FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Implementace a srovnání plánovacích algoritmů pro systémy reálného času
Student:	Josef Zápotocký
Vedoucí:	doc. Ing. Hana Kubátová, CSc.
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2020/2021

Pokyny pro vypracování

Prostudujte různé typy algoritmů použitelných pro systémy reálného času.
Implementujte vybrané z nich (nejlépe v jazyce C) tak, aby byly aplikovatelné pro vestavné systémy.
Srovnejte a zhodnoťte algoritmy a jejich implementaci s ohledem na předem daná návrhová omezení (determinismus, velikost, maximální pracovní frekvence, plnění deadlinů, apod.).
Výběr přizpůsobte možnostem využití při výuce bakalářského předmětu "Systémy reálného času", používání konkrétních přípravků v laboratorní výuce a operačního systému RTOS.

Elektronicky schválil/a doc. Ing. Hana Kubátová, CSc. dne 2. listopadu 2019 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Implementace a srovnání plánovacích algoritmů pro systémy reálného času

Josef Zápotocký

Katedra číslicového návrhu
Vedoucí práce: doc. Ing. Hana Kubátová, CSc.

12. května 2021

Poděkování

Zde bych chtěl poděkovat všem, kdo měli trpělivost se mnou během vypracovávání bakalářské práce, a to vedoucí doc. Ing. Haně Kubátové, CSc., mému oponentovi Ing. Jaroslavu Boreckýmu, Ph.D., pomáhajícímu učiteli češtiny Jiřímu Tischlerovi a zkoušející komisi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2021 .....

Odkaz na tuto práci

Zápotocký, Josef. *Implementace a srovnání plánovacích algoritmů pro systémy reálného času.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá plánovacími algoritmy pro systémy reálného času, zkoumá a modifikuje operační systém reálného času FreeRTOS. FreeRTOS je speciálně vyvinutý pro malé vestavěné systémy, tak aby uspokojil jak nároky uživatele tak paměťové nároky malých vestavných zařízení. Po podrobném popisu prioritního plánovače přijatého z FreeRTOS jsou navrženy dva učební plánovače: první je založen na známém algoritmu přednosti s nejbližší uzávěrkou (EDF), druhý je založen na algoritmu přednosti úkolu s nejmenší laxitou (LLF), původně vyvinutý pro systémy s více procesory. U každého navrhovaného plánovače je uveden popis funkčnosti plánovače, ukázka práce daného plánovacího algoritmu a následná implementace ve FreeRTOS. Poté je správnost plánovacích algoritmů implementované ve FreeRTOS ověřena testem. Plánvací algoritmy byly vybrány na základě užitečnosti v bakalářském předmětu BI-SRC na FIT ČVUT v Praze.

Klíčová slova FreeRTOS, EDF, LLF, Implementace, Testování

Abstract

This work deals with scheduling algorithms for real-time systems, examines and modifies the real-time operating system FreeRTOS. FreeRTOS is specially developed for small embedded systems to meet both user and memory requirements. After a detailed description of the priority scheduler adopted by FreeRTOS, two learning schedulers are proposed: the first is based on the known Earliest Deadline First Algorithm (EDF), the second is based on the Least Laxity First Algorithm (LLF), originally developed for multiprocessor systems. For each proposed scheduler, a description of the scheduler's functionality, a demonstration of the work of the scheduling algorithm and subsequent implementation in FreeRTOS is given. Then the correctness of the planning algorithms implemented in FreeRTOS is verified by a test. Planning algorithms were selected on the basis of usefulness in the bachelor's course BI-SRC at FIT CTU in Prague.

Keywords FreeRTOS, EDF, LLF, Implementation, Testing

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza a návrh **5**
2.1 FreeRTOS 6
2.1.1 Struktura jádra 6
2.1.2 Struktura úloh 7
2.1.3 Stavy úloh 10
2.1.4 Inicializace úloh 12
2.1.5 Uspání úloh 15
2.1.6 Přepínání kontextu 16
2.1.7 Systém tiků 18
2.1.8 Ukázka plánování 19
2.2 EDF 22
2.2.1 Ukázka plánování 23
2.3 LLF 23
2.3.1 Ukázka plánování 23

3 Implementace **25**
3.1 EDF 25
3.2 LLF 33

4 Testování **41**
4.1 Testování FreeRTOS plánovače 41
4.2 Testování EDF 45
4.3 Testování LLF 49
4.4 Závěr testování 53

Závěr **55**

Literatura **57**

A Seznam použitých zkratek **59**

B Obsah přiloženého CD **61**

Seznam obrázků

2.1 Graf stavů a přechodů úlohy 11
2.2 xTaskCreate 13
2.3 Kontext spouštění úlohy 17
2.4 Zásobník úlohy po uložení kontextu 17
2.5 Příklad interakce přepínání obsahu úloh 20
2.6 Příklad interakce přepínání obsahu úloh 20
2.7 Příklad interakce přepínání obsahu úloh 21
2.8 Příklad interakce přepínání obsahu úloh 21
2.9 Plánování úloh 23
2.10 Plánování úloh 24

3.1 Nový list pro EDF 26
3.2 Implemenetace EDF 27
3.3 Implemenetace EDF 28
3.4 Implemenetace EDF 29
3.5 Nový list pro LLF 35

4.1 Průběh FreeRTOS plánovače 44
4.2 Testování Plánování úloh1 48
4.3 Průběh EDF 48
4.4 Plánování úloh 52
4.5 Průběh LLF 52

Seznam tabulek

2.1 FreeRTOS vrstvy 6

Úvod

Pokrok v technologiích za posledních 30 let a jejich uplatňování ve světě na místech, kde může jít vyloženě o život, nám přivedlo nároky na ně takové, že už jednoduché algoritmy bývají někdy příliš málo, a proto nám tyto nároky daly operační systémy reálného času, u kterých je vyžadováno, aby vždy byly v čas splněny kritické úlohy v daný časový okamžik bez toho aniž by se systém zasekl, či dané úloze překážela jiná úloha v dodělání. Proto do daných systémů reálného času se implementovaly různé algoritmy pro plánovaní úloh, které se s daným problémem plánování úloh vypořádávají jinak, a bohužel dochází i k případům, že se přerozdělení úloh nedá uspořádat tak aby je systém zvládl všechny zpracovat [1]. Pokud systém nedokáže zpracovat dané úlohy, tak selže, a selhání těchto systémů může mít kritické následky.

Práce se proto zabývá otevřeným operačním systémem reálného času FreeRTOS [2], který bývá uplatněn v mnoha odvětvích a zařízeních. Například je uplatněn v mesh sítí či v železniční dopravě. Úkolem zde bude modifikovat jádro tohoto operačního systému, aby používal konkrétní námi implementované plánovače, a to plánovače EDF a LLF. Tyto plánovací algoritmy byly záměrně vybrány, jelikož jejich znalost a pochopení je součástí bakalářského předmětu Systémy reálného času (BI-SRC), a tato bakalářská práce by měla sloužit jako způsob, jak porozumět jejich rozhodování jiným způsobem než si to vše kreslit na papír v praktické části výuky.

Práce představí, co je to FreeRTOS po technické stránce, které soubory budou modifikovány, aby výsledná aplikace používala daný plánovací algoritmus. Bude zde zmíněna teorie k vybraným plánovacím algoritmům a jaký plánovací algoritmus FreeRTOS používá. Po implementaci algoritmů budou následně otestovány a porovnány mezi sebou. Následně si určíme, který z nich obstál nejlépe a nějakou teorií za tím proč.

Kapitola 1

Cíl práce

Cílem této bakalářské práce je prostudovat plánovací algoritmy pro systémy reálného času a následně vybrat a implementovat v jazyku C nové plánovací algoritmy pro FreeRTOS. U každého vybraného navrhovaného plánovacího algoritmu je uveden popis implementace ve FreeRTOS a příklad funkčnosti daného plánovacího algoritmu. Ke každému z těchto algoritmů budou vytvořeny testovací úlohy, aby se provedly a aby se neprovedly. Poté je jejich správnost ověřena testovací fází, ve které budou srovnány a zhodnoceny z hlediska rychlosti, determinismu, plnění dealinů a podobně. Výběr plánovacích algoritmů bude přizpůsoben možnostem využití při výuce bakalářského předmětu "Systémy reálného času"(BI-SRC) vyučovaném na FIT ČVUT v Praze.

Pro práci byly vybrány dva algoritmy dynamických priorit první z nich je založen na známém algoritmu Earliest Deadline First (EDF), druhý je založen na novějším algoritmu Least Laxity First (LLF)

Kapitola 2

Analýza a návrh

Jeden z problémů plánování pro systémy reálného času je určit, v jakém pořadí by se měly úkoly provést, aby byly uspokojeny jejich časové nároky a omezení. Abychom mohli plánovat v systému reálného času, potřebujeme mít dostatek informací, jako je deadline, čas ukončení, pro případ porušení deadlinu, a jak dlouho trvá splnění každého úkolu. Rovněž je dobré znát důležitost úkolu ve srovnání s ostatními úkoly a závislosti dané úlohy. Většina systémů předpokládá, že většina těchto informací je k dispozici předem.

Algoritmus plánovače v reálném čase lze klasifikovat podle několika vlastností:

1. **preemptivní / nepreemptivní chování**
   - **Preemptivní:** Úlohy reálného času můžou být přerušeny, pokud je potřeba.
   - **Nepreemptivní:** Úlohy nemůžou být přerušeny.
2. **Periodické / sporadické řízení úkolů**
   - **Periodické:** Jsou vydávány pravidelně za pevné sazby (období). Většina senzorického zpracování má periodickou povahu. Například digitální teploměr, který měří teplotu v průmyslové nádrži, produkuje data s pevnou rychlostí.
   - **Sporadické:** Úlohy v reálném čase jsou aktivovány nepravidelně se známou omezenou rychlostí, není u nich znám přesný průběh úlohy.
3. **Statické / dynamické plánování priorit:** Při plánování podle priorit je každému úkolu přiřazena priorita.
   - **Statické:** Priorita úloh je přiřazena na začátku programu a je stejná (statická) během celého běhu programu.

