Mikroprocesory

8. Další možnosti programování MCU. RTOS

Stanislav Vítek
Katedra radioelektroniky
České vysoké učení technické v Praze

V čem se dá ještě programovat

Micropython

- https://docs.micropython.org/en/latest/reference/index.html
- https://micropython.org/stm32/
- https://github.com/micropython/micropython/tree/master/ports/stm32

Micropython není jedinou implementací Pythonu pro mikrokontroléry

- CircuitPython, derivát Micropythonu, udržuje Adafruit, rozdíly
- MicroPython pro BBC micro:bit

Obecná kontrola MCU

Modul machine

- Abstraktní vrstva pro komunikaci s HW (shodná pro více kontrolérů)

```
import machine

machine.freq()          # get the current frequency of the CPU
machine.freq(240000000) # set the CPU frequency to 240 MHz
```

Moduly se specifickou funkcionalitou

- rp2, funkcionalita specifická pro RP2040

```
import rp2
```

Modul machine

- Modul obsahuje specifické funkce související s hardwarem na konkrétní desce.
- Většina funkcí modulu umožňuje přímý a neomezený přístup k hardwarovým blokům systému (jako je procesor, časovače, sběrnice atd.) a jejich ovládání.
- Při nesprávném použití může dojít k poruše, zablokování, v krajním případě i k poškození hardwaru.
- Na vhodném hardwaru nabízí MicroPython možnost psát obsluhy přerušení v jazyce Python. Obsluhy přerušení - známé také jako rutiny obsluhy přerušení (ISR) - jsou definovány jako callback funkce. Ty se provádějí v reakci na událost, jako je spuštění časovače nebo změna napětí na pinu.

Příklad přístup do paměti

- Příklad specifický pro platformu STM32

```
import machine
from micropython import const

GPIOA = const(0x48000000)
GPIO_BSRR = const(0x18)
GPIO_IDR = const(0x10)

# set PA2 high
machine.mem32[GPIOA + GPIO_BSRR] = 1 << 2

# read PA3
value = (machine.mem32[GPIOA + GPIO_IDR] >> 3) & 1
```

GPIO

- GPIO popisuje třída machine.Pin

```
from machine import Pin

p0 = Pin(0, Pin.OUT)    # create output pin on GPIO0
p0.on()                 # set pin to "on" (high) level
p0.off()                # set pin to "off" (low) level
p0.value(1)             # set pin to on/high

p2 = Pin(2, Pin.IN)     # create input pin on GPIO2
print(p2.value())       # get value, 0 or 1

p4 = Pin(4, Pin.IN, Pin.PULL_UP) # enable internal pull-up resistor
p5 = Pin(5, Pin.OUT, value=1) # set pin high on creation
```

GPIO s přerušením

```
import time
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(16, mode=Pin.OUT)


def button_isr(pin):
  pin_led.value(not pin_led.value())

pin_button.irq(trigger=Pin.IRQ_FALLING, handler=button_isr)

while True:
  ...
```

UART

```
from machine import Pin, UART
import time

uart = UART(1, baudrate=9600, tx=Pin(4), rx=Pin(5))
uart.init(bits=8, parity=None, stop=2)

led = Pin("LED", Pin.OUT)

while True:
    uart.write('t')
    if uart.any():
        data = uart.read()
        if data== b'm':
            led.toggle()
    time.sleep(1)
```

CircuitPython

```
import time
import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Javascript

- Tiny Javascript runtime Kaluma, github
- Javascript engine JerryScript

```
var led = 25;
pinMode(led, OUTPUT);

setInterval(() => {
  digitalToggle(led);
}, 1000);
```

Arduino

- Podpora platforem jazykem Wiring bývá založena na Arm Mbed

```
#define LED 25

void setup() {
  pinMode(LED, OUTPUT);
}

void loop() {
  digitalWrite(LED, HIGH);
  delay(1000);
  digitalWrite(LED, LOW);
  delay(1000);
}
```

Rust

- Kromě oficiální dokumentace Rust existuje řada tutoriálů a knih,
- Porovnání Rust vs. C
- Love: 10 Reasons Not To Use Rust (The Whole Truth)
- Hate: Stop writing Rust

```rust
#![no_std]
#![no_main]

use cortex_m_rt::entry;
use defmt::*;
use defmt_rtt as _;
use embedded_hal::digital::v2::OutputPin;
use embedded_time::fixed_point::FixedPoint;
use panic_probe as _;
use rp2040_hal as hal;

use hal::{
    clocks::{init_clocks_and_plls, Clock},
    pac,
    io::Sio,
    watchdog::Watchdog
};

#[link_section = ".boot2"]
#[used]
pub static BOOT2: [u8; 256] = rp2040_boot2::BOOT_LOADER_W25Q080;
```

```rust
#[entry]
fn main() -> ! {
    let mut pac = pac::Peripherals::take().unwrap();
    let core = pac::CorePeripherals::take().unwrap();
    let mut watchdog = Watchdog::new(pac.WATCHDOG);
    let sio = Sio::new(pac.SIO);

    let external_xtal_freq_hz = 12_000_000u32;
    let clocks = init_clocks_and_plls(
        external_xtal_freq_hz,
        pac.XOSC,
        pac.CLOCKS,
        pac.PLL_SYS,
        pac.PLL_USB,
        &mut pac.RESETS,
        &mut watchdog,
    )
    .ok()
    .unwrap();

    let mut delay = cortex_m::delay::Delay::new(core.SYST, clocks.system_clock.freq().integer());
```

