# Mikroprocesory

## 9. RTOS

Stanislav Vítek Katedra radioelektroniky České vysoké učení technické v Praze

## Obsah přednášky

1. Real-time systémy
2. Operační systémy reálného času
3. Architektura RTOS
4. Návrh vlastního RTOS pro SMTM32F4

## Problém

V (embedded) systém potřebujeme dělat víc věcí najednou

- číst senzory
- posílat data přes UART
- dělat regulaci
- občas logovat něco do flash
- hlídat timeouts
- obsluhovat tlačítka
- zobrazovat na displeji

**Chceme, aby to probíhalo současně, deterministicky a včas? → RT systém .**

# 1. Real-time systémy

## Charakteristiky real-time systémů

- Řízení událostí (reaktivní) vs. řízení podle času
- Požadavky na spolehlivost/odolnost proti poruchám (např. modulární redundance)
- Předvídatelnost
- Priority ve víceúlohových systémech

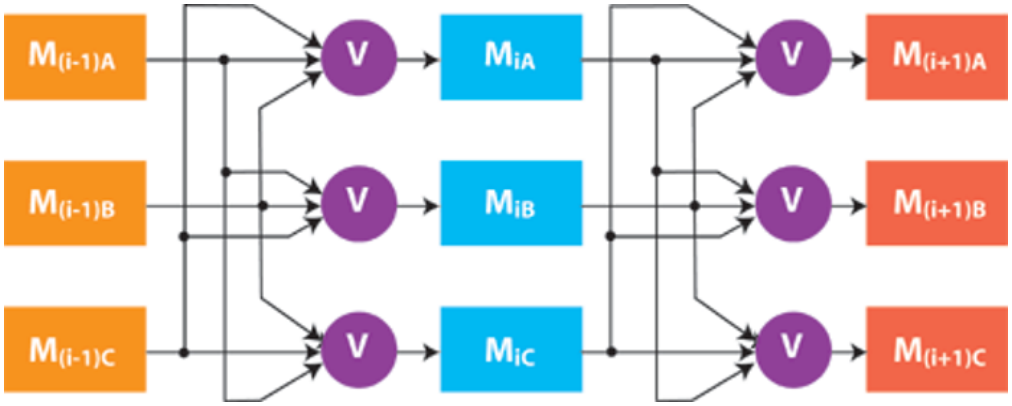


Figure 1: Příklad systému s modulární redundancí

## Klasifikace RT systémů

- **Hard RT**: reakce na vstupy musí přijít v požadovaném termínu - systémy řízení letů.
- **Soft RT**: termíny jsou důležité, ale systém bude správně fungovat i v případě, že budou termíny občas nedodrženy. Např. systém sběru dat.
- **Firm RT**: několik zmeškaných termínů nepovede selhání, ale více než několik zmeškaných termínů může vést k úplnému nebo katastrofickému selhání systému.

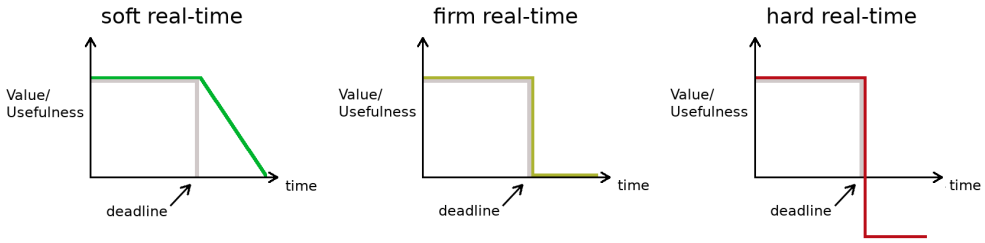


Figure 2: Klasifikace RT systémů

## Klasifikace RT systémů

### Statické

- Lze předvídat časy úloh

- Možnost statické analýzy (v době kompilace)
- Umožňuje dobré využití zdrojů (nízká doba nečinnosti procesorů)

**Dynamické**

- Nepředvídatelné časy vzniku úloh (asynchronní události)
- Statická analýza (v době kompilace) je možná pouze pro jednoduché případy
- Využití procesoru se dramaticky mění - návrh tak, aby zvládl “nejhorší případ”.
- Je třeba se vyvarovat příliš zjednodušujících předpokladů, např. předpokladu, že všechny úlohy jsou nezávislé, i když je to nepravděpodobné.

## Klasifikace RT systémů

**Periodické**

- Každá úloha (nebo skupina úloh) se provádí opakovaně s určitou periodou.
- Umožňuje použití některých technik statické analýzy
- Odpovídá charakteristikám mnoha skutečných problémů
- Je možné mít úlohy s termíny menšími, rovnými nebo většími, než je jejich perioda - pozdější jsou obtížně zpracovatelné, vyskytuje se více souběžných instancí úloh

**Neperiodické (sporadické, asynchronní nebo reaktivní)**

- Vytváří dynamickou situaci
- Časově omezené intervaly příchodu jsou snadněji zvládnutelné
- Systémy s omezenými zdroji nezvládnou neomezené časové intervaly příchodu

## Jak řešit problémy spojené s reálným časem?

**Správa systémových zdrojů**

*(procesor, paměť, vstupně/výstupní zařízení, atd.):*

- **Sleduje** stav a **vlastníka** každého zdroje.
- **Rozhoduje**, kdo získá přístup ke zdroji.
- **Určuje**, jak dlouho může být zdroj používán.

**V systémech podporujících současné provádění programů/úloh**

- **Řeší konflikty** o zdroje.
- **Optimalizuje výkon** při mnoha rozdílných úlohách.

# 2. Operační systémy reálného času

## RTOS

- **Často**: RTOS = jádro operačního systému.
- RTOS není **lepší OS**. Je to nástroj pro řízení času.
- **Vestavěné systémy**
  - jsou navrženy pro jeden konkrétní účel,
  - funkce jako uživatelské rozhraní nebo přístup k souborům/diskům nejsou potřeba.
- **RTOS poskytuje kontrolu nad zdroji**:
  - Žádné procesy/úlohy na pozadí, které „se prostě dějí“.
  - Omezený počet úloh.
- **RTOS umožňuje kontrolu nad načasováním** díky:
  - Možnosti manipulace s prioritami úloh.
  - Výběru z možností plánování.

## Co dělá RTOS na embedded platformách?

Dělá pár věcí, ale dělá je extrémně spolehlivě:

1. Multitasking – více vláken na jednom jádru (časově multiplexované).
2. Scheduling – rozhoduje, co se má spustit dál.
3. Synchronizace – mutexy, semafory, event groups.
4. Časování – delay, timeouty, periodicita.
5. Paměť – zásobníky úloh, alokace TCB (time control block), queue buňky.
6. ISR integrace – bezpečná komunikace s přerušeními.

A to celé na platformě, kde:

- stacky jsou malé, ISR jsou tvrdé real-time požadavky a priority jsou absolutní.

## Úlohy a funkce

Úloha je proces, který se opakuje

- nekonečná smyčka
- základní funkční blok RTOS

Funkce je procedura, kterou voláme

- synchronně (deterministicky) z programu nebo asynchronně jako obsluhu události
- běží, má vymezené místo v paměti, může vracet data

```
1 void process_data();
2
3 int add_two_numbers(int a, int b);
```

## Jak na MCU běžně plánujeme úlohy (bez RTOS)?

```c
while (1)
{
    if (task1_timer == 0)
    // pokud hodnota task1_timer ještě není 0, je dekrementována
    // každou 1ms během obsluhy přerušení časovače
    {
        task1_timer = t1;
        task1();            // task1 trvá m1 ms
    }
    if (task2_timer == 0)
    // pokud hodnota task1_timer ještě není 0, je dekrementována
    // každou 1ms během obsluhy přerušení časovače
    {
        task2_timer = t2;
        task2();            // task2 trvá m2 ms
    }
}
```

## Časové parametry úloh

Každá periodická úloha (task) je typicky definována čtveřicí parametrů:

`(WCET, T, D, φ)`

### WCET — Worst-Case Execution Time

- Nejhorší (maximální) doba vykonávání úlohy [ms/µs].