- **Dynamické:** Priorita je určována za běhu programu v okamžicích kde se plánovací algoritmus rozhoduje, kterou úlohu vybere.

2.1 FreeRTOS

FreeRTOS je operační systém reálného času (RTOS), který je navržen tak, aby byl dostatečně malý pro provoz na mikrokontroléru, i když jeho použití není omezeno na aplikace mikrokontroléru. Projekt FreeRTOS byl zahájen v roce 2002 a je v aktivním vývoji. Jeho oficiální podpora až 35 integrovaných systémových architektur a různých více platformových překladačů, je to jednoduchý a plně dokumentovaný API a díky opensourcové licenci se velmi rychle rozšířil na volném trhu, díky čemuž uživatelská základna rok od roku roste. FreeRTOS poskytuje základní funkce plánování v reálném čase, komunikaci mezi úkoly, časování a synchronizaci. Další funkce, jako například rozhraní příkazové řádky nebo network stack (síťový zásobník), mohou být zahrnuty jako doplňující funkce FreeRTOS. Plánovač FreeRTOS je preemptivní a založený na pevné prioritě. Při inicializaci úkolu je přiřazena priorita jednotlivým úlohám. Pokud má více úkolů stejnou prioritu, pak úlohy jsou rovnoměrně vybírány z cyklické fronty.

2.1.1 Struktura jádra

Protože FreeRTOS funguje v prostředích vestavných systémů, je navržen tak, aby minimalizoval využití paměti a je také vhodný pro mikroprocesory s nízkou frekvencí. Pro funkčnost potřebuje jádro FreeRTOS pouze tři zdrojové soubory, které dohromady mají méně než 9000 řádků kódu. Aby byl kompatibilní se všemi podporovanými architekturami, tvoří jádro FreeRTOS 2 vrstvy:

1. **hardware závislá:** je přizpůsobená pro každou podporovanou architekturu

2. **hardware nezávislá:** je společná pro všechny porty

Obrázek 2.1 ukazuje vrstvy FreeRTOS. Poskytují 3 zdrojové soubory, které tvoří minimální jádro (spolu s několika hlavičkovými soubory) pro tyto funkce:

Tabulka 2.1: vrstvy

FreeRTOS uživatelské úlohy a ISR kód
FreeRTOS hardwarově nezávislý kód
FreeRTOS hardwarově závislý kód
FreeRTOS hardware

- **task.c:** zde jsou definované úlohy, a je zde řízen životní cyklus programu. plánovací funkce jsou zde také definované.
- **queue.c:** V tomto souboru jsou definované struktury používané pro komunikaci a synchronizaci úkolů. Úkoly a přerušení spolu komunikují pomocí front. Pomocí front si předávají zprávy,semafory a zámky(mutexy), které se používají k synchronizaci kritických sekcích zdroje.
- **list.c:** je definována datová struktura seznamů a jejich funkce. Seznamy se používají jak funkcemi úloh, tak frontami.

2.1.2 Struktura úloh

Úkoly jsou implementovány jako funkce v jazyku C. Každý vytvořený úkol je malý program s vlastními právy v rámci svých přiřazených priorit. Každý úkol se spouští svůj vlastní obsah, bez závislostí na obsahu jiného úkolu. V každém okamžiku FreeRTOS vybere úkol, který bude spouštět v závislosti na prioritě. U každého úkolu FreeRTOS přidruží správnou datovou strukturu tzv. Task Control Block (TCB) [2], který obsahuje následující parametry:

```
 /* The old naming convention is used to
 prevent breaking kernel aware debuggers. */
typedef struct tskTaskControlBlock
{
 /* Points to the location of the last item placed on the tasks
    stack. THIS MUST BE THE FIRST MEMBER OF THE TCB STRUCT. */

    volatile StackType_t * pxTopOfStack;
    /* The MPU settings are defined as part of the port layer.
        THIS MUST BE THE SECOND MEMBER OF THE TCB STRUCT. */
    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS xMPUSettings;
    #endif

    #if ( configUSE_EDF_SCHEDULER == 1 )
        TickType_t xTaskPeriod;
    #endif
    /* The list that the state list item of a task is reference
       from denotes the state of that task (Ready, Blocked,
       Suspended ). */
    ListItem_t xStateListItem;
    /* Used to reference a task from  an event list. */
    ListItem_t xEventListItem;
    /* The priority of the task. 0 is the lowest priority. */
```

```
    UBaseType_t uxPriority;
    /* Points to the start of the stack. */
    StackType_t * pxStack;
    /* Descriptive name given to the task when created.
    Facilitates debugging only. */
    /*lint !e971 Unqualified char types are allowed for strings
        and single characters only. */
    char pcTaskName[ configMAX_TASK_NAME_LEN ];

    #if ( ( portSTACK_GROWTH > 0 ) ||
          ( configRECORD_STACK_HIGH_ADDRESS == 1 ) )
        /* Points to the highest valid address for the stack. */
        StackType_t * pxEndOfStack;
    #endif

    #if ( portCRITICAL_NESTING_IN_TCB == 1 )
        /* Holds the critical section nesting depth
           for ports that do not maintain their own
           count in the port layer. */

        UBaseType_t uxCriticalNesting;
    #endif

    #if ( configUSE_TRACE_FACILITY == 1 )
        /*  Stores a number that increments each time a
            TCB is created.  It allows debuggers to
            determine when a task has been deleted and
            then recreated. */
        UBaseType_t uxTCBNumber;
        /*  Stores a number specifically for use by
            third party trace code. */
        UBaseType_t uxTaskNumber;
    #endif

    #if ( configUSE_MUTEXES == 1 )
        /*  The priority last assigned to the task -
            used by the priority inheritance mechanism. */
        UBaseType_t uxBasePriority;
        UBaseType_t uxMutexesHeld;
    #endif

    #if ( configUSE_APPLICATION_TASK_TAG == 1 )
        TaskHookFunction_t pxTaskTag;
    #endif
```

```
#if ( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )
    void * pvThreadLocalStoragePointers[
                configNUM_THREAD_LOCAL_STORAGE_POINTERS
                ];
#endif

#if ( configGENERATE_RUN_TIME_STATS == 1 )
     /*  Stores the amount of time the task has spent
         in the Running state. */
    uint32_t ulRunTimeCounter;
#endif

#if ( configUSE_NEWLIB_REENTRANT == 1 )

    /* Allocate a Newlib reent structure that is specific to
     * this task. Note Newlib support has been included by
     * popular demand, but is not used by the FreeRTOS
     * maintainers themselves.  FreeRTOS is not responsible
     * for resulting newlib operation. User must be familiar
     * with newlib and must provide system-wide
     * implementations of the necessary stubs. Be warned
     * that (at the time of writing) the current newlib
     * design implements a system-wide malloc() that must
     * be provided with locks.
     *
     * See the third party link
     * http://www.nadler.com/embedded/newlibAndFreeRTOS.html
     * for additional information. */
    struct  _reent xNewLib_reent;
#endif

#if ( configUSE_TASK_NOTIFICATIONS == 1 )
    volatile uint32_t ulNotifiedValue[
                configTASK_NOTIFICATION_ARRAY_ENTRIES ];
    volatile uint8_t ucNotifyState[
                configTASK_NOTIFICATION_ARRAY_ENTRIES ];
#endif

/* See the comments in FreeRTOS.h with the definition of
 * tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE. */
 /*lint !e731 !e9029 Macro has been consolidated for
 readability reasons. */
#if ( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
```

```
        /*  Set to pdTRUE if the task is a statically allocated
            to ensure no attempt is made to free the memory. */
        uint8_t ucStaticallyAllocated;
    #endif

    #if ( INCLUDE_xTaskAbortDelay == 1 )
        uint8_t ucDelayAborted;
    #endif

    #if ( configUSE_POSIX_ERRNO == 1 )
        int iTaskErrno;
    #endif
} tskTCB;
```

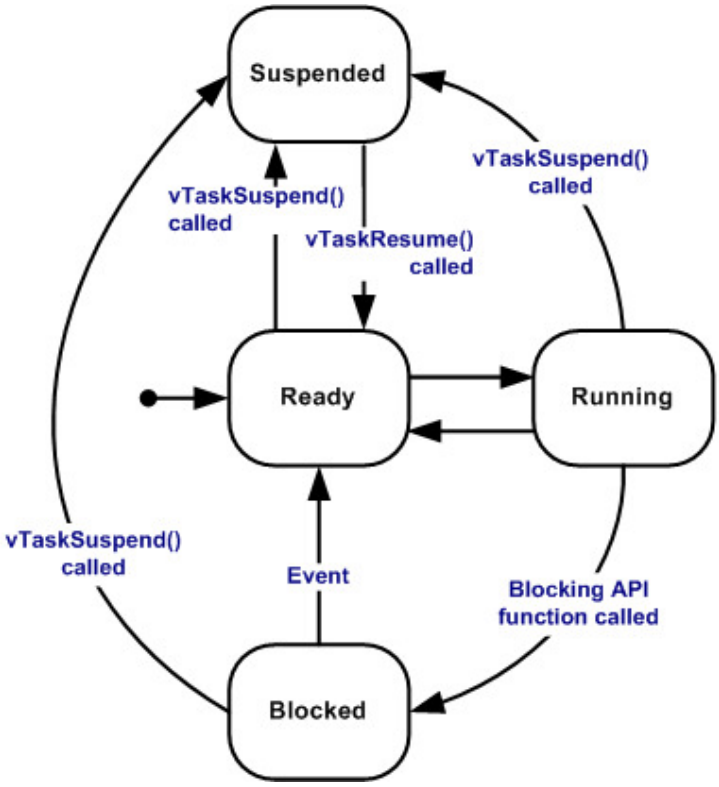
TCB obsahuje základní informace charakterizující danou úlohu:

- ukazatele zásobníků `*pxStack` který ukazuje na začátek zásobníku úlohy, `*pxTopOfStack` ukazuje na místo posledního vloženého předmětu zásobníku úlohy a `*pxEndOfStack`, tento ukazuje na místo prvního vloženého předmětu, převážně používán pro kontrolu přetečení zásobníku.
- `uxPriority` je proměnná obsahující prioritu úlohy, kdežto `uxBasePriority` obsahuje poslední přidělenou prioritu (použitý mechanismem pro dědění priority).
- Objekty typu `ListItem\_t xStateListItem` a `xEventListItem`: když je úloha vložena do listu (například list připravených úloh, jak uvidíme dále). Listy neobsahují jednoduchý ukazatele na TCB, ale ukazatele na další objekt stejného datového typu. Použití `ListItem\_t` nám zaručuje, že list spravující tyto položky bude mnohem chytřejší a operace nad ním zaberou méně výpočetního času.
- `pcTaskName` je statické pole obsahující jméno úlohy.