```
    let pins = hal::gpio::Pins::new(
        pac.IO_BANK0,
        pac.PADS_BANK0,
        sio.gpio_bank0,
        &mut pac.RESETS,
    );

    let mut led_pin = pins.gpio25.into_push_pull_output();

    loop {
        led_pin.set_high().unwrap();
        delay.delay_ms(500);
        led_pin.set_low().unwrap();
        delay.delay_ms(500);
    }
}
```

Lua

```
gpio_pin = 25
pico.gpio_set_function (gpio_pin, GPIO_FUNC_SIO)
pico.gpio_set_dir (gpio_pin, GPIO_OUT)
while true do
  pico.gpio_put (gpio_pin, HIGH)
  pico.sleep_ms (300)
  pico.gpio_put (gpio_pin, LOW)
  pico.sleep_ms (300)
end
```

Go

- Kompilátor TinyGo, getting started, YouTube tutoriál

```
package main

import (
    "machine"
    "time"
)

func main() {
    led := machine.LED
    led.Configure(machine.PinConfig{Mode: machine.PinOutput})
    for {
        led.Low()
        time.Sleep(time.Millisecond * 500)

        led.High()
        time.Sleep(time.Millisecond * 500)
    }
}
```

RTOS

Klasifikace systémů založených na MCU

Vestavný systém

- vykonává omezený soubor specifických funkcí;
- často komunikuje se svým okolím.

RT systém

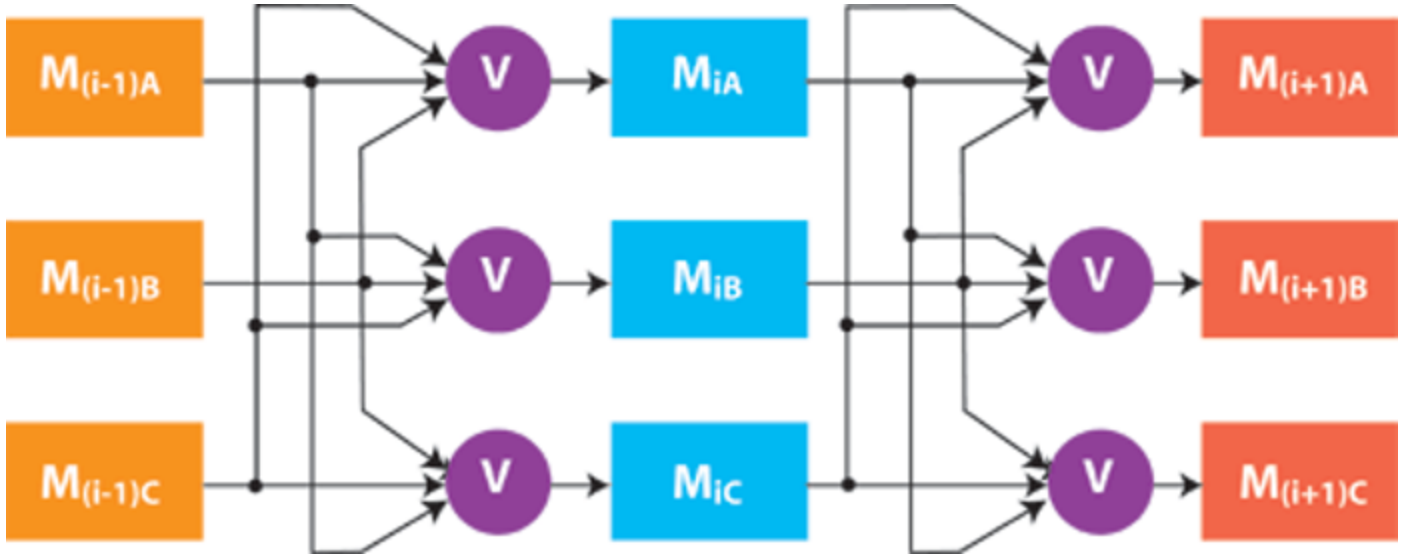
- správnost systému závisí nejen na logických výsledcích, ale také na čase, za který jsou výsledky vytvořeny.

Příklady systémů pracujících v reálném čase

- Real-time vestavné:
  - Systémy kritické z hlediska bezpečnosti: řízení jaderného reaktoru, řízení letu
  - GPS, MP3 přehrávač, mobilní telefon
- Real-time, ale ne vestavné:
  - Plaforma pro burzovní operace
  - Skype, Youtube, Netflix
- Vestavné, ale ne real-time:
  - Domácí termostat, zavlažovací systém
  - Pračka, lednička

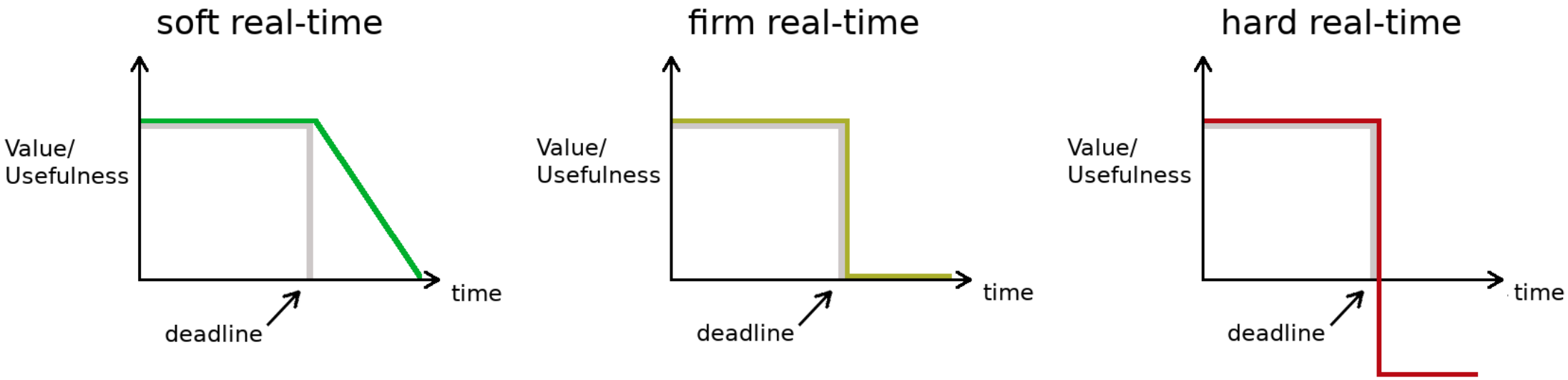
Charakteristiky real-time systémů