Využití

- analýze schedulovatelnosti (Liu–Layland),
- výpočtu odezvy,
- návrhu priorit,
- detekci kolizí s jinými úlohami.

### Perioda — T

Doba mezi dvěma aktivacemi úlohy.

Běžné zdroje periody:

- systémový časovač (SysTick), periodické přerušení (např. `TIM2`).
- aplikace – snímání senzorů, řízení PWM,

### Deadline - D

Maximální přípustný čas, do kdy musí být úloha dokončena.

Tj. čas mezi aktivací a okamžikem, kdy výsledek už není použitelný.

- implicitní deadline: `D = T`
- konkrétní deadline: `D < T` nebo `D > T`.

**Offset — φ (phi)**

Počáteční zpoždění před první aktivací úlohy.

Používá se u úloh, které nesmí startovat současně → sníží interference.

Příklad:

- Task1: čtení IMU → offset 0 ms
- Task2: logování na SD kartu → offset 7 ms (aby se nepotkali v jedné ms slicu)

Použití:

- redukce špičkového zatížení CPU,
- eliminace kolizí na sběrnici (I2C/SPI),
- stabilizace systému s více periodickými úlohami.

## Jak na MCU plánujeme úlohy?

Nekonečný cyklus (super loop):

- probíhá sekvenčně
- blokuje se na délku trvání úloh `m1`, `m2`
- `task1_timer` i `task2_timer` se odpočítávají jen v ISR, takže
  - běh úloh záleží jen na momentě, kdy se cyklus dostane k testu `if`
  - pokud se úloha nevejde do svého `časového okna`, dojde ke skluzu v provádění

Pojďme provést časovou analýzu, tj. jak často se jednotlivé úlohy (funkce) volají.

## 1. scénář

Task1: `t1=5`, `m1=1` Task2: `t2=10` a `m2=15`

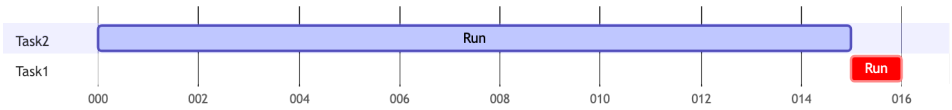


Figure 3: Ilustrace 1. scénáře

**Reálné chování**

- `task1` poběží cca jednou za ~16 ms (protože se neustále čeká 15 ms v `task2`)
- `task2` poběží stále dokola s periodou 15 ms (podle toho, jak často cyklus proběhne)

## 2. scénář

Task1: `t1=20`, `m1=1` Task2: `t2=10` a `m2=15`

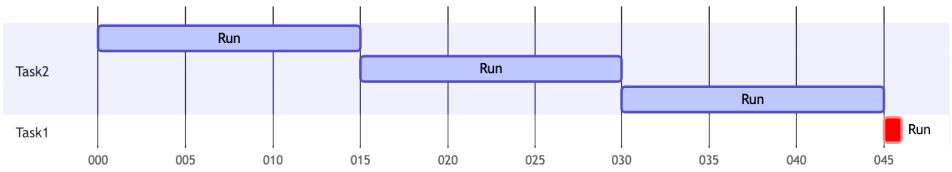


Figure 4: Ilustrace 2. scénáře

**Reálné chování**

- `task2` běží s cca 15 ms opakováním (nestíhá svou 10ms periodu)
- `task1` se zavolá jen tehdy, když náhodou trefí okno mezi běhy `task2`, ale takové okno není → neběží nikdy (nebo ultra sporadicky)

## 3. scénář

Task1: `t1=20`, `m1=1` Task2: `t2=25` a `m2=15`

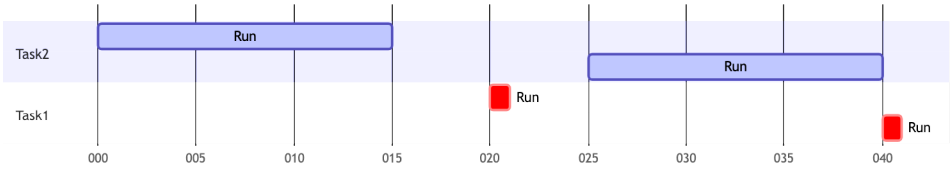


Figure 5: Ilustrace 3. scénáře

**Reálné chování**

- `task1` běží vždy přesně v periodě.
- `task2` také, nic ho neruší - idle zóna mezi 15–20 a 40–45 = CPU rezerva.

## 4. scénář

Task1: `t1=4`, `m1=1` Task2: `t2=8` a `m2=4`

**Reálné chování**

- `task1` každých cca 5 ms (místo 4 ms)
- `task2` každých cca 10 ms (podle fázování)

## Teorie plánovatelnosti: Liu & Layland

Mez použitelnosti $n$ nezávislých periodických úloh Pro $n$ úloh s periodami $T_i$ a časy běhu $C_i$ je limit $U$


Figure 6: Ilustrace 4. scénáře

$$U = \sum_{i=0}^{n} \left( \frac{C_i}{T_i} \right)$$

$$U \leq n \left( 2^{\frac{1}{n}} - 1 \right)$$

Pro 2 úlohy ($n = 2$) → $U = 0.8284$

- pokud je $U$ větší → úlohy nejsou garantovatelně plánovatelné
- ani v ideálním preemptivním systému.

## Analýza příkladu

### 1. scénář

`U = 1/5 + 15/10 = 0.2 + 1.5 = 1.7 ≥ 0.82`

X Neplánovatelné ani teoreticky

### 2. scénář

`U = 1/20 + 15/10 = 0.05 + 1.5 = 1.55 ≥ 0.82`

X Neplánovatelné

### 3. scénář

`U = 1/20 + 15/25 = 0.05 + 0.6 = 0.65 <= 0.82`

OK Teoreticky plánovatelné v preemptivním scheduleru

### 4. scénář

`U = 1/4 + 4/8 = 0.25 + 0.5 = 0.75 <= 0.82`

OK Teoreticky realizovatelné

# 3. Architektura RTOS

## Architektonické přístupy návrhu real-time systému

- Systémy s cyklickým dotazováním (pooling)
- Systémy řízené přerušeními
- Systémy foreground/background
- Multitasking

## Cyklické dotazování

- **Nejjednodušší RT jádro**.
- Jediná a opakující se instrukce testuje flag - došlo k události nebo ne?
- **Příklady**:
  - Neblokující LCD instrukce,
  - Neblokující "get string" přes UART kanál.
- Není potřeba žádná komunikace mezi úlohami ani plánování.
  - Existuje pouze jedna úloha.
- **Vynikající pro zpracování vysokorychlostních datových kanálů**, zejména když:
  - Události nastávají v širokých intervalech.
  - Procesor je věnován pouze zpracování datového kanálu.

## Cyklické dotazování

## Výhody:

- **Jednoduché na napsání a ladění**.
- **Čas reakce je snadno určen** (ve srovnání s programováním založeným na úlohách, kde jsou dvě úlohy místo jedné).