2.1.3 Stavy úloh

Jak nám určuje graf 2.1, úloha může existovat pouze v následujících stavech

- **Běží:** Úloha, na kterou ukazuje systémová proměnná `*pcCurrentTCB`, je, jak název napovídá, v běhu. V daný okamžik může běžet pouze jedna úloha.
- **Připravena:** Úlohy, které jsou připraveny k běhu a čekají na RTOS k naplánování, ale nic nedělají, jelikož jiná úloha se stejnou nebo vyšší prioritou momentálně běží.



Obrázek 2.1: Graf stavů a přechodů úlohy [FreeRTOS.org]

- **Blokovaná:** Úloha v tomto stavu nemůže být plánována, protože čeká na zásah zvenčí nebo časem závislou událost. Například: úloha po zavolání metody `vTaskDelay(TickType\_t)` se sama zablokuje a čeká, až od systému přijde událost o uplynulém čase, nebo úloha může čekat na semaforu, než dokončí svojí rutinu jiná úloha a pošle událost o dokončení.
- **Pozastavená:** Úloha se do tohoto stavu může dostat pouze zavoláním metod `vTaskSuspend()` [3] a `vTaskResume()` [3]. Úlohy ve stavu zadržení nemohou být plánovány.

TCB neobsahuje proměnou, která přímo reprezentuje stav dané úlohy, místo toho FreeRTOS obsluhuje list obsahující, úlohy pro každý ze stavů (Připravena, Blokovaná a Pozastavena), proto úlohy mohou být implicitně sledovány tak, že vložíme danou úlohu do daného listu.

- **Připravené úlohy:**

```
PRIVILEGED_DATA static List_t
                pxReadyTasksLists[ configMAX_PRIORITIES ];
            /*< Prioritised ready tasks. */
```

`pxReadyTasksLists[]` je pole listů, obsahující tolik listů, kolik je nakonfigurovaných priorit. I-tá pozice nám určuje list úloh s prioritou i.

- **Blokované úlohy:**

```
/*  Delayed tasks (two lists are used - one for delays that
have overflowed the current tick count. */
PRIVILEGED_DATA static List_t xDelayedTaskList2;
/*  Points to the delayed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;
/*  Points to the delayed task list currently being used to
    hold tasks that have overflowed the current tick count.
    */
PRIVILEGED_DATA static List_t * volatile
                                pxOverflowDelayedTaskList;
/*  Tasks that have been readied while the scheduler was
    suspended. They will be moved to the ready list when
    the scheduler is resumed. */
PRIVILEGED_DATA static List_t xPendingReadyList;
```

- **Pozastavené úlohy:**

```
/*< Tasks that are currently suspended. */
PRIVILEGED_DATA static List_t xSuspendedTaskList;
```

jednoduchý list obsahující pozastavené úlohy.

2.1.4 Inicializace úloh

Úloha je vytvářena zavoláním metody `xTaskCreate()` [2.2] ze souboru *task.c* `xTaskCreate()` vytváří novou úlohu s přiřazenou prioritou a vloží ji do listu připravených úloh. Podrobně se úloha vytváří takto:

Obrázek 2.2: xTaskCreate

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
                        /*  lint !e971 Unqualified char types are
                            allowed for strings and single
                            characters only. */
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask )



#if ( portSTACK_GROWTH > 0 )
{
    pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
    if( pxNewTCB != NULL ) {
        pxNewTCB->pxStack = ( StackType_t * ) pvPortMalloc(
                                    ( ( ( size_t ) usStackDepth )
                                    * sizeof( StackType_t ) ) );

        if( pxNewTCB->pxStack == NULL ) {
            vPortFree( pxNewTCB );
            pxNewTCB = NULL;
        }
    }
}
#else /* portSTACK_GROWTH */
{
    StackType_t * pxStack;
    pxStack = pvPortMalloc( ( ( ( size_t ) usStackDepth ) *
                                sizeof( StackType_t ) ) );
    if( pxStack != NULL ) {
        /* Allocate space for the TCB. */
        pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
        if( pxNewTCB != NULL ){
            /* Store the stack location in the TCB. */
            pxNewTCB->pxStack = pxStack;
        } else {
            vPortFree( pxStack );
        }
    } else {
```

```
        pxNewTCB = NULL;
    }
}
#endif /* portSTACK_GROWTH */
```

paměťový prostor je vyhrazen pro novou TCB strukturu a její zásobník, pokud je dostatečně volného prostoru na paměti. Kód včetně poznámek se nachází v souboru *task.c*.

```
/*
 * Called after a Task_t structure has been allocated either
 * statically or dynamically to fill in the structure's members.
 */
static void prvInitialiseNewTask( TaskFunction_t pxTaskCode,
        const char * const pcName, const uint32_t ulStackDepth,
        void * const pvParameters, UBaseType_t uxPriority,
        TaskHandle_t * const pxCreatedTask, TCB_t * pxNewTCB,
        const MemoryRegion_t * const xRegions ) PRIVILEGED_FUNCTION;
```

TCB proměnné a zásobník jsou zde inicializovány, nový zásobník dané úlohy je inicializován ve stejném duchu jako zásobník pro úlohy pozastavené plánovačem. Tímto způsobem plánovač nepotřebuje řešit inicializaci jako speciální případ pro řízení úloh, když tyto dvě řešení vypadají stejně, jako při inicializaci uspaných úloh. Metoda `prvInitialiseNewTask()` je hardwarově závislá, a proto je implementována v daném port.c souboru pro danou architekturu. Jak brzy uvidíme, když úloha je přerušena, všechen obsah úlohy je uložen na zásobník. Takže nově vytvořený zásobník je upraven, a přesto vypadá, jako kdyby nové registry byly vloženy, přestože zásobník nebyl ještě použit.

```
        prvAddTaskToReadyList( pxTCB )
```

je makro, které po zavolání vloží nově vytvořenou úlohu do patřičného prioritního listu `pxReadyTasksList[]`. Jak bylo řečeno dříve, pole `pxReadyTasksList[]` obsahuje jeden list pro každou prioritu, kterou systém má nastavenou, kde 0 je nejmenší priorita. Úloha s prioritou p bude zařazená do příslušného listu `pxReadyTasksList[p]`. `prvAddTaskToReadyList()` je definován takto:

```
#define prvAddTaskToReadyList( pxTCB ) \
    traceMOVED_TASK_TO_READY_STATE( pxTCB ); \
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority ); \
    vListInsertEnd( &( pxReadyTasksLists[ \
                        ( pxTCB )->uxPriority ] ), \
```

```
                        &( ( pxTCB )->xStateListItem ) ); \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
```

V zásadě, proměnná `UBaseType_t uxTopReadyPriority`, reprezentující v každém okamžiku prioritu úlohy v běhu, je porovnána s prioritou nové úlohy, a pokud je tato nová priorita vyšší, tak `uxTopTaskReadyPriority` je aktualizován na novou prioritu. Poté, `xStateListItem` je vložena na konec příslušného prioritního listu v poli `pxReadyTasksList[]`.

Úloha je nyní vytvořena a připravena na spuštění plánovačem úloh.

2.1.5 Uspání úloh

Již víme, jak funguje vytvoření úloh a její inicializace do stavu **Ready**. Nyní se podíváme na to, jak se úloha může dostat do stavu **Blokovaná** voláním metody `xTaskDelayUntil()`. Tato metoda definuje frekvenci, při jaké se úlohy periodicky spouštějí, takže může být použita pro implementaci periodických úloh. Jak uvidíme, FreeRTOS měří čas periodickým inkrementováním hodnotu v proměnné `xTickCount`. `xTaskDelayUntil()` přesune úlohu volající metodu do `pxDelayedTaskList`, kde čeká, než přijde než přijde událost, a následně je přesunuta do `xReadyTasksList[]` periodicky.

```
\*
 * @param pxPreviousWakeTime Pointer to a variable that holds the
 * time at which the task was last unblocked.  The variable must
 * be initialised with the current time prior to its first use
 * (see the example below).  Following this the variable is
 * automatically updated within xTaskDelayUntil ().
 *
 * @param xTimeIncrement The cycle time period.  The task will be
 * unblocked at time *pxPreviousWakeTime + xTimeIncrement. Calling
 * xTaskDelayUntil with the same xTimeIncrement parameter value
 * will cause the task to execute with a fixed interface period.
 *\
```

Metoda pracuje tímto způsobem:

```
uxListRemove( &( pxCurrentTCB->xStateListItem )
```

`pxCurrentTCB` je ukazatel na úlohu v běhu, která si zavolala `vTaskDelayUntil()`, takže její `xStateListItem` je vymazán z listu připravených úloh, ve kterém byl původně uložen.

```
listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xStateListItem ),
                         xTimeToWake );
```

`xStateListItem` nyní obsahuje hodnotu kdy bude odblokován.