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



Klasifikace RT systémů

- Hard RT: reakce na vstupy musí přijít v požadovaném termínu - systémy řízení letů.
  - Real RT: navíc velmi krátká odezva, např. systém navádění raket
- 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.



Klasifikace RT systémů

Statické

- Lze předvídat časy příchodu úkolů
- Možnost statické analýzy (v době kompilace)
- Umožňuje dobré využití zdrojů (nízká doba nečinnosti procesorů)

Dynamické

- Nepředvídatelné časy příjezdu
- 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

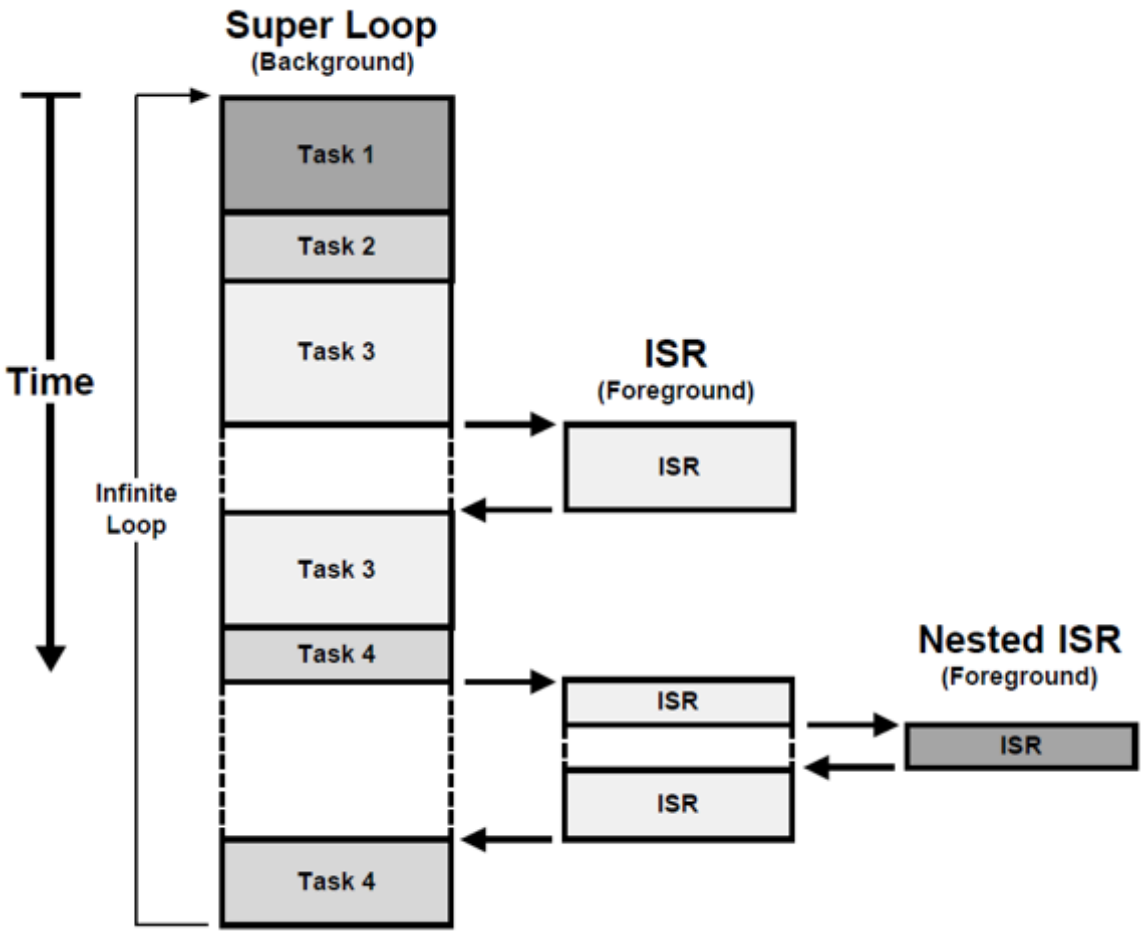
- Každá úloha (nebo skupina úloh) se provádí opakovaně s určitou periodou.
- Umožňuje použití některých technik statické analýzy
- Odpovídá charakteristikám mnoha skutečných problémů
- Je možné mít úlohy s termíny menšími, rovnými nebo většími, než je jejich perioda - pozdější jsou obtížně zpracovatelné,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

Běžný MCU systém

- Úroveň úlohy, úroveň přerušení
- Kritické operace se musí provádět na úrovni přerušení (není dobré)
- Doba odezvy/časování závisí na celé smyčce
- Změna kódu ovlivňuje časování
- Jednoduché, levné systémy



Real-time OS

- Co je to Operační Systém (OS)
- Jaké jsou základní koncepty operačního systému
- Co je to Real-time operační systém (RTOS)
- Jádro RTOS

Co je operační systém?

OS je vysoce organizovaná kolekce SW vybavení, která slouží jako

- rutiny pro kontrolu počítače (např. prástup k periferiím)
- umožňuje běh programů

Funkce operačního systému

- **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:**
  - **Řeší konflikty** o zdroje.
  - **Optimalizuje výkon** při více uživatelích.

Typy operačních systémů