## Nevýhody:

- Může selhat kvůli nárazům událostí.
- Obecně není dostatečné pro zpracování složitých systémů.
- **Zbytečné využívání CPU** času, zejména když se dotazovaná událost vyskytuje zřídka.

## Systémy řízené přerušením

Hlavní smyčka nedělá skoro nic, těžká práce v ISR

#### Výhody:

- Velmi nízká latence — reakce *hned*.
- Elegantní u zdrojů, které se hlásí samy (UART RX, ADC end-of-conversion).
- Skvělé pro měkký reálný čas.

**Nevýhody:**

- ISR nesmí být dlouhé → jinak blokují ostatní přerušení.
- Priority přerušení mohou způsobit inverze a timing chaos.
- Nelze snadno řídit složité interakce mezi úlohami.

## Systémy foreground/background

- **Nejčastější hybridní řešení pro vestavěné aplikace**.
- Zahrnují **přerušením řízené procesy** (přední) a **hlavní proces** (pozadí).
- Všechna řešení reálného času jsou pouze speciálním případem systémů přední/pozadí:
  - **Cyklické dotazování** = systém pouze pozadí.
  - **Systémy pouze s přerušeními** = systém pouze přední.
- Vše, co není časově kritické, by mělo být v **pozadí**.
  - Pozadí je proces s **nejnižší prioritou**.

**Výhody:**

- Jednoduché, stabilní, velmi běžné.
- ISR je jen **trigger** → žádná logika
- Background je **deterministický state machine**

**Nevýhody:**

- Stále to není skutečný multitasking.
- ISR část musí být krátká, jinak se zvyšuje latence.
- Komplexita roste rychle s počtem událostí.

**Typické použití:**

- Většina jednoduchých firmware v automotive, průmysl, malé řídicí jednotky.

## Multitasking

- **Oddělené úlohy**, které sdílejí jeden procesor (nebo více procesorů).
- Každá úloha běží ve svém vlastním **kontextu**:
  - Má **vlastní procesor**
  - Vidí **své vlastní proměnné**
  - Může být **přerušena**
- Úlohy mohou vzájemně interagovat, aby vykonaly celý program.

**Jak sdílí mnoho úloh stejný CPU?**

- Cyklické exekutivní systémy.
- Systémy s rotujícím výběrem (např. Round Robin).
- Předem přerušitelné prioritní systémy.

**Kooperativní multitasking**

Úloha musí dobrovolně odevzdat CPU (vhodné pro low-power, deterministické systémy).

```
void taskA() {
    do_work();
    yield();
}
```

**Preemptivní multitasking**

Úloha může být přerušen časovačem → umožňuje pevné priority.

```
void SysTick_Handler() {
    scheduler_tick();
}
```

## Preemptivní plánování podle priorit

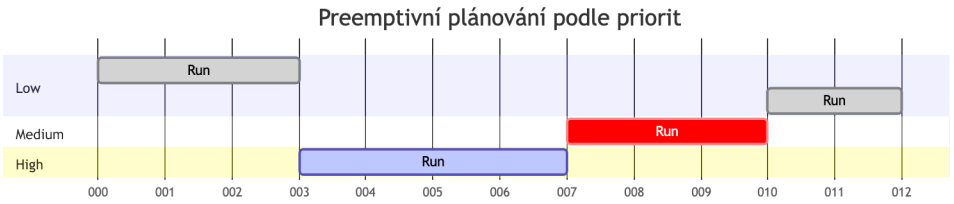


Figure 7: Ilustrace preemtivního plánovače

**Chování (preemptivní podle priorit):**

- **0-3ms**: Low běží (žádná jiná úloha)
- **3ms**: High přichází → okamžitě preemptuje Low
- **3-7ms**: High běží (nejvyšší priorita), Low čeká
- **7ms**: Medium přichází → preemptuje Low
- **7-10ms**: Medium běží, Low stále čeká
- **10-12ms**: Low konečně dokončuje

## Plánovací algotitmus 1 - Round Robin

- Každá úloha se stejnou prioritou dostane časový kvantum
  - time slice.
- Po vypršení → další úloha stejné priority.
- Výhody: férový, jednoduchý, nízké nároky.
- Nevýhody: nehodí se pro tvrdý real-time — jitter.

https://cs.wikipedia.org/wiki/Round-robin_scheduling

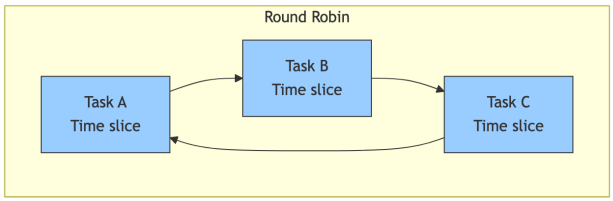


Figure 8: Algoritmus výběru další úlohy

## Round Robin - Timeline

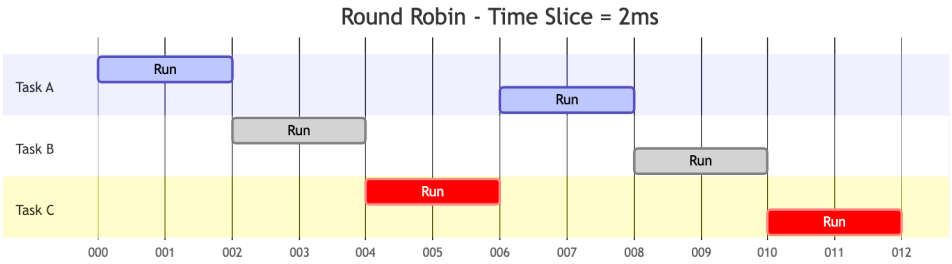


Figure 9: Ilustrace plánovacího algoritmu Round Robin

**Příklad:** 3 úlohy se stejnou prioritou, time slice = 2ms. Každá úloha běží 2ms, pak předá řízení další.

## Plánovací algoritmus 2 - Earliest Deadline First

### Hardcore real-time

- Úloha s nejbližším deadline běží první.
- Optimální pro periodické úlohy podle Liu & Layland (pokud splní utilization testy).

### Výhody

- Vysoká procesorová využitelnost (až 100% v ideálních podmínkách).
- Elegantní matematika, ovšem složitější

### Nevýhody

- Je třeba třídit připravené úlohy podle deadline → většinou heap/priority queue.

## EDF - Timeline

**Příklad:** T1 má deadliny v čase 5, 10, 15ms. T2 má deadline v 8ms. T3 má deadline ve 12ms. Plánovač vždy vybírá úlohu s nejbližším deadlinem.

## Plánovací algoritmus 3 - Rate Monotonic

- Klasika pro pevně periodické úlohy.