```
vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
```

`xStateListItem` je následně vložen do `pxDelayedTaskList`, jenž obsahuje `xStateListItem` všech úloh, které jsou momentálně blokované a seřazené podle času odblokování. Tudíž `pxDelayedTaskList` má vždy na první pozici
`xStateListItem` úlohy, které budou v nejbližší době odblokována. `vListInsert()` vkládá položky do listu tak, aby zachovala jejich řazení.

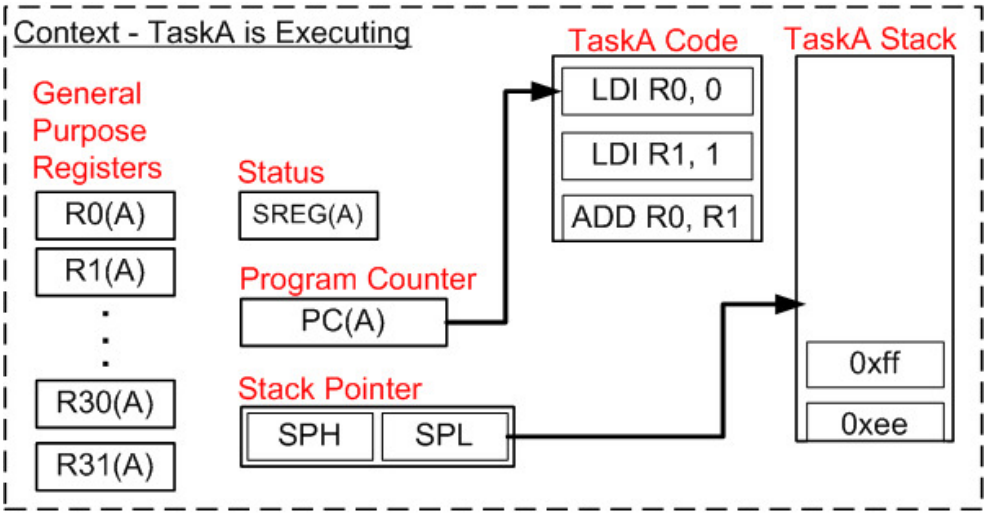
```
/* If the task entering the blocked state was placed at the
 * head of the list of blocked tasks then xNextTaskUnblockTime
 * needs to be updated too. */
if( xTimeToWake < xNextTaskUnblockTime )
{
    xNextTaskUnblockTime = xTimeToWake;
}
```

nakonec systémová proměnná `xNextTaskUnblockTime` obsahuje hodnotu času, kdy bude zavolána metoda pro odblokování procesu, pokud je potřeba. V tento okamžik je potřeba definovat přepínání kontextu. Hardwarově specifická funkce `portYIELD_WITHIN_API()` je volána, a úloha s nejvyšší prioritou z listu připravených úloh je povolána do běhu. V další sekci se zaměříme na to, jak je kontext přepínán.

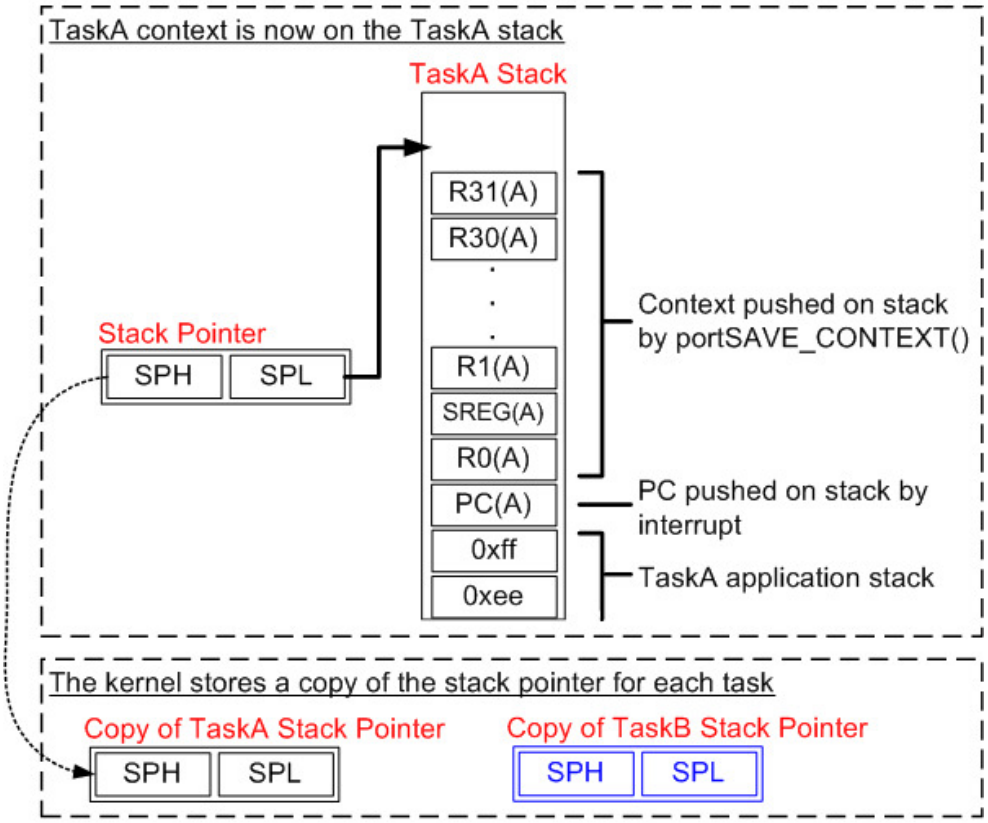
2.1.6 Přepínání kontextu

Přepínání kontextu musí probíhat transparentním způsobem, pokud jde o příslušné úkoly. Ve skutečnosti úkol neví, kdy bude pozastaven nebo znovu spuštěn systémem, může klidně pokračovat ve svém běhu, jako kdyby nedošlo k žádnému přepnutí kontextu. Je starostí operačního systému aby, že když běžící úloha je pozastavena, aby její obsah by měl být uložen v jejím zásobníku, a bylo připraveno její obnovení, když se úloha znovu spustí.

Obrázek 2.3 ukazuje, jak úkol běží. Registr ukazatele zásobníku (SP) ukazuje na zásobník běžící úloh, registr čítače programu (PC) ukazuje na další instrukci v kódu úlohy a registry CPU jsou používány úlohou. `portSAVE_CONTEXT()` je hardwarová funkce zodpovědná za ukládání obsahu úloh. Registry PC a SP spolu s dalšími registry jsou vloženy do zásobníku úlohy. Obrázek 2.4 ukazuje zásobník úlohy po uložení obsahu úlohy na zásobník. Jádrem si uloží kopii SP



Obrázek 2.3: Kontext spouštění úlohy [FreeRTOS.org]



Obrázek 2.4: Zásobník úlohy po uložení kontextu [FreeRTOS.org]

a operační systém si ukládá SP všech pozastavených úloh, aby si je mohly načíst při obnovení úkolů.

Kontext úlohy je obnoven funkcí `portRESTORE_CONTEXT()`, jádro pak načte SP úlohy, která měla v sobě záznam o probuzení, následně je obsah uložené úlohy vložen zpět do správných registrů procesorů.

2.1.7 Systém tiků

Už jsme viděli, že když je volána funkce `xTaskDelayUntil()`, volající úloha určí čas, ve kterém vyžaduje své probuzení. FreeRTOS měří čas pomocí systémové proměnné `xTickCount`. Přerušení tikem aktivuje Interrupt Service Routine (ISR), která inkrementuje proměnou `xTickCount` s přísnou časovou přesností, což umožňuje systému reálného času měřit čas při zvolené frekvenci přerušení. Pokaždé, když se inkrementuje proměnná `xTickCount`, operační systém musí zkontrolovat, zda je čas probudit úkol. Je možné, že úkol probuzený během ISR bude mít vyšší prioritu než priorita přerušeného úkolu. Pokud k tomu dojde, ISR by měl vrátit nově probuzenou úlohu. Když jsou úlohy vráceny tímto způsobem, tam můžeme o systému říct, že systém je nutně preemptivní. Níže bude popsána funkce ISR ze souboru port.c:

```
static void vPortSystemTickHandler( int sig )
{
    Thread_t *pxThreadToSuspend;
    Thread_t *pxThreadToResume;
    uint64_t xExpectedTicks;
    /* Signals are blocked in this signal handler. */
    uxCriticalNesting++;
#if ( configUSE_PREEMPTION == 1 )
    pxThreadToSuspend = prvGetThreadFromTask(
                            xTaskGetCurrentTaskHandle() );
#endif
/* Tick Increment, accounting for any lost signals or drift in
 * the timer. */
    xTaskIncrementTick();
#if ( configUSE_PREEMPTION == 1 )
    /* Select Next Task. */
    vTaskSwitchContext();
    pxThreadToResume = prvGetThreadFromTask(
                            xTaskGetCurrentTaskHandle() );
    prvSwitchThread(pxThreadToResume, pxThreadToSuspend);
#endif
    uxCriticalNesting--;
}
```

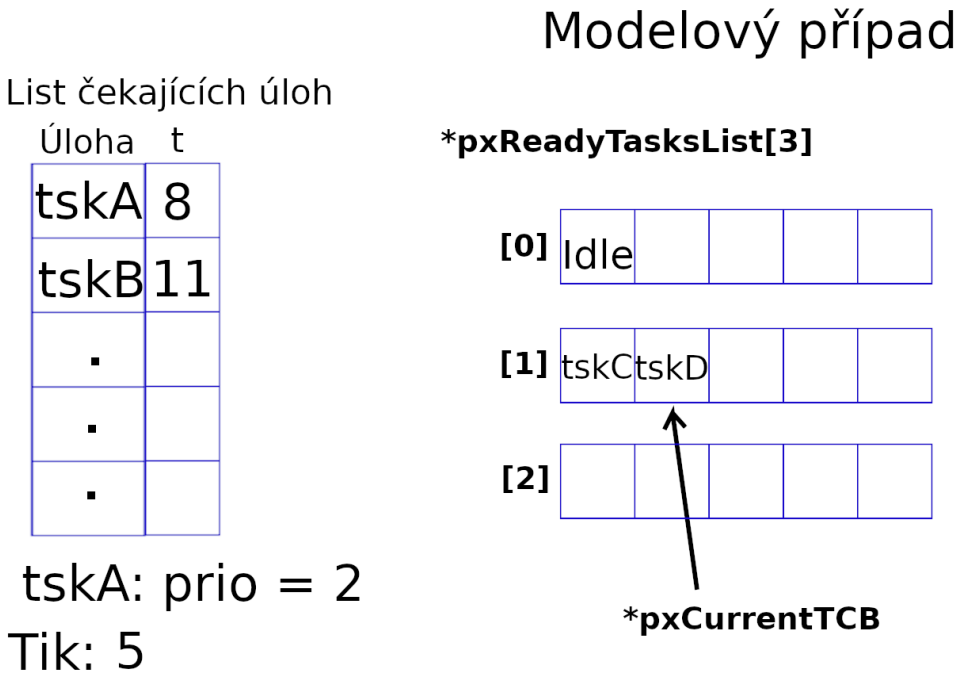
první věc, kterou `vPortSystemTickHandler()` udělá, je to, že si uloží obsah metodou `pxThreadToSuspend = prvGetThreadFromTask( )`, pak dvě hardwarově nezávislé funkce jsou volány.

- `xTaskIncrementTick()` inkrementuje systémovou proměnnou `xTickCount` a zkontroluje, jestli je čas úlohu opět probudit ze stavu **Blokovaná** do stavu **Připravené**. Pokud ano, pak úloha je odebrána z `pxDelayedTaskList` a je přesunuta do patřičného `pxReadyTasksLists[]`. Funkce vrátí `pdTRUE`, pokud stejné úlohy jsou probuzeny, aby mohly dát ISR vědět, jestli je potřeba přepínání kontextu.
- `vTaskSwitchContext()` nastaví `*pxCurrentTCB` do TCB úlohy s největší prioritou uloženou v `pxReadyTasksLists[]`

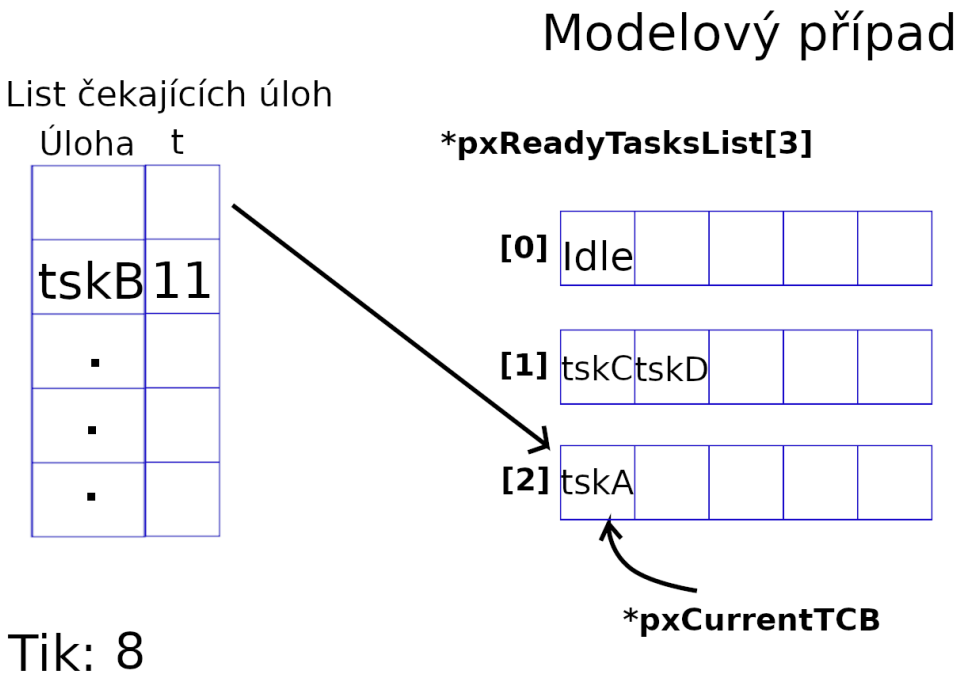
Nakonec funkce:
`pxThreadToResume = prvGetThreadFromTask( xTaskGetCurrentTaskHandle() );`
nám obnoví obsah ze zásobníku úlohy, na kterou ukazuje `*pxCurrentTCB`.

2.1.8 Ukázka plánování

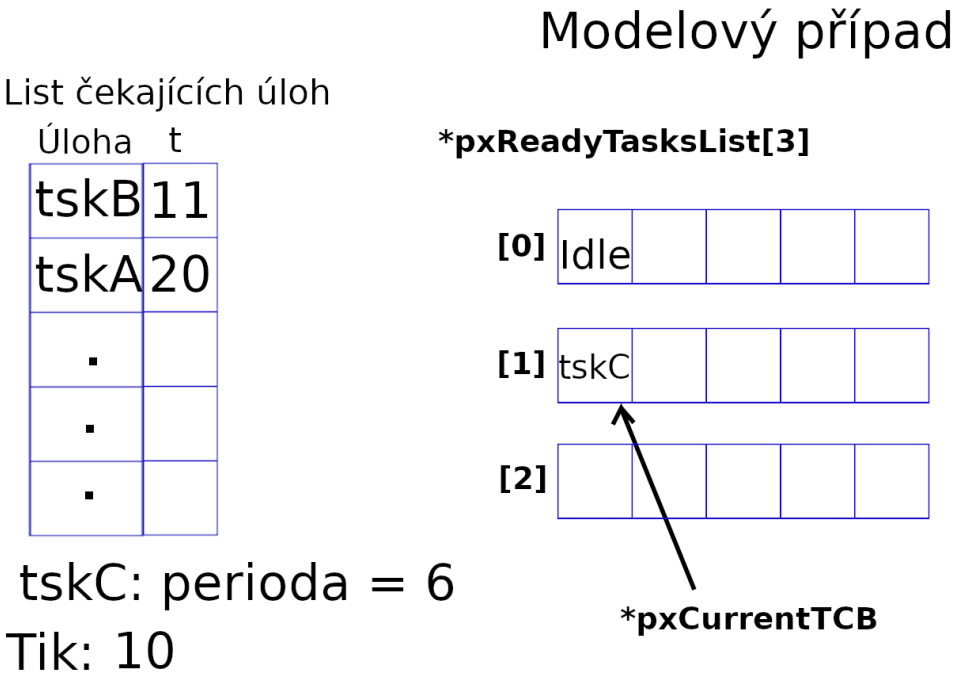
V této sekci je ukázán příklad preemptivního plánování ve FreeRTOS. První obrázek 2.5 nám zadává modelovou situaci k představě. Mějme tedy již běžící, kde jsou čtyři námi zadefinované úlohy, z toho pouze tři jsou periodické a jedna (**tskD**) je spouštěna vnější událostí. Systém je momentálně v pátém tiku a provádí se **tskD**, úlohy **tskA** a **tskB** čekají na svůj čas vzbuzení. V okamžik, kdy přijde systém k tiku 8 (obrázek 2.6), úloha **tskA** se vloží do prioritní fronty `pxReadyTasksList[2]` a zároveň je odebrán z listu čekajících úloh. Vložením úlohy **tskA** do `pxReadyTasksList[2]` se docílí toho, že úloha **tskD** čeká, než se dokončí všechny úlohy v `pxReadyTasksList[2]`. Tímto způsobem se FreeRTOS dosáhl toho, že je preemptivní. V tiku deset (obrázek 2.7) se již zvládla zpracovat úloha **tskD** a **tskA**, ta se však, na rozdíl od úlohy **tskD**, která není periodická, vrátila do listu čekajících úloh, kde čeká na svůj nový čas spuštění. V tomto okamžiku se zpracovává úloha **tskC**. Po zpracování všech úloh v obrázku 2.8 se v tiku 14 nacházíme na malý moment situaci, kde žádná úloha není volána a náš ukazatel `pxCurrentTCB` zpracovává pouze dummy úlohy **IDLE**, jelikož v listu nejnižší priority je jediná. Zde je názorně vidět, že o plánovači dodávané dodavatelem je drobná forma RMS, jelikož jsou nastaveny priority staticky na začátku plánování a systém je preemptivní, pouze systém nehlídá ani nedefinuje deadliny úloh.



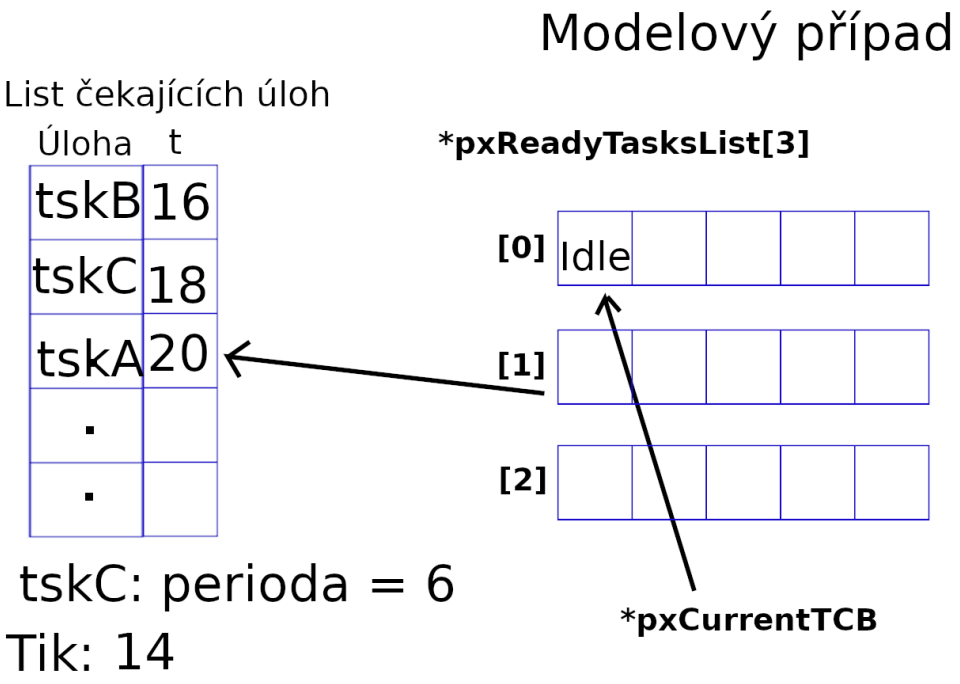
Obrázek 2.5: Příklad interakce přepínání obsahu úloh



Obrázek 2.6: Příklad interakce přepínání obsahu úloh



Obrázek 2.7: Příklad interakce přepínání obsahu úloh



Obrázek 2.8: Příklad interakce přepínání obsahu úloh

2.2 EDF

První plánovač, který implementujeme, je založen na algoritmu Earliest Deadline First (EDF). EDF přijímá zásady preemptivního plánování založené na dynamických prioritách, což znamená, že priorita úkolu se může během jeho spuštění změnit a zpracovávání kteréhokoli úkolu může být přerušeno na žádost o provedení úkolu s vyšší prioritou.