- **Nejjednodušší**: Malé jádro na vestavěném procesoru.
- **Složitější**: Plně vybavený komerční operační systém:
  - Podpora více uživatelů a zabezpečení.
  - Podpora grafiky.
  - Podpora síťové komunikace.
  - Komunikace s periferiemi.
  - Současné provádění programů.

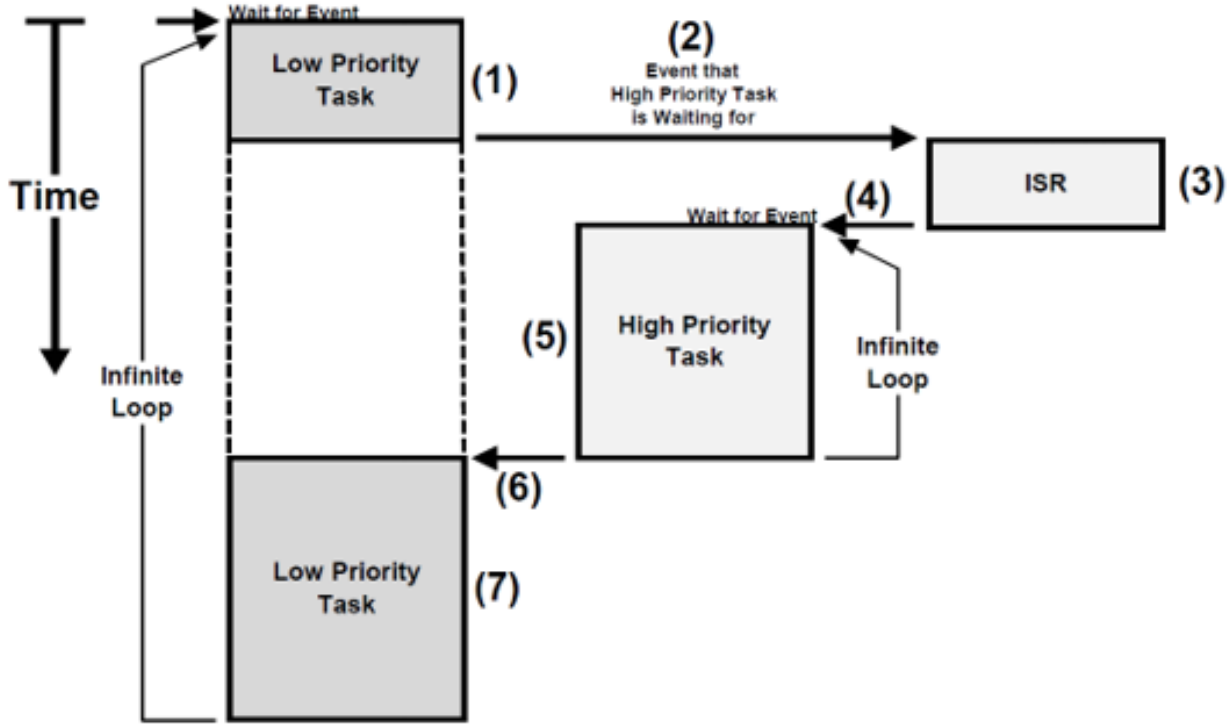
Úlohy a funkce

- Úloha je proces, který se opakuje
  - nekonečná smyčka
  - základní funkční blok RTOS
- Funkce je procedura, kterou voláme
  - běží, existuje, může vracet data
  - příklady
    - process_data()
    - int add_two_numbers(int a, int b)

RT systémy

- Časové požadavky různých podnětů/odpovědí
  - architektura systému umožňovat rychlé přepínání mezi zpracovateli podnětů.
- Různé prioritám, neznámé pořadí a různé časové požadavky
  - sekvenční smyčka obvykle nevyhovuje.

RT systémy se proto obvykle navrhují jako procesy spolupracující s jádrem reálného času, které tyto procesy řídí.



RTOS

- **Často**: RTOS = jádro operačního systému.
- **Vestavěné systémy** jsou navrženy pro jeden konkrétní účel, takže 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.

Komponenty jádra RTOS

- **Plánovač úloh**
  - Určuje, která úloha bude provedena jako další v multitaskingovém systému.
- **Dispatcher úloh**
  - Provádí kroky potřebné pro zahájení úlohy.
- **Komunikace mezi úlohami**
  - Podporuje komunikaci mezi jedním procesem (tj. úlohou) a jiným.

Strategie návrhu jádra reálného času

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

Cyklické dotazování (Polled Loops)

- **Nejjednodušší RT jádro**.
  - Jediná a opakující se instrukce testuje flag, který označuje, zda 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 dedikován pouze zpracování datového kanálu.

Příklad cyklického dotazování