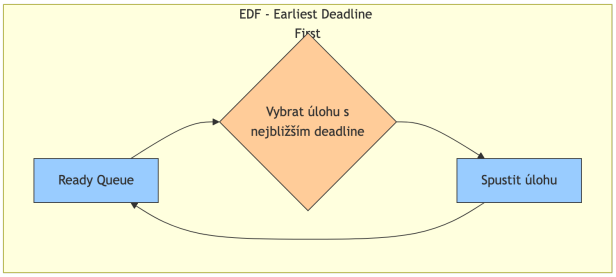


Figure 10: Algoritmus výběru další úlohy

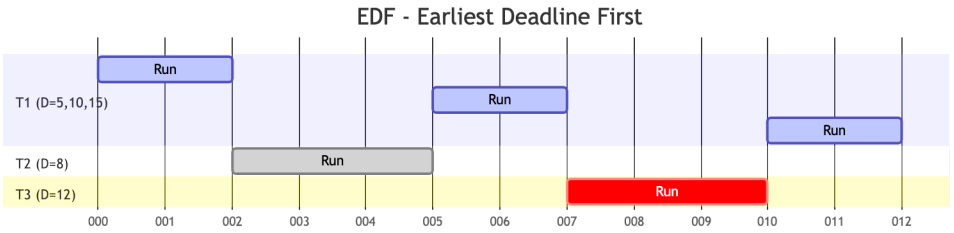


Figure 11: Ilustrace plánovacího algoritmu Earliest Deadline First

- Čím kratší perioda → tím vyšší priorita.
- Statické, žádná změna priorit za běhu.

https://cs.wikipedia.org/wiki/Rate_monotonic_scheduling

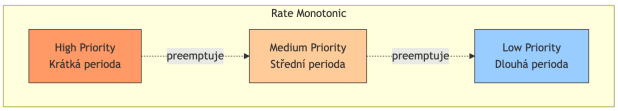


Figure 12: Algoritmus výběru další úlohy

## Rate Monotonic - Timeline

**Příklad:** T1 má periodu 4ms (vyšší priorita), T2 má periodu 8ms (nižší priorita). T1 běží vždy první, T2 pouze když T1 nečeká na další periodu.

## Časování plánovače

### Ticked scheduler

- pravidelný SysTick (např. 1 kHz),
- jednoduchý,
- má větší spotřebu → nevhodné pro low-power (MCU se probouzí 1000×/s),
- výhody:
  - lehká implementace `delay()`, snadná aktualizuce timeoutů

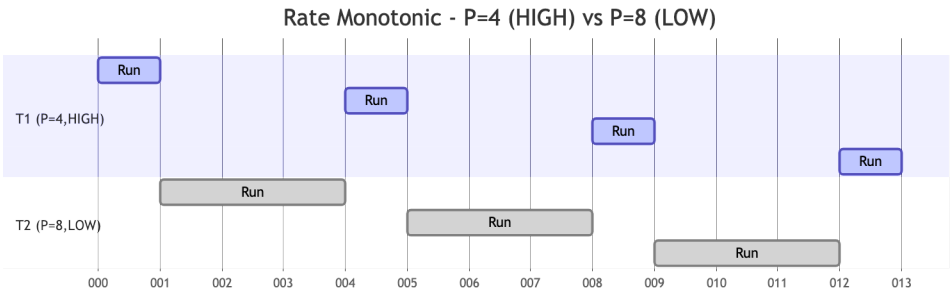


Figure 13: Ilustrace plánovacího algoritmu Rate Monotonic

**Tickless scheduler**

- SysTick běží jen když je potřeba,
- RTOS vypočítá nejbližší timeout → nastaví wake-up timer → MCU může spát.
- složitější implementace (počítání „spánkové mezery“),
- v super low-power IoT jednoznačně výhra.

## Životní cyklus úlohy

**běžící (RUNNING)**

- využívá procesor
- přerušením ji lze přesunout do stavu připravena

**připravena (READY / sleep)**

- čeká na CPU / plánovač
- plánovač přesouvá do stavu běžící

**blokována (BLOCKED)**

- běžící úloha se dotáže na data z periferie
- po získání dat se přesouvá do stavu připravena

## Příklad plánování ve víceúlohovém systému

**Máme dvě úlohy:**

- Úloha A (HIGH priority) – kritická práce, třeba obsluha senzoru.
- Úloha B (LOW priority) – méně důležitý úkol, např. blikání LED.

Oba běží nekonečně, oba chtějí vstoupit do CPU, ale scheduler jim přidělí čas úplně jinak.

**Co se stane po startu systému**

1. Scheduler najde nejvyšší READY prioritu.
2. To je Úloha A → dostane CPU.
3. Úloha B vůbec nepřijde na řadu, dokud jí Úloha A nedá prostor (yield, delay, bloknutí, waiting).

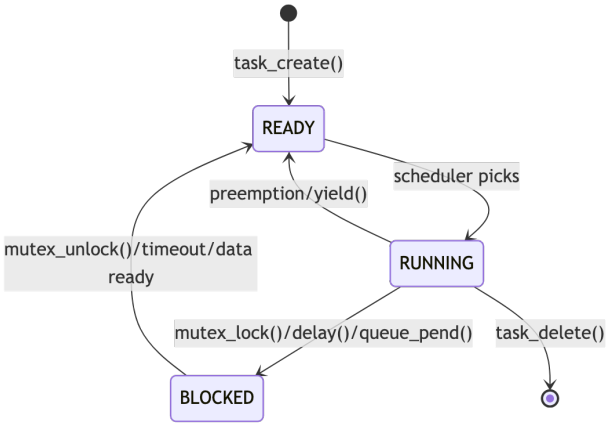


Figure 14: Životní cyklus úlohy

**V preemptivním RTOSu má priorita absolutní váhu.**

## Typické chování bez blokování

Pokud Úloha A jen točí while(1):

```
while(1) {
    // nějaká “důležitá” práce
}
```

→ Úloha B nikdy neběží. Ani na mikrosekundu.

Není žádný time slicing napříč prioritami

V RTOSu neplatí, že *každá úloha dostane trochu času*. RTOS není Linux.

## Co musí Úloha A udělat, aby Úloha B dostala CPU

1. Zavolá `yield()`
   - dobrovolně pustí CPU
   - scheduler vybere nejvyšší READY prioritu
   - to je opět A (pokud je READY)
   - takže to B moc nepomůže, ale v některých systémech to může umožnit krátké proběhnutí B.
2. Zavolá `delay()`
   - A se označí jako BLOCKED.
   - Timeout bude za 10 ms.
   - Scheduler vybere další připravenou úlohu → B.
3. A zablokuje mutex / queue
   - Pokud čeká na něco:
     - fronta prázdná,
     - mutex držen jinou úlohou,
     - semafor = 0,

→ Úloha A jde do stavu BLOCKED, B dostane CPU.

## Přepínání kontextu (Context Switching)

- Při přepnutí úloh dochází k tzv. **přepnutí kontextu**.
- **Uloží se minimální množství informací potřebných k obnovení přerušeného procesu**:
  - obsah registrů,
  - obsah programového čítače,
  - registry paměťových stránek,
  - paměťově mapovaný I/O,
  - speciální proměnné.
- Během přepínání kontextu jsou často **zakázána přerušení**.
- **Systémy reálného času** vyžadují **minimální čas pro přepínání kontextu**.