Algoritmus přiřazuje priority úkolům jednoduchým způsobem: priorita úkolu je inverzně úměrná jeho absolutnímu deadlinu. Jinými slovy, nejvyšší prioritu má úloha, která má v aktuální okamžik deadline nejdříve. V případě dvou nebo více úkolů se stejným deadlinem a zároveň při splnění podmínky pro úspěšné naplánování se mezi nimi vybírá náhodně.

Algoritmus je vhodný pro práci v prostředí, kde platí tyto předpoklady (jsou stejné jako u RMS):

1. Požadavky pro úlohy s hard deadliny jsou periodické.
2. Úlohy jsou nezávislé (žádné vzájemné vyloučení nebo precedenční omezení).
3. Deadline intervaly (relativní deadliny) D_i jsou stejné jako periody p_i.
4. Doba výpočtu c_i je předem známá a konstantní.
5. Dobu pro přepnutí kontextu je možné zanedbat.

Díky těmto předpokladům můžeme úkol charakterizovat pouze pomocí 2 vlastností, a to periodou p_i a dobou výpočtu c_i. Použijeme $T_1, T_2, \ldots, T_n$ pro označení n periodických úloh (Tásků), k nim přiřazené jejich periody $p_1, p_2, \ldots, p_n$ a doby výpočtů $c_1, c_2, \ldots, c_n$. Takže úloha T_i je spouštěna s periodou p_i a měla by zkonzumovat c_i časových jednotek před tím, než dojde ke svému deadlinu, tudíž při ukončení úlohy by mělo platit ($c_i \leq_i$).

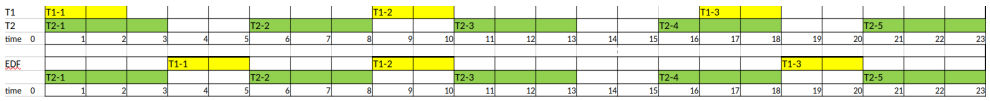
[4]Proto si zadefinujeme postačující podmínku pro plánovatelnost úloh takto:

Věta 1 *Úlohy plánované pomocí algoritmu jsou plánovatelné pokud platí podmínka:*

$$\mu = \sum_{i=1}^{N} \frac{c_i}{p_i} \leq 1$$

Nuže pojďme uvažovat, že máme následující dvě úlohy: $A : c_1 = 2, p_1 = 8$ a $B : c_2 = 3, p_1 = 5$

$$\mu = \frac{2}{8} + \frac{3}{5} = \frac{34}{40} = 0.85 \leq 1$$



Obrázek 2.9: Plánování úloh v EDF [BI-SRC]

2.2.1 Ukázka plánování

s odkazem na větu 1, EDF plánovací algoritmu je schopen obsloužit všechny úkoly, aniž by byly porušeny deadliny některé z úloh. Obrázek 2.9 nám popisuje, jak úlohy $T1 = A$ a $T2 = B$ jsou plánovány. V čase $t = 3$ úloha B dokončí svojí rutinu a je naplánována úloha A, a tak to pokračuje dál a dál...V čase $t = 3$ se rozhoduje mezi pokračováním v úloze B, nebo jestli zahájit úlohu A, protože úloha B má v daný čas větší prioritu, tak se pokračuje v běhu úlohy B. Takto se pokračuje dál až do nekonečna. Priorita by se zde počítala jako $P_i = p_i$. To je vše k tomto příkladu z přednášek BI-SRC [5]

2.3 LLF

Least Laxity First algoritmus je, stejně jako algoritmu EDF, určen pro jednoprocesorové zařízení, i když se dají použít i na multiprocesorových systémech, přesto není jejich použit mimo jednoprocesorové zařízení optimální. V tomto algoritmu se přiřazuje priorita na základě nejmenší laxity. Laxitu si můžeme představit jako volnost v tom, na jak dlouho se může úloha odložit, aniž bychom porušili její zpracování v rámci periody, čili [6]:

$$L_i(t) = D_i(t) - C_i(t)$$

, kde $L_i(t)$ je laxita procesu i v čase t, $D_i(t)$ je kolik času zbývá do deadlinu procesu i v čase t, $C_i(t)$ je čas, který už se na úloze ještě potřebuje odpracovat v čase t nebo také WCET (Worst Case Execution Time).

WCET $C_i(t)$ lze také znázornit jako $C_i(t) = t_i + c_i$ a po dosazení nám vznikne formule:

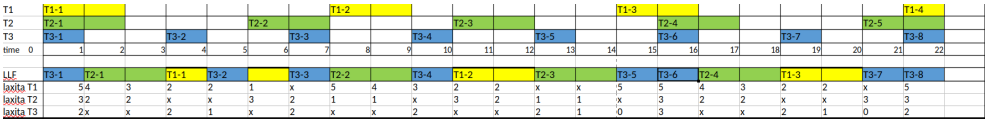
$$L_i(t) = D_i(t) - (t_i + c_i)$$

.

2.3.1 Ukázka plánování

Zde si ukážeme na typicky školním příkladu, jak algoritmus plánování LLF se vypořádává s prioritami. Tudíž uvažujme následující příklad tří úloh:

- Úloha **A (T1)**, kde čas zpracování je $c_A = 2$ a perioda $p_A = 7$
- Úloha **B (T2)**, kde čas zpracování je $c_B = 2$ a perioda $p_A = 5$



Obrázek 2.10: Plánování úloh v LLF [BI-SRC]

- Úloha **C (T3)**, kde čas zpracování je $c_C = 1$ a perioda $p_A = 3$

Tak v obrázku 2.10 je znázorněné, jak se chovají jednotlivé úlohy, a pak v řádku LLF reprezentující jednoprocesorové vlákno v našem FreeRTOS. Dále je spočítaná laxita pro jednotlivé úlohy v daný časový okamžik, kde písmenem x znázorňuji spící úkol, čekající na zpracování v další periodě. Jako reprezentativní a vysvětlující příklad si zvolím čas 16 a ten tady podrobněji projdu.