```
while (TRUE) /* infinite loop, do forever */
{
  data_here==TRUE then  /* check for IFF data */
  {
    process_data();     /* call process_data() function*/
    data_here = FALSE;  /* reset flag */
  }
  /*
    testing another flag
  */
}
```

Jak to zlepšit?

Výhody a nevýhody cyklického 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 souběhu událostí
- Obecně není dostatečné pro zpracování složitých systémů.
- Zbytečné využívání času, zejména když se dotazovaná událost vyskytuje zřídka

Použití cyklického dotazování (Polled Loops)

- **Často se používá v rámci jiných schémat reálného času**, například pro:
  - **Dotazování senzorů na data**.
  - **Kontrolu uživatelských vstupů**
    - klávesnice,
    - UART data
- **Opačné k systémům řízeným přerušeními**.

Co je přerušení (rekapitulace)?

- **Hardwarový signál**, který iniciuje událost.
- Po obdržení přerušení procesor:
  - Dokončí právě vykonávanou instrukci.
  - Uloží programový čítač (aby se mohl vrátit na stejný bod vykonávání).
  - Načte programový čítač na adresu kódu obsluhy přerušení (ISR).
  - Vykoná obsluhu přerušení (ISR).
- Přerušení mohou být:
  - **Prioritizována** (některá přerušení jsou obsluhována dříve než jiná).
  - **Zakázána** (procesor je nezkontroluje nebo je ignoruje).
  - **Maskována** (procesor vidí pouze některá přerušení).

Co je přerušení (rekapitulace)?

- V praxi mohou RTOS zpracovávat několik přerušení v prioritním režimu:
  - Přerušení mohou být povolena/vypnuta (nastavením příslušných registrů).
  - Nejvyšší prioritu mají přerušení obsluhována jako první.
- Procesor musí velmi často kontrolovat přerušení: Pokud nějaké dorazilo, okamžitě zastaví svou činnost a vykoná přidruženou ISR.
  - Procesor opakuje: vykoná jednu operaci; zkontroluje přerušení; pokud nějaké existují, pozastaví úlohu a vykoná ISR.

ISR (Obsluha přerušení)

- **ISR je program**, který se spustí jako reakce na přerušení:
  - **Zakáže všechna přerušení**.
  - **Vyčistí flag přerušení**, který způsobilo jeho vyvolání.
  - **Spustí kód** pro obsluhu události.
  - **Opět povolí přerušení**.
  - **Ukončí** svou činnost, aby se procesor mohl vrátit k běžné úloze.
- **Měla by být co nejrychlejší**, protože během obsluhy přerušení nemůže probíhat žádná jiná činnost (pokud přerušení nastávají velmi často, úlohy jsou pozastaveny a postupují velmi pomalu, v nejhorším případě jedna instrukce na ISR).

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.

Přepínání kontextu (Context Switching)

- Když procesor přepne z jedné úlohy na jinou, říká se, že došlo k **přepnutí kontextu**.
- **Uloží se minimální množství informací potřebných k obnovení přerušeného procesu**:
  - Obsah registrů.
  - Obsah programového čítače.
  - Obsah registrů koprocesoru (pokud je použit).
  - 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**.

Jak sdílí mnoho úloh stejný CPU?

- Plánovací strategie
  - **Nepreemptivní plánování** - jakmile je proces naplánován k provedení, běží až do dokončení nebo dokud není z nějakého důvodu zablokován (např. čekání na vstup/výstup).
  - **Preemptivní plánování** - Provádění prováděných procesů může být zastaveno, pokud proces s vyšší prioritou vyžaduje obsluhu.
- Plánovací algoritmy
  - Round-robin - běžícímu procesu přiděluje kvantum času, po jeho uběhnutí je proces odstaven, předpokládá se konstatní priorita
  - Rate monotonic - statické přidělování priorit
  - Earliest deadline first - zpracování úloh probíhá na základě mezní doby platnosti procesu

Systémy foreground/background

- **Nejčastější hybridní řešení pro vestavěné aplikace**.
- Zahrnují **přerušení řízené procesy** (přední) a **nepřerušení řízené procesy** (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 a nevýhody multitaskingu

Výhody:

- **Segmentuje problém na malé, zvládnutelné části** (princip modulárního návrhu počítačového systému).
- Vytváří **modulárnější software** (umožňuje snadnější opětovné použití částí).
- Umožňuje **návrháři softwaru stanovit priority úloh** (některé úlohy mají přednost před jinými).

Nevýhody:

- V závislosti na implementaci **časování nemusí být deterministické** (jitrování způsobené variacemi v časování příchozích dat).
- **Přepínání kontextu přidává režii**.

Plně vybavený RTOS

- **Rozšiřuje řešení přední/pozadí**:
  - Přidává **síťová rozhraní**.
  - Přidává **ovladače zařízení**.
  - Přidává **komplexní nástroje pro ladění**.
- **Nejčastější volba pro složité systémy**.
- K dispozici je mnoho **komerčně dostupných operačních systémů**.

Multitasking

Pokud operační systém může vykonávat více úloh, které se zdají být prováděny současně, říká se, že je **multitaskingový**.

- **Koncový procesor** může vykonávat pouze jednu úlohu v daném okamžiku.
- Nicméně, **rychlé přepínání mezi úlohami** může způsobit, že se zdá, jako by každá úloha běžela současně.

Životní cyklus úlohy

- Úloha se nachází v různých stavech
  - běžící
    - využívá procesor
    - přerušením ji lze přesunout do stavu připravena
  - připravena (ready / sleep)
    - čeká na CPU / plánovač
    - plánovač přesouvá do stavu běžící
  - blokována
    - bežící úloha se dotáže na data z periferie
    - po získání dat se přesouvá do stavu připravena

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.
    - **Příklad 1**: Úloha čte data ze senzoru při 15 Hz. Uloží 1024 bajtů dat a poté musí signalizovat zpracovatelské úloze, aby data převzala a zpracovala, aby měla místo pro zápis dalších dat.
    - **Příklad 2**: Úloha určuje stav systému (např. Normální režim, Urgentní režim, Spící, Zakázáno). Musí informovat všechny ostatní úlohy v systému o změně stavu.
    - **Příklad 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 jsou úlohy obvykle zabráněny přístupem k paměti jiných úloh, což má dobrý důvod.

Sdílená paměť

Globální proměnné použité jako flagy

- Různé úlohy jsou spouštěny a každá z nich, jakmile skončí, **inkrementuje proměnnou nazvanou** `finished`.
- Jakmile `finished` dosáhne hodnoty odpovídající celkovému počtu spuštěných úloh, **výpočet je dokončen**.