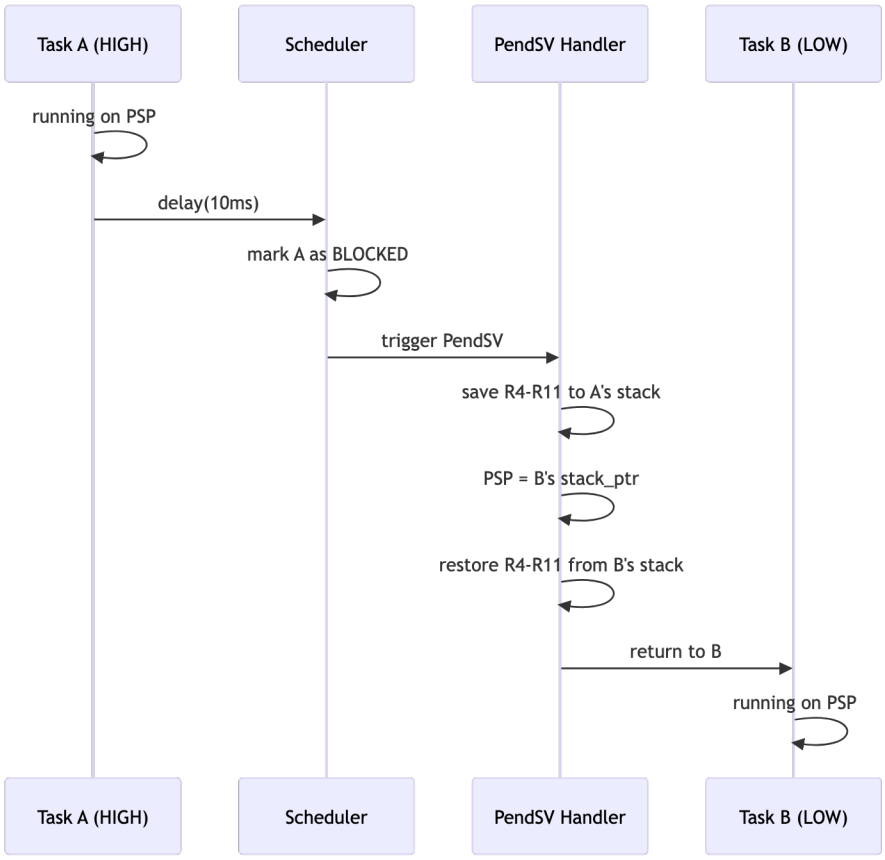


Figure 15: Přepínání kontextu na procesorech s jádrem M4

**Výhody:**

- Paralelní (logické) běhy úloh.
- Úkoly mají vlastní stack → čistší architektura.

- Preemptivní systém umožňuje pevné priority → základ reálného času.
- Možnost využití protokolů (mutex, semafor, fronty).

**Nevýhody:**

- Kontextové přepínání něco stojí (čas i stack).
- Více možností vytvořit deadlock nebo priority inversion.
- Programátor musí řešit synchronizaci.

**Typické příklady:**

- FreeRTOS, RTX, Zephyr, ThreadX.

## Synchronizační mechanismy

**Mutex**

- Exkluzivní přístup,
- priority inheritance,
- má vlastníka → mutex musí odemykat stejná úloha.

**Binární semafor**

- 0/1 token,
- vhodné pro signalizaci ISR → úloha.

**Počítací (counting) semafor**

- Fronta signálů,
- vhodné pro ISR, pipeline, přenos dat.

**Fronta (Queue)**

- Nejflexibilnější objekt v RTOSu,
- umožňuje posílat kompletní struktury mezi úlohami.

**Event groups**

- Umožní čekat na více událostí najednou → OR/AND logika
- Typické použití:
  - Protokol: mám data? mám clock? mám link?
  - Synchronizace více ISR do jedné úlohy

## Správa zdrojů - mutex

V embedded systémech je skoro vždycky nějaký sdílený zdroj — buffer na UART, globální proměnnou, hardware periférii, nebo třeba nějaký kus vyšší vrstvy logiky.

Ve víceúlohovém systému se mohou úlohy pokoušet přistupovat současně ke stejnému zdroji.

Současný přístup bez koordinace:

- jeden něco zapíše,
- druhý to přepíše v půlce,

- výsledkem je cosi mezi, což pak debuguješ tři hodiny a nadáváš na scheduler.

**Mutex je způsob, jak tomu zabránit.**

## Jak mutex funguje?

Mutex je token, který může držet vždy jen jedna úloha.

- Úloha, která přistupuje ke zdroji mutex zamkne.
- Po ukončení činnosti (konec kritické sekce) úloha mutex odemyká.

A RTOS dělá to, že:

- pokud mutex máš → úloha pokračuje,
- pokud je zamčený → RTOS úlohu zablokuje a přepne na jinou.

### Uložení a blokování úlohy

Úloha, která se snaží zamknout obsazený mutex:

- přepne se do stavu BLOCKED,
- scheduler ji vyřadí z fronty čekajících (tj. plánovatelných) úloh
- uloží ji do waiting listu mutexu,
- vyvolá se přepnutí na jinou úlohu.

Důležitá věc: uložení do fronty mutexu se musí stát atomicky.

- řeší se krátkým `__disable_irq()`, aby se zabránilo jinému ISR

## Životní cyklus mutexu

Typický životní cyklus:

1. Úloha A: chce mutex → dostane ho → owner = A.
2. Úloha B: chce mutex → je blokován → jde do waiting listu.
3. Úloha A: skončí práci → mutex_unlock().
4. RTOS: probudí Úlohu B → owner = B.
5. Úloha B pokračuje tam, kde přestal.

Mutex by měl mít tyto základní vlastnosti:

- mutex může odemknout jen úloha, která ho vlastní,
- uvolnění mutexu probíhá v kritické sekci,
- blokovací operace nesmí běžet v ISR (ISR nesmí čekat),
- waiting list musí být konzistentní,
- RTOS by měl umožnit i `try_lock` (neblokující).

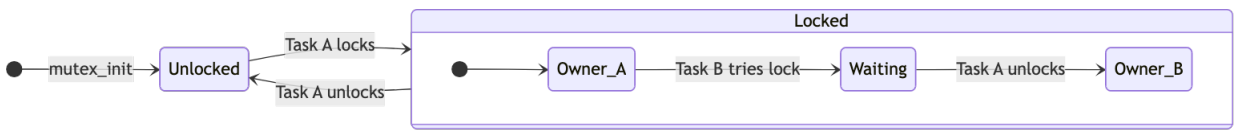


Figure 16: Životní cyklus mutexu

## Priority Inversion - problém mutexů

- Úloha s **vysokou prioritou** čeká na mutex držený úlohou s **nízkou prioritou**.
- Mezitím úloha se **střední prioritou** běží a blokuje nízkou prioritu.
- **Důsledek:** Vysoká priorita čeká déle než by měla, protože střední priorita běží místo nízké.

**Řešení: Priority Inheritance Protocol**

- Úloha s nízkou prioritou, která drží mutex, **dočasně zdědí prioritu** čekající úlohy s vysokou prioritou
- Brání tomu, aby úloha se střední prioritou blokovala vysokou prioritu
- Po uvolnění mutexu se priorita vrátí na původní hodnotu

