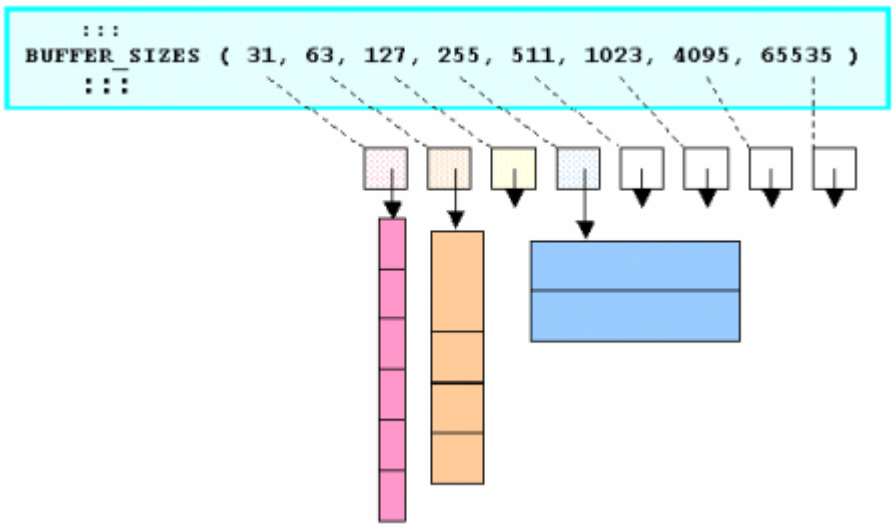
  - Většina OS zprávu kopíruje 2x
    - Z paměti odesílajícího do fronty zpráv spravované OS v jeho části paměti
    - Z fronty zpráv OS do paměti příjemce

  - Může být pomalé pro RT
    - Lze zrychlit pouze převedením části paměti odesílajícího pod příjemce a předat pointer
    - Úlohy ale musí s takovou praktikou RTS počítat
      - Nelze „jen tak“ převést kus paměti

- Správa paměti
  - Na rozdíl od non-real time systému je to kritická záležitost i z hlediska času
    - Vzpomeňte si na swapující wokna
    - Nicméně, problém už je v samotném paměťovém manažeru

    - Paměť se používáním malloc/free fragmentuje a nějaký čas trvá, než se najde vhodný volný blok paměti při volání malloc
      - Prostě se to nestihne do deadline
      - RTS taky může běžet několik let bez restartu, ne jako běžný počítač, který se stejně jednou po čase restartuje

- Defragmentace
  - U defragmentačních algoritmů nelze předpovídat, jak dlouho jim to potrvá
  - V systému se projevují jako náhodně načasované zatížení procesoru
    - Tj. zvyšují riziko, že úlohy nestihnou dodržet deadlines
  - Dilema
    - Odmítnout malloc, aby se zamezilo fragmentaci, i když je dostatek paměti,
    - Nebo povolit občasné ubírání výkonu?
- V jednoduchosti je síla
  - dva seznamy
  - jeden volné, druhý obsazené, bloky paměti o stejné velikosti
  - dvojic seznamů může být několik podle velikosti bloků
    - bloky se už dále nedělí – nefragmentují
  - Memory Pool



http://www.linuxdevices.com/files/misc/kalinsky_whitepaper_files/figure7.gif

RTOS

- Real Time Operating System
- Umožňuje vytvořit systém reálného času
- Sám o sobě nezaručuje, že výsledky budou vypočítány včas
    - To je úkol programátora, aby vytvořil správný program

- Umožňuje dodržet termíny
    - Obecně – Soft Real Time
    - Deterministicky – Hard Real Time

- Na rozdíl od tradičního OS, nejde na prvním místě o to, aby se zpracovalo co nejvíce dat, ale aby
    - bylo možné rychle reagovat na různé události s dopředu známou dobou trvání
    - se dosáhlo minimální režie při přepínání úloh
        - pochopitelně se to odrazí i v době zpracování přerušení

- Plánování úloh
    - Událostně řízené
        - úloha se přepne pouze tehdy, když nastane událost s vyšší prioritou, než má běžící úloha
        - Kooperativní multithreading – úloha se po nějaké době dobrovolně vzdá procesoru

    - Sdílení času
        - tj. virtualizace procesoru
        - úlohy se přepínají nejenom událostmi, ale i podle hodin

- Barrel Procesor
  - Hw garantuje, že v každém cyklu vykoná jednu instrukci z N běžících vláken jednou za N cyklů
  - Nulová režie přepínání vláken
  - I kdyby některé vlákno uvízlo, nebo zpracovávalo příliš přerušení, další vlákna běží dál ve stanovených časech
  - N je konečné a vysoká N jsou nákladná na design i na výrobu