```
int main(void)
{
    initialize_all();
    Initialize_kernel(3, SCHEDULING_QUANTUM);
    // void RegisterTask(double task_period, void* task_function)
    //      task_period: Task Period (in secs). 0 for non-periodic tasks
    //      task_function: Pointer to the task's function
    finished = 0;
    RegisterTask(0, (void*) &task1); //task1 increments finished when done
    RegisterTask(0, (void*) &task2); //task2 increments finished when done
    RegisterTask(0, (void*) &task3); //task3 increments finished when done
    /// Function to starts the tasks. This gives control to the scheduler.
    Run_tasks();
    // In our kernel this means that all tasks are being executed and the processor
    // will not get beyond this instruction.
    // In more advanced kernels, the tasks are spawned at the background
    // and the next instruction in the main loop is executed.
    while (finished != 3) ;
        printf("Done");
}
```

Sdílená paměť: Poštovní schránky (Mailboxes)

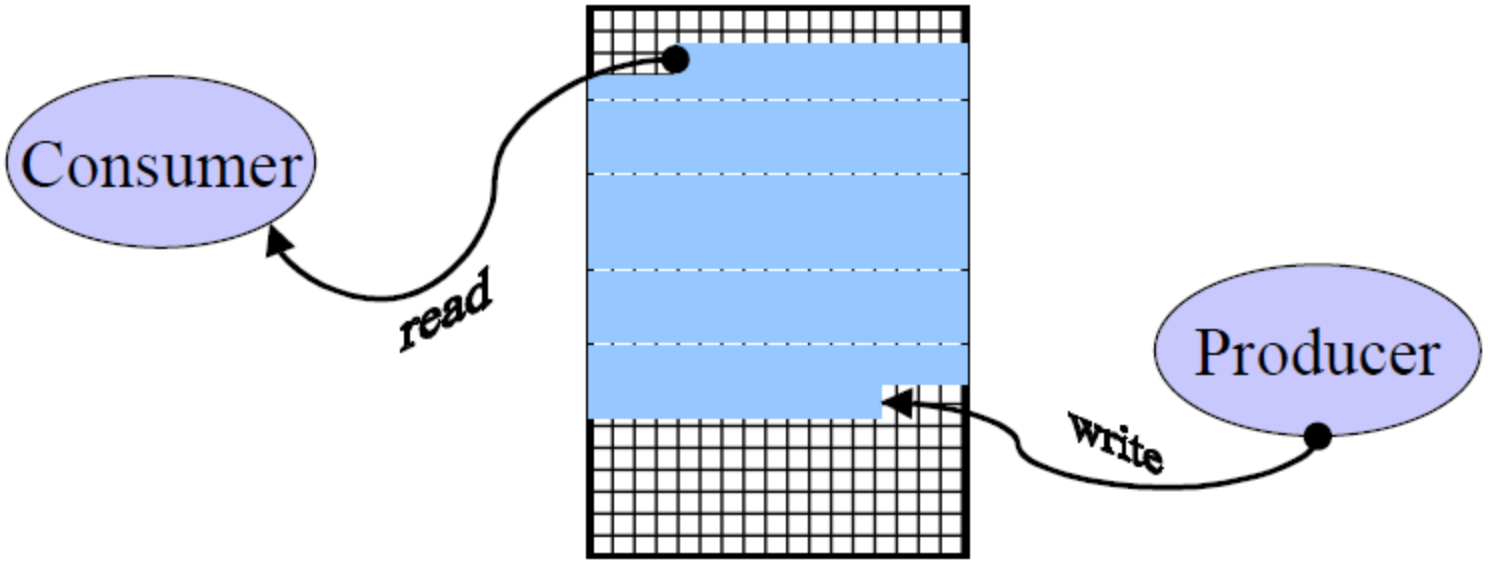
- **Post()** - zápisová operace, která umístí data do poštovní schránky.
- **Pend()** - čtecí operace, která získá data z poštovní schránky.

Funkce je podobná použití bufferu nebo sdílené paměti s následujícími rozdíly:

- Pokud **žádná data nejsou dostupná**, úloha čekající na **pend()** je **pozastavena**.
- **Vstavená vzájemná exkluze**: Pokud někdo právě provádí **post()**, úloha čekající na **pend()** musí počkat.
- **Žádný procesorový čas není zbytečně ztracen** na polling poštovní schránky, aby se zjistilo, jestli už jsou nějaká data dostupná.
- **Pend() může mít časový limit**, pro případ, že by žádná data nepřišla.

Sdílená paměť: Bufferování dat

- 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.



Sdílená paměť: kolize a nekonzistence

- **Sdílená paměť** může být tak jednoduchá, jako globální proměnná v C programu, nebo blok společné paměti poskytovaný operačním systémem.
- V programu s jednou úlohou víte, že **pouze jedna funkce bude přistupovat k proměnné v daném okamžiku**.
- Při dvou úlohách, které aktualizují stejný kousek paměti, mohou vzniknout **konflikty**:
    - Například, zvažme instrukci `x = x - a;`
        a. Načte se uložená proměnná `x`.
        b. Odečte se `a` (předpokládejme, že je již uložen v jednom z 32 registrů).
        c. `x` je nahrazeno výsledkem `x - a`.

Sdílená paměť: kolize a nekonzistence

- **Jedna úloha může přerušit jinou** v jakémkoli bodě:
  - Jedna instrukce v C je reprezentována několika assemblerovými instrukcemi; přerušení může nastat uprostřed.
  - **Hardwarová událost způsobující přerušení** může dokončit 4 assemblerové instrukce (což stačí k dokončení C instrukce jako `x = x - a;`, ale není dostatečné pro složitější řádek kódu v C).
- RTOS **řadí časové segmenty** na začátku a na konci na libovolných místech, pokud není explicitně instruováno.

V systémech s více úlohami je třeba být při modifikaci sdílených dat **opatrný**

- Chceme zajistit, aby v určitých kritických sekcích kódu **žádné dvě úlohy neměly přístup k datům současně**.

- Pokud nastavíme **flag** (paměť je zaneprázdněná, prosím počkejte), můžeme narazit na stejný problém, jako v předchozím příkladu:
    i. Předpokládejme, že `flag = 0`.
    ii. **Task1** provádí `while (flag != 0);`.
    iii. RTOS přepíná kontext na **Task2**.
    iv. **Task2** provádí `while (flag != 0);`.
    v. **Task2** vykoná svou další instrukci, která nastaví `flag = 1;`.
    vi. RTOS přepíná kontext na **Task1**.
    vii. **Task1** vykoná svou další instrukci (po smyčce) a nastaví `flag = 1;`.
    viii. Obě úlohy si myslí, že mají **exkluzivní vlastnictví paměti** odpovídající flagu... a začnou přistupovat k paměti "současně" (RTOS přepíná Task1 a Task2 do a z kontextu).

Tento scénář vede k problémům s nekonzistencí dat, protože dvě úlohy mohou provádět změny v paměti, aniž by věděly, že druhá úloha má přístup k té samé paměti.

Atomicita operací

- OS, který podporuje multitasking, musí současně podporovat atomická synchronizační primitiva - semafory
    - atomická operace je taková, kterou nelze přerušit
- Hlavní idea je taková, že dojde k uzamčení tzv. kritické sekce - to je zejména práce se zdroji
- Terminologie se liší
    - set / release
    - lock / unlock
    - wait / signal

Příklad