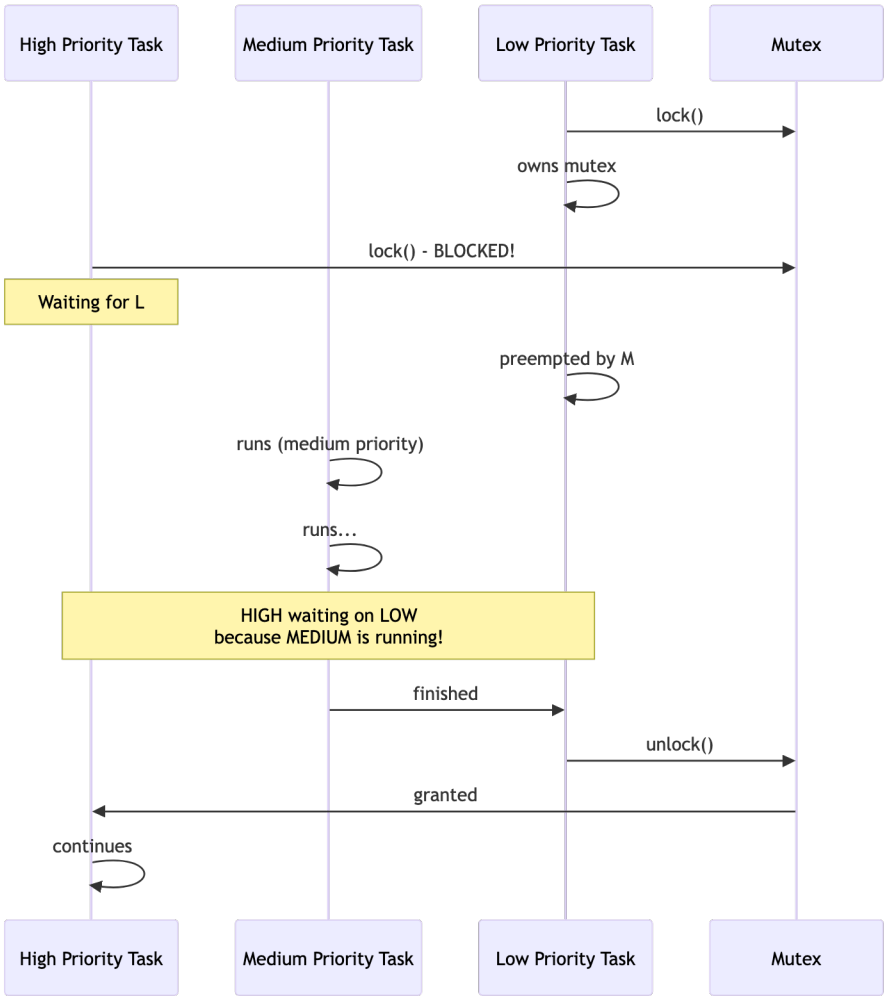


Figure 17: Ilustrace inverze priority

## Komunikace mezi úlohami

**Úlohy nepracují izolovaně**. Často je potřeba sdílet data nebo je upravovat v sérii. Protože **pouze jedna úloha může běžet v daném okamžiku**, musí existovat mechanismy pro komunikaci mezi úlohami.

**Příklady:**

1. Úloha čte cyklicky data ze senzoru. Data uložía poté musí signalizovat zpracovatelské úloze, aby data převzala a zpracovala, aby měla místo pro zápis dalších dat.
2. Úloha určuje stav systému (např. Normální režim, Urgentní režim, Spící, Zakázáno). Musí informovat všechny ostatní úlohy v systému o změně stavu.
3. Uživatel komunikuje s jiným uživatelem přes síť. Úloha pro příjem zpráv z této sítě musí doručit zprávy do terminálového programu, a terminálový program musí doručit zprávy do úlohy pro vysílání přes síť.

## Komunikace mezi úlohami v běžných OS

Běžné operační systémy mají mnoho možností pro předávání zpráv mezi procesy, ale většina z nich má významnou režii a není deterministická:

- **Pipes** (trubky): Jsou to spojení mezi dvěma procesy, kde standardní výstup jednoho procesu se stává standardním vstupem jiného procesu. Systém dočasně uchovává pipované informace, dokud nejsou přečteny příjemcem.
- **Fronty zpráv**: Asynchronní komunikační protokol, což znamená, že odesílatel a příjemce zprávy nemusí komunikovat se zprávovou frontou ve stejný čas. Zprávy umístěné ve frontě jsou uloženy, dokud je příjemce nevyzvedne.
- **Semafory**: Jsou proměnné nebo abstraktní datové typy, které se používají k řízení přístupu k společným prostředkům více procesy v konkurenčním systému, jako je multiprogramovací operační systém.

## Komunikace mezi úlohami v běžných OS

- **Vzdálené volání procedur (RPC)**: Protokol, který může jeden program použít k požádání o službu z programu umístěného v jiném počítači v síti, aniž by musel rozumět podrobnostem sítě.
- **Sokety**: Jsou jedním z bodů dvoucestné komunikační linky mezi dvěma programy běžícími v síti. Socket je vázán na číslo portu, aby TCP vrstva mohla identifikovat aplikaci, kam mají být data odeslána. Endpoint je kombinace IP adresy a čísla portu.
- **Datagramy**: Jsou samostatné, nezávislé jednotky dat, které nesou dostatečné informace k tomu, aby byly směrovány ze zdroje na cílový počítač, aniž by bylo nutné spoléhat se na předchozí výměny mezi těmito počítači a přenášenou sítí.

## Komunikace mezi úlohami v RTOS

- **Úlohy obvykle mají přímý přístup ke společnému paměťovému prostoru**, a nejrychlejší způsob sdílení dat je prostřednictvím sdílené paměti.
- V běžných operačních systémech je úlohám obvykle zabráněn přístup k paměti jiných úloh, což má dobrý důvod.

**Sdílená paměť**

- Globální proměnné použité jako flagy
- FIFO / buffer
  - **post()** - zápisová operace, která umístí data do bufferu
  - **pend()** - čtecí operace, která data z bufferu

## Funkce FIFO pro komunikaci mezi úlohami

- Pokud **žádná data nejsou dostupná**, úloha čekající na **pend()** je **pozastavena**.
- **Vzájemná exkluze**: Pokud někdo právě provádí **post()**, úloha čekající na **pend()** musí počkat.
- **Žádný procesorový čas není zbytečně ztracen** na polling bufferu, aby se zjistilo, jestli už jsou nějaká data dostupná.
- **Pend() může mít časový limit**, pro případ, že by žádná data nepřišla.
- Pokud máte **producenta** a **konzumenta**, kteří pracují různými rychlostmi, **buffer** může zajistit hladký běh systému.
  - Dokud **buffer není plný**, producent může zapisovat.
  - Dokud **buffer není prázdný**, konzument může číst.

# 4. Návrh vlastního RTOS pro SMTM32F4

Cíl: jednoduché, robustní preemptivní RTOS pro Cortex-M4 (STM32F4).

Minimální funkčnost:

- tvorba úloh,
- preemptivní plánovač s prioritami,
- časovač (tick),
- synchronizační primitiva (mutex/semafor),
- fronty zpráv, základní správa paměti (pooly).

## Požadavky a rozhodnutí na vysoké úrovni

1. Preemptivní plánovač: založen na prioritách (víceúrovňový), s volitelným time-slice (round-robin) mezi stejnými prioritami.
2. Jádro: kooperativní části jen jako volitelná funkce, hlavní mechanika — preempce.
3. Syscall model: volání z aplikace přes API (C). Použít SVC pro ochranu přístupu k jádru (volitelné). Přepínání kontextu přes PendSV.
4. Tick: buď pravidelný SysTick (1 ms) nebo tickless režim pro nízkou spotřebu.
5. Paměť: jednoduché fixed-size block pooly + heap (volitelné).
6. Footprint: minimalizovat — jádro v C s malými kritickými asm částmi (context switch).

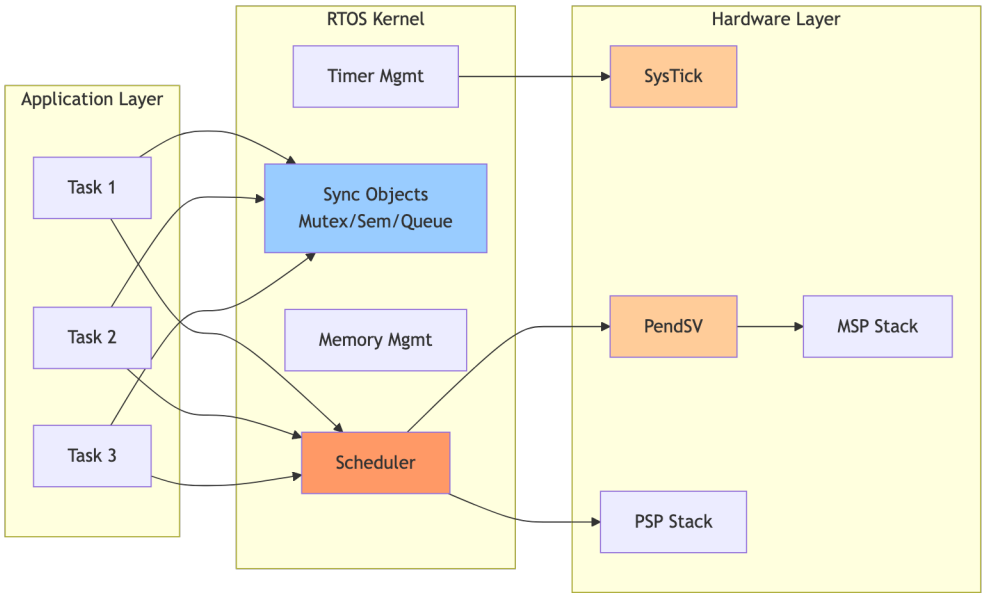


Figure 18: Architektura RTOS

## MSP – Main Stack Pointer

- Hlavní zásobník pro systémový kód.
- CPU po resetu vždy startuje s MSP.