- Earliest Deadline First
  - Vlákna jsou v prioritní frontě
  - Jakmile dojde k události jako vytvoření, či dokončení vlákna, fronta se prohledá a vyberou se vlákna s nejbližším termínem
    - Z nich se pak vybere podle priority
  - Pokud nároky vláken nepřesahují 100% výkonu procesoru, EDF je umí naplánovat tak, aby se všechny vykonaly včas
  - V opačném případě nelze určit, kolik vláken nestihne dodržet termín

- Monotonic scheduling
  - Viz RMA

RTS bez RTOS

- To use, or not to use a full-fledged RTOS?
  - That's the question. Obviously :-)

- Dostatečně malý projekt se může obejít bez velkého RTOS, potřebujeme-li jeho funkčnost v množství menším než malém
  - Ale co je dostatečně malý projekt?

- Například pro ledničku, kde jenom monitorujeme teplotu a rozsvěcíme/zhášíme žárovku, není RTOS nezbytně nutný

- Ale co když ji necháme ovládat hlasem, regulovat teplotu, zobrazovat informace na display, necháme ji automaticky doobjednávat pivo, jak ho dopijeme a navíc ještě umožníme vypočítávat, kolik promile máme v krvi?

- Například pokladní systémy v hypermarketech také běhaly pod DOSem, i když všichni měli doma WXP
  - Ovšem tohle není ze světa RTS

- Jak se rozhodnout?
  - RTOS má také sám o sobě nějakou režii
  - Stejně jste to vy jako programátoři, kdo se musí postarat, aby se stíhali deadlines
  - Pokud se bude RTS někdy portovat, musí být v prvé řadě možné portovat RTOS
  - Scheduler RTOS může pěkně zkomplikovat ladění
  - Cena některých RTOS
  - V okamžiku, kdy si sami musíte naprogramovat většinu funkcí RTOS, je načas uvažovat o RTOS

- Lednička bez RTOS

```
int main (void) {
  InitSystem();

  for (;;) {
    ScanTemperature();
    UpdateLCD();

    CheckDoor();
    TurnLightOnOff();
    //hi-tech - normálně to jde mechanicky:)

    Pause(1000); //=> menší spotřeba energie
  }

  //co bychom neudělali pro blaho překladače
  return 0;
}
```

- Úlohy běží serializovaně
- Komunikace mezi úlohami lze realizovat pomocí globálních dat, která není nutné hlídat s kritickou sekcí
- Kooperativní multitasking

- Ultra-Hi-Tech Lednička
  - Umí automaticky objednávat pivo
    - Tzn. komunikuje se světem
  - Internet => TCP/IP => window
    - Data netečou konstantní rychlostí ani jedním směrem
      - Tj. potřebujeme data zpracovávat různou rychlostí
      - => event driven tasks – aperiodické

- Uživatelsky příjemné prostředí
  - Za 40s po zavření dveří displej pomalu zhasne
  - Při zadávání hodnot bude blikat kurzor
  - Průběžná aktualizace zobrazovaných hodnot

  - Vytvořit pro každou činnost samostatnou úlohu a tu volat z hlavní smyčky je zbytečné přetěžování systému
  - Efektivní je použít timer a z jeho obsluhy aktivovat jednotlivé úlohy

  - Pro display postačí 40s
  - Kurzor bliká s frekvencí 0,5s
  - Aktualizace bude chtít 1s

  - Ani zde však není nutné aktivovat úlohy častěji, než je nezbytné
    - Pokud je timery řízených událostí více, může to být další důvod, proč uvažovat o RTOS

- Mobilní telefon
  - Příklad velikosti projektu z praxe, kdy je to na rozhraní, zda použít RTOS, či nikoliv
  - Některé ho mají, některé ne

- Třetí možnost
  - Základní funkce OS stejně musíte napsat, možná tak časem váš vlastní kód vytvoří váš vlastní RTOS
  - In-House RTOS může být dokonce podmínkou
    - Digital Battlefield
      - JSTARS, F22, AH-64D, M1A2, Predator

Java Real-Time System

- Java neumí
  - Používat striktně prioritní plánování vláken
    - Tím pádem zámky Javy nemohou umět inverzi priorit, priority ceiling, atd.
  - RT vhodnou správu paměti
    - Garbage collector představuje náhodné zatížení systému a pozastavování vláken, takže se nedá garantovat dodržení deadlines