```
void SensedDataUpdate()
{
  lock(sensed_data);
  update(curr_sensed_data);
  unlock(sensed_data);
}
```

```
void SensedDataTransmit()
{
  lock(transmitter);
  // ---
  lock(sensed_data);
  transmit(curr_sensed_data);
  unlock(sensed_data);
  unlock(transmitter);
}
```

Jaký RTOS vybrat?

Spolehlivost - certifikace

- **DO-178B** pro systémy avioniky
- **IEC 61508** pro průmyslové řídicí systémy
- **ISO 62304** pro zdravotnické zařízení
- **SIL3 / SIL4** pro dopravu a jaderné systémy (Safety integrity level)

Operační systém RIOT

- Bezplatný operační systém s otevřeným zdrojovým kódem (LGPLv2.1)
- Programování je v jazyku C/C++ nebo Rust
- Podporuje standardní nástroje jako jsou gcc, gdb nebo valgrid.
- Architektury: AVR, ARM7, Cortex-M0, Cortex-M0 +, Cortex-M3, Cortex-M4, Cortex-M7, ESP8266, MIPS32, MSP430, PIC32, x86.
- Desky: Airfy Beacon, Arduino Due, Arduino Mega 2560, Arduino Zero, Atmel samr21-Xplained Pro, f4vi, mbed NXP LPC1768, Micro::bit, Nordic nrf51822 (DevKit), Nordic nrf52840 (DevKit), Nucleo desky (téměř všechny) a mnoho dalších.

Web: https://www.riot-os.org/

Příklad programu v RIOT OS

```
gpio_t pin_out = GPIO_PIN(PORT_B, 5);
if (gpio_init(pin_out, GPIO_OUT)) {
    printf("Error to initialize GPIO_PIN(%d %d)\n", PORT_B, 5);
    return -1;
}

while(1)
{
  printf("Set pin to HIGH\n");
  gpio_set(pin_out);
  xtimer_sleep(2);

  printf("Set pin to LOW\n");
  gpio_clear(pin_out);
  xtimer_sleep(2);
}
```

Web: https://www.hackster.io/ichatz/control-external-led-using-riot-os-b626da

Mongoose OS

- Framework pro vývoj firmwaru internetu věcí (Apache Licence 2.0 nebo Enterprise)
- Cílem je komplexní řešení pro vývoj a správu
- Nízkopříkonové MCU: ESP32, ESP8266, TI CC3200, STM32
- Cloudové integrace: AWS, Google, Azure, IBM Watson (komerční licence)
- K dispozici je Dashboard mDash (Apache 2.0)
    - stav připojených zařízení - online / offline, verze firmwaru, doba provozu atd.
    - aktualizace firmwaru OTA,
- K dispozici je také mobilní aplikace pro iOS i Android.

Web: https://mongoose-os.com/

Mongoose OS

Příklad čtení dat ze senzoru DHT22

```
#include "mgos.h"
#include "mgos_dht.h"

static void timer_cb(void *dht) {
  LOG(LL_INFO, ("Temperature: %lf", mgos_dht_get_temp(dht)));
}