V tomto okamžiku se k nám po spánku vrátily úlohy B a C, zde tyto dvě úlohy mají laxitu stejnou s laxitou, kterou měli na začátku programu, čili laxita C je $L_C = 19 - (16 + 1)$, a to $L_C = 2$, stejným způsobem se dostanem k výsledku úlohy B a ta nám vyjde $L_B = 3$. Teď spočítáme, jak vypadá laxita úlohy A, ta na rozdíl od ostatních nezačala v daný okamžik a v čase 15 se nedokázala naplánovat, ale princip počítání laxity se nemění, proto se na ni pojďme podívat. Laxita úlohy $L_A = 22-(16+2)$, a tak dostáváme laxitu úlohy A. Nyní porovnáme laxity a získáme výtěze našeho plánování, tím se stává úloha C s laxitou 2. V časovém okamžiku 16 se naplánuje úloha C a stejným způsobem se získává naplánovaná úloha i v následujících krocích.

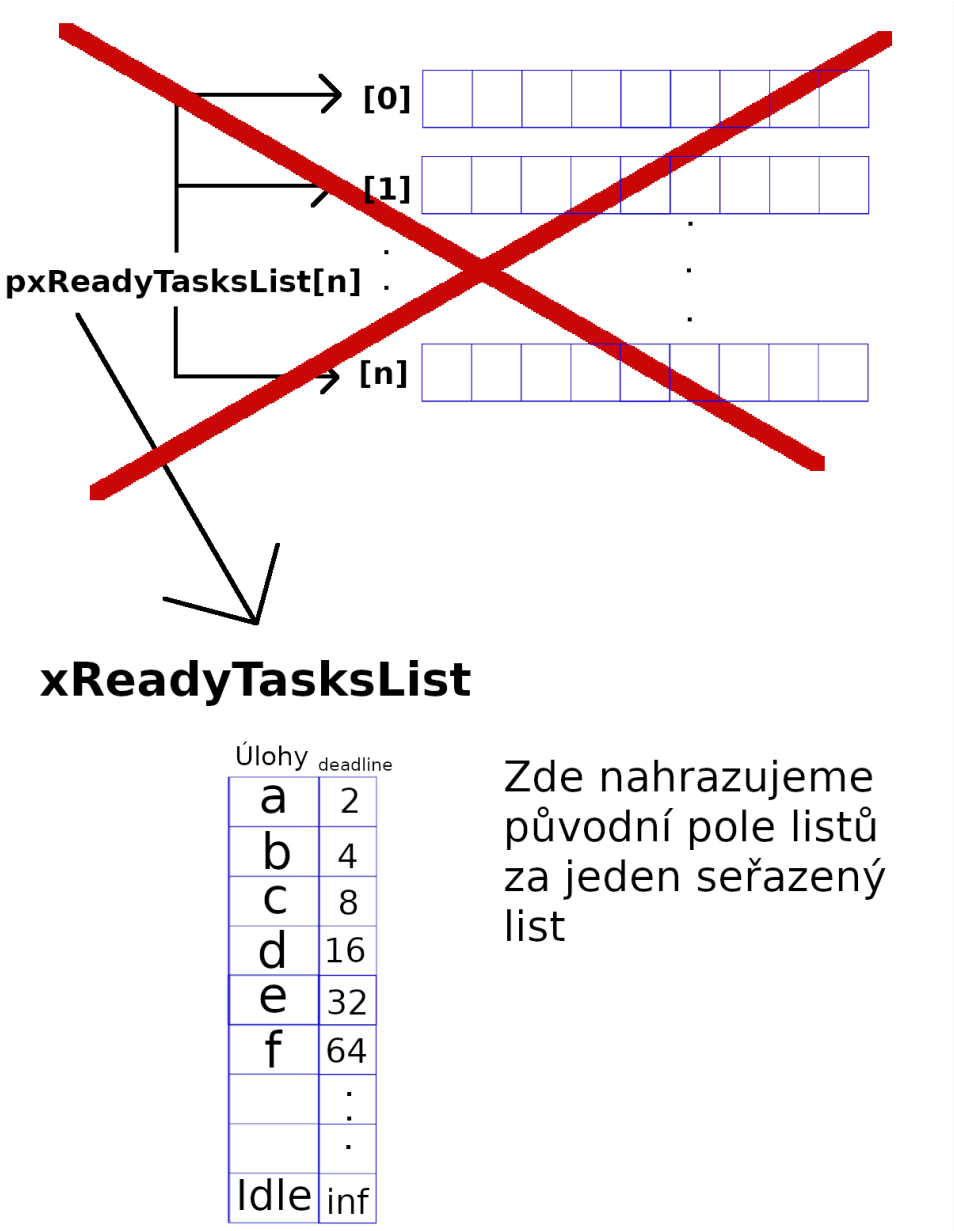
Kapitola 3

Implementace

V této kapitole se zaměříme na implementaci testovacích souborů, modifikací již existujících metod ze souboru *task.c*, o kterých již bylo zmíněno v předchozích kapitolách. Na rozdíl od tradiční implementace na přípravek, bude se implementovat vše nad POSIXovým portem dodávaný FreeRTOS.org z třetí strany. Díky POSIXovému portu se nemusíme limitovat při implementaci pouze na porty přípravků a operačním systémem Windows. Aplikace je tedy jednoduše spustitelná a kompilovatelná na linuxových operačních systémech. Primárně je tato práce otestována na operačním systému Ubuntu 20.04 LTS a tudíž bezproblémový provoz na jiných operačních systémech není zaručen.

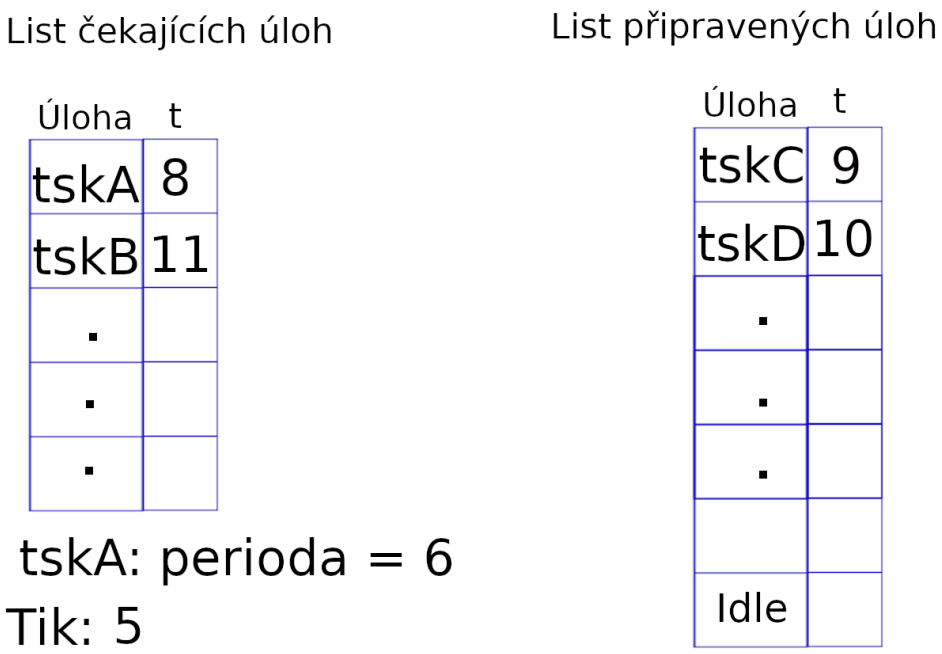
3.1 EDF

V předchozí kapitole bylo zmíněno, že FreeRTOS používá plánovač založený na statických prioritách. Cíl v této sekci je jasný, a to popsat, jak se dá implementovat EDF plánovací algoritmus do FreeRTOSu používajíce již existující struktury, které nám FreeRTOS nabízí, či případně doplnit o své struktury. Nápad je takový, že vytvoříme nový list `xReadyTaskListEDF` (obrázek 3.1), ve kterém budeme spravovat priority našich úloh. Tento nový list bude obsahovat úlohy seřazené podle naší dynamické priority vzestupně, kde pozice v tomto listu bude reprezentovat prioritu a na první pozici bude naše momentálně spouštěná úloha. Zbytek architektury FreeRTOSu je zachován a místy se vyskytují drobné úpravy. V následujících částí této sekce budou ukázány části kódu této práce. Pro naši práci si můžeme stále předpokládat platnost našich předpokladů: Veškeré zde zmíněné změny v kódu budou referovat na obsah souboru *task.c*, jelikož definice a struktury přenosného plánovače jsou zadefinovány zde. Abychom si dodrželi konzistenci s souborem *task.c*, bude veškerý kód obalen v prostředí podmíněného překlad, kde budeme kontrolovat nastavení definice `configUSE_EDF_SCHEDULER`. Tato definice je vložena do souboru *FreeRTOSConfig.h*. Když tato definice bude nastavena, pak FreeRTOS bude

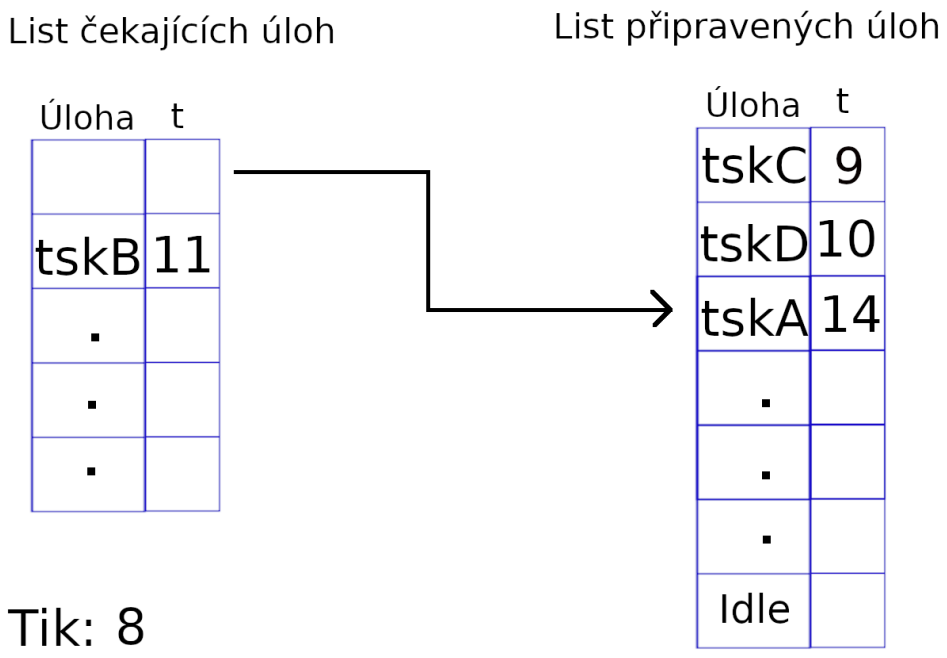


Obrázek 3.1: Nový list obsahuje úlohy seřazené vzestupně podle vzrůstajícího deadlinu. Na první pozici se nachází úloha s nejbližším deadlinem.

- Požadavky pro úlohy s hard deadliny jsou periodické.
- Úlohy jsou nezávislé (žádné vzájemné vyloučení nebo precedenční omezení).
- Deadline intervaly (relativní deadliny) D_i jsou stejné jako periody p_i.
- Doba výpočtu c_i je předem známá a konstantní.
- Dobu pro přepnutí kontextu je možné zanedbat.



Obrázek 3.2: Zde zobrazuji běžící úlohu C a její deadline v tiku 9



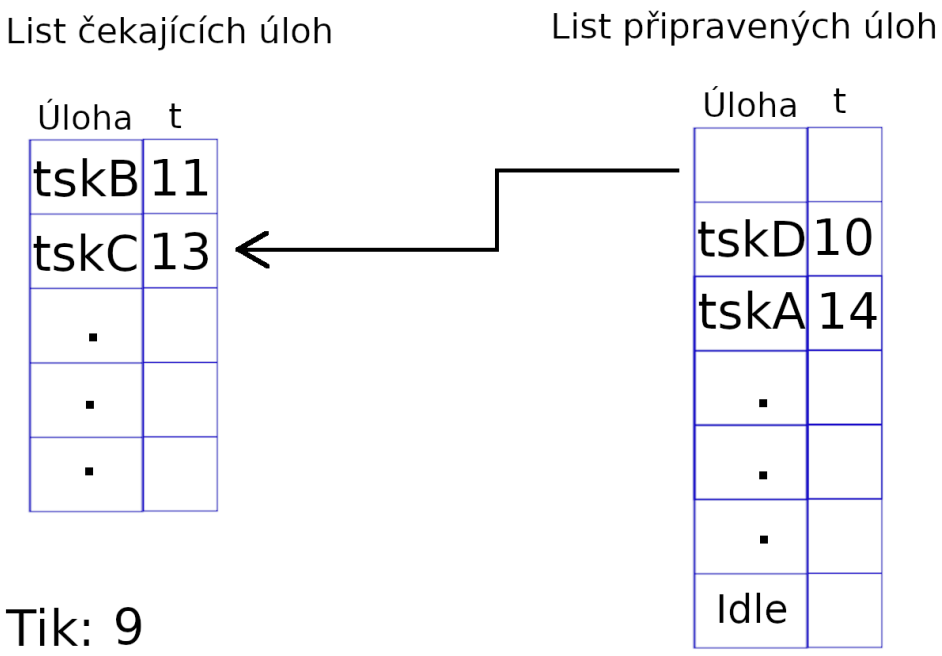
Obrázek 3.3: Zde mechanizmus vzbudí úlohu A a vloží ji do listu připravených úloh. Zároveň ji odebereme z listu čekajících úloh. Do listu připravených úloh ji vložíme na místo, tak aby její deadline byl seřazen vzestupně vůči ostatním úlohám. Deadline byl vypočítán jako $D_i = t + p_i$

používat plánovač EDF, v opačném případě bude pracovat jak v originálním stavu.

Nejdříve si zadefinujeme naší novou předpokládanou strukturu listu `xReadyTasksListEDF`