- ISR běží na MSP.

Proč ho RTOS potřebuje:

- RTOS ho používá výhradně pro obsluhu přerušení a výjimek.
- Díky tomu je kontext ISR oddělen od kontextu jednotlivých vláken.

Konkrétní použití:

- `PendSV` handler pushuje/pulluje registry na MSP (přepínání kontextu).
- `SysTick` handler používá MSP.
- Úlohy nikdy neběží na MSP.

## PSP – Process Stack Pointer

- Oddělený zásobník pro vlákna (úlohy).

Proč ho RTOS potřebuje:

- Umožňuje, aby každý úloha měla vlastní stack.
- CPU přepíná mezi MSP → PSP automaticky podle bitu CONTROL.

Konkrétní použití:

- RTOS nastaví bit CONTROL.SPSEL = 1 → procesy běží na PSP.
- Při přepnutí kontextu RTOS uloží PSP staré úlohy do TCB.
- Obnovení PSP nového úlohy z TCB.

## SysTick – časovač pro scheduler

24bit downcounter, přednastavený na frekvenci 1ms (obvykle)

Proč ho RTOS potřebuje:

- Generuje tick přerušení → heartbeat plánovače.
- Slouží k odměřování:
  - časů blokování,
  - time slicing,
  - aktivování timeoutů.

Konkrétní použití (v SysTick_Handler):

- inkrement systémového času a dekrement časovačů úloh
- aktivace PendSV, pokud se některá úloha probudí → potřeba přepnout kontext

## PendSV – mechanismus přepínání kontextu

Vyhrazená výjimka s nejnižší prioritou.

Proč ho RTOS potřebuje:

- Odděluje časově náročné úkony (uložení/obnovení registrů) od rychlých ISR.

Konkrétní použití:

- Plánovač napíše do ICSR bit PENDSVSET.
- CPU spustí PendSV až když není žádné ISR s vyšší prioritou.
- V PendSV:

- uložit kontext (R4–R11 + PSP)
- vybrat novou úlohu z ready listu
- obnovit jeho PSP + registry

## Jak vypadá minimalistické RTOS jádro pro Cortex-M

### TCB (Task Control Block)

- pointer na stack,
- priorita,
- stav (READY/BLOCKED/RUNNING),
- ukazatele do ready listu.

### Scheduler

- ready fronty rozdělené podle priorit,
- O(1) výběr úlohy (bitmapa, index nejvyšší priority),
- volání z PendSV.

### Context switch

- spouští se přes PendSV_Handler,
- ukládá R4–R11 nebo obnovuje R4–R11,
- automatika Cortex-M uloží R0–R3, LR, PC, xPSR.

### SysTick (pokud je ticked scheduler)

- volá rtos_tick_handler(),
- decrement timeoutů,
- odblokování delay() úloh,
- penduje PendSV pokud došlo k unblocking vyšší priority.

### Synchronizační objekty

- mutex, semafory (binary/counting), queue.

## Task Control Block (TCB)

Struktura pro popis úlohy

```
typedef enum { TASK_READY, TASK_RUNNING, TASK_BLOCKED,
  TASK_SUSPENDED, TASK_TERMINATED } task_state_t;

typedef struct tcb {
    uint32_t *stack_ptr;        // aktuální PSP (stack pointer)
    uint32_t *stack_base;       // base stack (pro kontrolu overflow)
    uint32_t stack_size;
    uint8_t priority;           // 0 = highest
    task_state_t state;
    struct tcb *next;           // pro ready list
    // IPC: waiting-for details, timeout, event flags...
} tcb_t;
```

## READY List

Centrální datová struktura plánovače RTOS, která obsahuje všechny úlohy, které jsou schopné běžet, ale aktuálně neběží.

To znamená:

- **READY** = úloha může okamžitě běžet, není blokovaná, má přidělený stack, nic jí nebrání.
- **RUNNING** = právě běžící úloha, která je vybrána z READY listu.
- **BLOCKED** = čeká na objekt (mutex/semafor/delay).
- **SUSPENDED** = vypnutá, nemá být volána.

Takže READY list drží jen ty úlohy, které se účastní plánování.

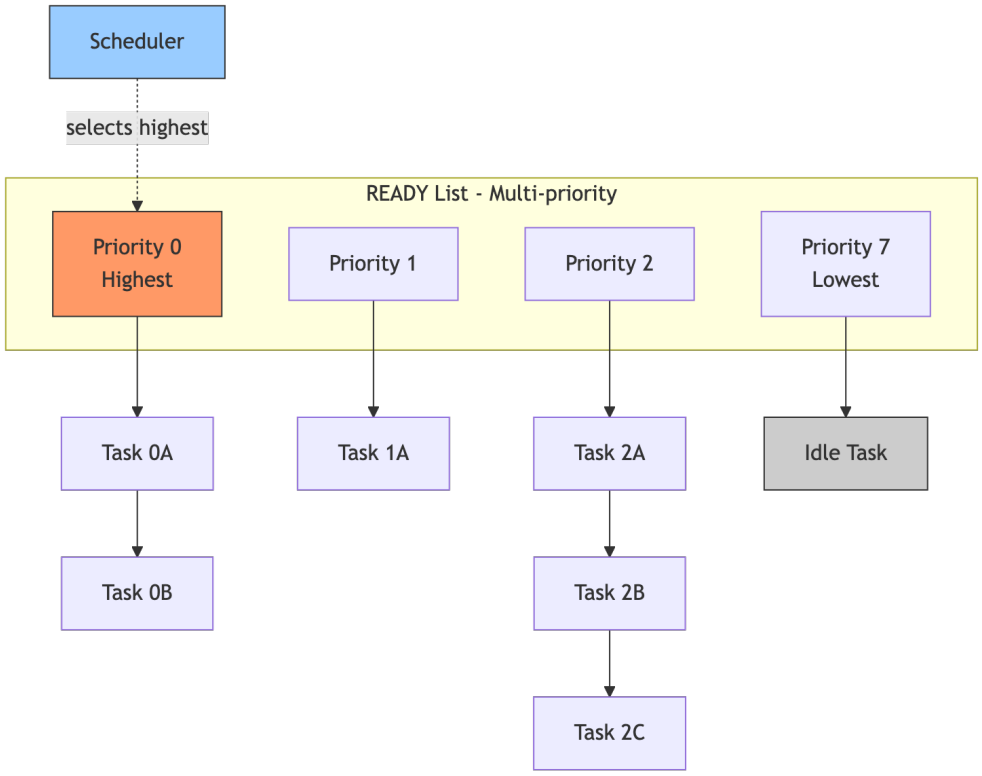


Figure 19: Architektura READY listu

## Proč je tento seznam klíčový?

RTOS neustále provádí:

- Příchod nové události → úloha přechází do READY
- Přerušení → úloha je odblokována → READY
- Preemptivní přepnutí → vybírá se úloha z READY
- Úloha zavolá delay / vezme mutex → je odstraněna z READY

Celý kernelový state machine úloh je postavený okolo READY listu.