enum mgos_app_init_result mgos_app_init(void) {
  struct mgos_dht *dht = mgos_dht_create(mgos_sys_config_get_app_pin(), DHT22);
  mgos_set_timer(1000, true, timer_cb, dht);
  return MGOS_APP_INIT_SUCCESS;
}
```

Contiki OS

- Operační systém s otevřeným zdrojovým kódem (BSD licence)
- Poskytuje multitasking, jádro 10kB RAM + 30kB ROM, jazyk C
- Simulátor Cooja umožňuje emulovat celou Contiki síť

Web: https://github.com/contiki-ng/contiki-ng/wiki

Projekt Zephyr

- Malý škálovatelný RTOS, původně pod patronací Linux Foundation (Apache licence)
- Jádro Zephyr je odvozeno od komerčního VxWorks Microkernel Profile.
- MCU: https://docs.zephyrproject.org/latest/boards/index.html
- Raspberry Pi Pico tutorial

Web: https://zephyrproject.org/

Blinky LED

- Zdrojový kód
- Konfigurace specifického boardu

Nucleus

- OS od divize Embedded Software společnosti Mentor Graphics
- Honeywell: systém varování před přiblížením k zemi v aplikaci pro letecký průmysl.
- Garmin: Avionics Navigator CNX80
- ZOLL: automatizovaný externí defibrilátor AED Plus
- MCU: https://www.mentor.com/embedded-software/nucleus/processor-support

---

Update 2023: RTOS převzal Siemens

Mbed OS

- RTOS určený primárně pro procesory ARM (licence Apache 2.0)
- Dostupný zdrojový kód: https://github.com/ARMmbed/mbed-os
- Podporuje BLE, NFC, RFID, LoRa, 6LoWPAN-ND, Thread, Wi-SUN, Ethernet, Wi-Fi, LPWAN
- Propojení s platformou Pelion IoT
- Nástroje: Mbed CLI, Mbed Online Compiler, Mbed Studio
- Desky, Komponenty

Web: https://os.mbed.com/

Co MicroPython?

```
import dht, machine, uasyncio as asyncio

async def report_dht_data(d):
  while True:
    await asyncio.sleep(2)
    try:
      wait_ms = d.start()
      await asyncio.sleep_ms(wait_ms)
      d.receive()
    except Exception as ex:
      print("error: {}".format(ex))
      continue

    print("temp: {}°C humi: {}%RH".format(d.temperature(), d.humidity()))

d = dht.DHT22(machine.Pin(4))
loop = asyncio.get_event_loop()
loop.create_task(report_dht_data(d))
loop.run_forever()
```

Využití více jader mikrokontroléru - vlákna

Raspberry Pi Pico

- Mikrokontrolér RP2040 obsahuje dvě jádra, **Core 0** a **Core 1**
- Defaulní mód:
  - **Core 0** vykonává všechny úlohy
  - **Core 1** zůstává ve standby módu
- MicroPython nabízí modul _thread, který umožňuje rozdělit úlohy na jednotlivá jádra.

Komunikace mezi jádry

- Aby spolu jádra mohla komunikovat, je modul Raspberry Pi Pico vybaven dvěma samostatnými FIFO.
- Každé jádro může přistupovat pouze k jedné FIFO, což pomáhá vyhnout se **race condition** nebo současnému zápisu na stejné místo v paměti.
- Modul **_thread** poskytuje synchronizační primitiva, **semafor** a **zámek**.
    - O vytvoření objektu pro uzamčení je k dispozici funkce **allocate_lock()**

Modul _thread

- Kód je automaticky spuštěn na jádře **Core 0**
- Modul **_thread** umožňuje spustit kód na **Core 1**

```
def thread_function():
  # code to be running on Core 1

new_thread = _thread.start_new_thread(thread_function, args, [,kwargs])
```

- **thread_function** je odkaz na standardní funkci jazyka Python, která obsahuje kód nového vlákna. Za ním musí následovat tuple obsahující argumenty funkce a pak nepovinný slovník nebo klíčová slova argumentů funkce.

Příklad

```
from time import sleep
import _thread

def core0_thread(counter = 0):
    while True:
        print(counter)
        counter += 2
        sleep(1)

def core1_thread(counter = 1):
    while True:
        print(counter)
        counter += 2
        sleep(2)

second_thread = _thread.start_new_thread(core1_thread, ())

core0_thread()
```

Komunikace mezi vlákny

```
def core0_thread():
    global run_core_1
    # do something
    # signal core 1 to run
    run_core_1 = True
    # wait for core 1 to finish
    while run_core_1:
      pass

def core1_thread():
    global run_core_1
    while True:
        # wait for core 0 to signal start
        while not run_core_1:
            pass
        # do something
        # signal core 0 code finished

run_core_1 = False

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Sdílení zdrojů

- Někdy musíme velmi pečlivě sledovat, kdo a kdy může mít přístup k některým datům nebo prostředkům, např. k rozhraní SPI.
    - Pokud se obě vlákna pokusí použít nebo aktualizovat stejný prostředek současně, dojde buď k poškození dat, nebo k potenciálnímu pádu části kódu.
- V jednoduchých a dobře definovaných situacích funguje uspokojivě **vlajka**. Když potřebujete flexibilnější ovládání, musíte použít zámek.
    - Zámek umožňuje řídit přístup.
    - Vytvoříme objekt zámku a prostředek může používat pouze jeho vlastník.
    - Každé další vlákno musí počkat, až vlastník zámku zámek uvolní, a teprve potom může jedno z čekajících vláken převzít vlastnictví a získat přístup k prostředku.

Sdílení zdrojů - race condition

```
def core0_thread():
  while True:
    print('A')
    sleep(0.5)

def core1_thread():
  while True:
    print('B')
    sleep(0.5)

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Sdílení zdrojů - použití zámku (mutex)

```
def core0_thread():
  global lock
  while True:
    lock.acquire()
    print('A'); sleep(0.5)
    lock.release()

def core1_thread():
  global lock
  while True:
    lock.acquire()
    print('B'); sleep(0.5)
    lock.release()

lock = _thread.allocate_lock()

second_thread = _thread.start_new_thread(core1_thread, ())
core0_thread()
```

Komunikace mezi vlákny využívající Frontu (Queue)

- Queue() má implementaci bezpečnou pro vlákna se všemi potřebnými zamykacími mechanismy.
  - může v podstatě předat frontu jako argument druhému vláknu.

Core0 thread:

```
q = queue.Queue()
_thread.start_new_thread(second_core_thread, (q) )

while True:
  msg = q.get()
```

Core 1 thread (second_core_thread):

```
def writer(q):
  queue.put(....)
```