- Jako odpověď vznikla Real Time Specification for Java
  - RTSJ
  - implementace RTSJ
    - Java Real Time System
    - Timesys

  - Specifikuje minimální požadavky na správu vláken
  - Různé modely plánování vláken je možné doinstalovat do JVM

  - Část paměti lze vyloučit z aktivit garbage collectoru
  - Vybraná vlákna lze označit za nepřerušitelná garbage collectorem

  - Původní verze příkladu
    - http://www.cs.york.ac.uk/rts/CRTJbook/missile
    - Všimli jste si někdy, v kolika filmech Kirk s Picardem zničili Enterprise a jak často nařídili autodestrukci?
      - Také příklad RTS

```
import javax.realtime.*;

public class Enterprise extends RealtimeThread{

 public Enterprise(PriorityParameters pp,
                   MemoryArea ma) {
   super(pp, null, null, ma, null, null);
 }
 public void run() {
    AutoDestruct autoDestructControl =
                           new AutoDestruct();
    Crewman captain = new Crewman
        (autoDestructControl, 1,
         "picard, delta, omega, alfa",
          new PriorityParameters(
              PriorityScheduler.MAX_PRIORITY),
           new VTMemory(1024));


    Crewman firstOfficer = new Crewman
        (autoDestructControl, 2,
         "riker, theta, epsilon, gamma",
          new PriorityParameters(
              PriorityScheduler.MAX_PRIORITY),
           new VTMemory(1024));

    captain.start();
    firstOfficer.start();
  }
  public static void main(String [] args)  {
    Enterprise starShip =
      new Enterprise (
      new PriorityParameters(
          PriorityScheduler.MAX_PRIORITY),
      ImmortalMemory.instance());
     starShip.start();
  }
}
```

```
public class Crewman
       extends NoHeapRealtimeThread {

  private AutoDestruct myController;
  private String authorization;
  private int myRole;

  public CrewMan(AutoDestruct controller,
                 int role, String code,
                 PriorityParameters pp,
                 MemoryArea ma) {
    super(pp, ma);
    myController = controller;
    authorization = code;
    myRole = role;
  }

  public void run() {

   boolean result;

   if (myRole == 1)
    result = myController.init1(authorization);
    else
      result=myController.init2(authorization);

   myController.AudioWarnings(awOff);
       //We're expecting a boarding party!
   try {
   //30 minutes to deal with the boarding party
     sleep(1800000);
   } catch(InterruptedException ie) {
     myController.abortSequence(authorization);
   };
  }
}
```

```
public class AutoDestruct{

  private LTMemory shared;
  private AutoDestructAction adController1,
                             adController2;
  private ImmortalAction immortalController1,
                         immortalController2;


  public AutoDestruct () {
    adController1 = new AutoDestructAction();
    adController2 = new AutoDestructAction ();
    immortalController1 = new ImmortalAction();
    immortalController2 = new ImmortalAction();
    SizeEstimator s = new SizeEstimator();
    s.reserve(Decrypt.class,2);
    s.reserve(Barrier.class,1);
    shared = new LTMemory(s.getEstimate(),
                          s.getEstimate());
  }
```

```
class AutoDestructAction implements Runnable {

   String authorization;
   boolean result;

   public void run() {
     Barrier sync;
     Decrypt check = new Decrypt();
     boolean confirmed =
                check.confirm(authorization);

    //We need to use two authorization codes
    //=> access to shared memory => protection
    synchronized(
      RealtimeThread.getCurrentMemoryArea()) {

      sync = (Barrier) shared.getSync();

      if (sync == null) {
         try {
           sync = new Barrier(2); //2 officers
           shared.setSync(sync);
         } catch(InterruptedException ie) {};
      } //if
    } //synchronized

   result = sync.waitB(confirmed);
  }
}

class ImmortalAction implements Runnable {
   AutoDestructAction adController;
   public void run() {
     //enter shared scoped memory
     shared.enter(adController);
   }
}
```

```
  public boolean init1(final String
                       authorizationCode) {
    try {
      immortalController1.adController =
                          adController1;
      adController1.authorization =
                         authorizationCode;
      ImmortalMemory.instance().
         executeInArea(immortalController1);
    } catch(Exception e) {};

    return adController1.result;
  }


  public boolean init2(final String
                       authorizationCode) {
    try {
      immortalController2.adController =
                          adController2;
      adController1.authorization =
                         authorizationCode;
      ImmortalMemory.instance().
         executeInArea(immortalController2);
    } catch(Exception e) {};

    return adController2.result;
  }

}
```