## Varianta A — Jedna fronta (jednoduché RTOS)

`READY → [T3] → [T1] → [T2]`

Zpravidla FIFO.

**Nevýhody:**

- Scheduler musí skenovat frontu a hledat nevyšší prioritu.
- Preemptivní chování není efektivní.
- Používá se jen v učebnicových nebo zjednodušených RTOS.

## Varianta B — Více front podle priority

(FreeRTOS, ThreadX, RTX, Zephyr)

Každá priorita má vlastní READY frontu (kruhová FIFO).

```
prio 0 → [T0a] → [T0b]
prio 1 → [T1a]
prio 2 → [T2a] → [T2b] → [T2c]
```

Scheduler:

- Najde nejvyšší neprázdnou frontu
- Vezme první úlohu
- Tu dá do RUNNING

## READY list v naší implementaci

Každá položka je hlava jednosměrného seznamu.

```
#define MAX_PRIORITIES 8

tcb_t* ready_list[MAX_PRIORITIES];
```

Úloha se vkládá takto:

```
void ready_insert(tcb_t* t) {
    uint8_t p = t->priority;
    t->next = NULL;

    if (!ready_list[p]) {
        ready_list[p] = t;
    } else {
        tcb_t* it = ready_list[p];
        while (it->next) it = it->next;
        it->next = t;
    }
}
```

A scheduler to má velmi jednoduché:

```
tcb_t* scheduler_pick(void) {
    for (int p = 0; p < MAX_PRIORITIES; p++) {
        if (ready_list[p])
            return ready_list[p];
    }
    return idle_task;
```

```
}
```

## Plánovač

- Tick handler (SysTick) inkrementuje systémový čas, slučuje timeouty, a případně zahájí preemptivní přepnutí pokud je přípravena úloha s vyšší prioritou.
- `PendSV` : přepnutí kontextu (najdi příští úlohu, ulož aktuální, obnov příští).
- Vyvolání preemption:
  - v ISR nebo v ticku -> `SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk`
  - návrat z IRQ spustí `PendSV` .

### Přepnutí kontextu

- Cortex-M má dvě zásobníky: MSP (privileged) a PSP (process).
- PSP pro zásobník úloh a MSP pro kernel/ISR.

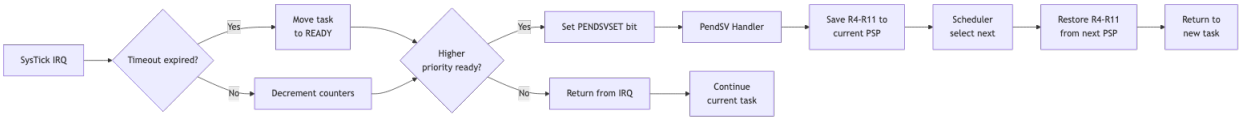


Figure 20: Popis funkcionality plánovače

## Minimální PendSV handler (ideově)

```
PendSV_Handler():
    disable_interrupts
    save r4-r11 on PSP
    current_tcb->stack_ptr = PSP
    next_tcb = scheduler_select_next()
    PSP = next_tcb->stack_ptr
    restore r4-r11 from PSP
    enable_interrupts
    return_from_exception  // EXC_RETURN
```

- Všechny operace na PSP jsou bezpečné, protože PendSV je nejnižší priorita.
- Vyhýbáme se přímému zápisu do registrů během jiných ISR.

## SysTick (tick timer)

```
volatile uint32_t sys_ticks = 0;

void SysTick_Handler(void) {

    sys_ticks++;
    // zpracuj timeouts: decrement wait counters,
    // uvolni blokované úlohy když timeout == 0
    // pokud existuje připravená úloha s vyšší prioritou
    // než current -> pend PendSV

    if (scheduler_should_preempt()) {
```

```
        SCB->ICSR = SCB_ICSR_PENDSVSET_Msk; // vyvolat PendSV
    }
}
```

## API - návrh funkcí

```
// task management
int rtos_task_create(void (*entry)(void*), void *arg, uint32_t *stack_mem,
  uint32_t stack_size, uint8_t priority);
void rtos_task_yield(void);
void rtos_task_delete(int task_id);
void rtos_task_sleep(uint32_t ms);
// synchronizace - mutex
void rtos_mutex_init(mutex_t *m);
void rtos_mutex_lock(mutex_t *m);
void rtos_mutex_unlock(mutex_t *m);
// synchronizace - semafor
void rtos_sem_init(sem_t *s, uint32_t initial);
int  rtos_sem_wait(sem_t *s, uint32_t timeout_ms);
void rtos_sem_post(sem_t *s);
// message queue
int rtos_queue_send(queue_t *q, void *msg, uint32_t timeout_ms);
int rtos_queue_receive(queue_t *q, void *out_msg, uint32_t timeout_ms);
```

## Ukázkový mini-scheduler (schematicky, C)

```
tcb_t *current_tcb;
tcb_t *ready_lists[MAX_PRIORITIES];
uint32_t ready_bitmap; // each bit = nonempty list

tcb_t* scheduler_select_next(void) {
    if (ready_bitmap == 0) return idle_tcb;

    // index nejvyššího bitu (depending on bit numbering)
    int highest = 31 - __builtin_clz(ready_bitmap);
    tcb_t *list = ready_lists[highest];

    // round robin: pop head, push tail
    tcb_t *next = list;
    ready_lists[highest] = next->next;
    next->next = NULL;
    if (!ready_lists[highest]) ready_bitmap &= ~(1u << highest);
    return next;
}
```

## Datová struktura mutexu

```
typedef struct mutex {
    tcb_t *owner;         // která úloha drží mutex
    uint8_t locked;       // 0 = volný, 1 = držen
    tcb_t *waiting_list;  // fronta čekajících úloh
} mutex_t;
```

- `owner`: identifikuje aktuálního držitele, důležité pro priority inheritance.

- `waiting_list`: fronta úloh čekajících na mutex, FIFO.

#### API mutexu

```
void mutex_init(mutex_t *m);
void mutex_lock(mutex_t *m);
void mutex_unlock(mutex_t *m);
```

## Kolik RTOS běží v reálných produktech?

- Apple AirPods → běží 2 různé RTOSy na jednom SoC.
- Tesla autopilot board → běží až 4 různé RTOS instance.
- Drony DJI → FreeRTOS + vlastní kontrolní RTOS pro flight controller.
- Různé hodinky, fitness trackery: často ne jeden, ale dva RTOSy (applikační + radio).

#### ThreadX (Azure RTOS)

- Extrémně rychlý scheduler.
- Deterministické API.
- Má *preemption threshold*, což je hodně unikátní → úloha může dočasně blokovat preempci úloh do určité priority.

#### Zephyr RTOS

- Moderní, modulární, podporuje SMP.
- Vhodný pro komplexní wearable / IoT produkty.
- Silná integrace se subsystémy (BLE stack, networking, FS…).

#### µC/OS-II / µC/OS-III

- Extrémně deterministické.
- Certifikovatelné (avionika, automotive).

#### TI-RTOS, RTX5, RIOT, ChibiOS, NuttX

- RTX5 → velmi čistá implementace, součást CMSIS.
- ChibiOS → ultra rychlé context switchování.
- NuttX → velmi *POSIX-like* (téměř jako mini-Linux).