```
#if ( configUSE_EDF_SCHEDULER == 1 )
    PRIVILEGED_DATA static List_t xReadyTasksListEDF;
    /*< Ready tasks in order by their deadline */
#endif
```

Pak si přidáme do metody `prvInitialiseTaskLists()`, přidáme kód pro inicializaci našeho nového listu. Metodu modifikujeme takto:



Obrázek 3.4: Úloha C ukončila svůj běh a přesunula se do listu čekajících úloh. Úloha D je nyní na začátku listu a bude spuštěna.

```
static void prvInitialiseTaskLists( void )
{
    ...
    // < initialise task ready list
    #if ( configUSE_EDF_SCHEDULER == 1)
        {
            vListInitialise( &xReadyTasksListEDF );
        }
    #endif

    ...

}
```

Další změnu provedeme v makru `prvAddTaskToReadyList()`, která slouží pro přidání úlohy do listu připravených úloh. Změnu provedeme následovně:

```
/*
 * There shoudl be all of the modification of how to add task
 * to its list
 */
#if ( configUSE_EDF_SCHEDULER == 1 )
    #define prvAddTaskToReadyList( pxTCB )                  \
        listSET_LIST_ITEM_VALUE( &(                         \
            ( pxTCB )->xStateListItem ),                    \
            ( pxTCB )->xTaskPeriod + xTickCount);           \
        vListInsert( &( xReadyTasksListEDF ),               \
            &( ( pxTCB )->xStateListItem ) );
#else

    /* ORIGINAL CONTENT
    * Place the task represented by pxTCB into the appropriate
    * ready list for the task.  It is inserted  at the end of
    * the list.
    */
    #define prvAddTaskToReadyList( pxTCB )                  \
            traceMOVED_TASK_TO_READY_STATE( pxTCB );        \
        taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority ); \
        vListInsertEnd( &( pxReadyTasksLists[               \
            ( pxTCB )->uxPriority ] ),                      \
            &( ( pxTCB )->xStateListItem ) );               \
        tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
#endif
```

Metoda `vListInsert()` je volána pro vložení ukazatele TCB do `xReadyTasksListEDF`. Tato metoda vkládá předměty do list tak, aby list byl seřazen podle hodnoty předané druhým parametrem, v našem případě `( pxTCB )->xStateListItem`. Tudíž zde můžeme s jistotou předpokládat, že do této proměnné budeme ukládat deadline úlohy. Časová složitost vkládání úloh do listu je lineární a jednoduchý, proto je vhodný pro zařízení s malou pamětí.

Jak jsme si zobrazili v obrázcích 3.2, 3.3 a 3.4, když přesouváme úlohy, tak potřebujeme vědět čas deadlinu, abychom ji zvládli vložit na správnou pozici. Naštěstí deadline se dá jednoduše spočítat jako $D_i = t + p_i$, tudíž budeme potřebovat, aby úlohy si ukládali údaj o své periodě. To docílíme tak, že do TBC vložíme následující kód:

```
/*  The old naming convention is used to
    prevent breaking kernel aware debuggers. */
typedef struct tskTaskControlBlock
```

```
{
    ...
    #if ( configUSE_EDF_SCHEDULER == 1 )
        TickType_t xTaskPeriod;
    #endif
    ...
} tskTCB;
```

Pochopitelně abychom tento údaj mohli využít, budeme potřebovat vytvořit novou metodu `xTaskPeriodicCreate()`. Toto je však zkopírovaná a trochu modifikovaná metoda `xTaskCreate()`, která navíc sbírá informaci o periodě.

```
#if ( configUSE_EDF_SCHEDULER == 1 )

    BaseType_t xTaskPeriodicCreate( TaskFunction_t pxTaskCode,
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask,
                        TickType_t period ){

    ...


    if( pxNewTCB != NULL ) {
            pxNewTCB->xTaskPeriod = period;
            listSET_LIST_ITEM_VALUE( &( ( pxNewTCB )->xStateListItem ),
                            ( pxNewTCB )->xTaskPeriod + xTickCount);
            prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t )
                usStackDepth, pvParameters, uxPriority,
                pxCreatedTask, pxNewTCB, NULL );
            prvAddNewTaskToReadyList( pxNewTCB );
            xReturn = pdPASS;
        } else {
            xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
        }

        return xReturn;
    }

#endif
```

Řízení speciální spící úlohy také neunikne modifikaci. Inicializace spící úlohy je prováděna v metodě `vTaskStartScheduler()`, která zahajuje celý proces řízení reálného času a inicializuje veškeré nutné struktury. Protože FreeRTOS je velmi specifický v tom, jak se mají úlohy spouštět, tak správné ošetření spící úlohy je nezbytné. V klasickém případě je ošetření spící úlohy jednoduché, prostě se jí přiřadí nejnižší priorita. Tím se zajišťuje, že úloha je volána, pouze když není žádná jiná úloha, která by chtěla pracovat. S naším EDF plánovačem můžeme nasimulovat nízkou prioritu tím, že dáme deadline téměř nekonečný. Metoda `vTaskStartScheduler()` inicializuje naši spící úlohu a vloží ji do `xReadyTasksListEDF`.

Metoda je definována takto:

```
#if ( configUSE_EDF_SCHEDULER == 1 )
{
    TickType_t initIDLEPeriod = 0xfffffff;
    xReturn = xTaskPeriodicCreate( prvIdleTask,
                                   configIDLE_TASK_NAME,
                                   configMINIMAL_STACK_SIZE,
                                   ( void* ) NULL,
                                   ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ),
                                   &xIdleTaskHandle,
                                   initIDLEPeriod );
}
#else
{
    xReturn = xTaskCreate( prvIdleTask,
                  configIDLE_TASK_NAME,
                  configMINIMAL_STACK_SIZE,
                  ( void * ) NULL,
                  ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ),
                  /* In effect ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ),
                      but tskIDLE_PRIORITY is zero. */
                  &xIdleTaskHandle ); /*lint !e961 MISRA exception,
                      justified as it is not a redundant explicit cast
                      to all supported compilers. */
}
#endif
```

Spící úlohu inicializujeme s periodou `TickType_t initIDLEPeriod = 0xfffffff;`. Předpokládám, že pro účely výuky nikdo nebude čekat 0xfffffff sekund, aby přetekly ostatní úlohy danou spící úlohu, a systém tak zůstal na nějakou chvíli ve stavu spánku. Takto zaručíme, že spící úloha bude vždy poslední.

Poslední úpravy se budou týkat mechanismu přepínaní kontextů úloh. Pokaždé, když úloha je pozastavena nebo pozastavená úloha, s vyšší prio-

ritou než momentálně běžící, se vzbudí, dochází k přepnutí kontextu. Metoda `vTaskSwitchContext()` je tu od toho, aby patřičně aktualizovala ukazatel `*pxCurrentTCB` na nově spuštěnou úlohu:

```
        void vTaskSwitchContext( void )
{


    ...


        #if ( configUSE_EDF_SCHEDULER == 1 )
        {
            pxCurrentTCB = ( TCB_t * )
                    listGET_OWNER_OF_HEAD_ENTRY( &(
                            xReadyTasksListEDF ) );
        }
        {
            /*lint !e9079 void * is used as this macro is used
                with timers and co-routines too.  Alignment is
                known to be fine as the type of the pointer
                stored and retrieved is the same. */
            taskSELECT_HIGHEST_PRIORITY_TASK();
        }


    ...


}
```

Metoda `taskSELECT_HIGHEST_PRIORITY_TASK()` je nahrazena abychom docílili toho, že se vždy vybere úloha na prvním místě v listu `xReadyTasksListEDF`.

Nyní máme všechny potřebné části k provozuschopnosti plánovače.

3.2 LLF

Po implementaci plánovacího algoritmu EDF, je na čase začít implementaci algoritmu Least laxity first. I tento algoritmus jako EDF využívá dynamické priority. Cíl této sekce je opět jasný, a to popsat, jak se dá implementovat LLF plánovací algoritmus do FreeRTOSu používající již existující struktury, které nám FreeRTOS nabízí, či případně doplnit o své struktury. Nápad je takový, že vytvoříme tentokrát dva nové listy `xReadyTaskListLLF`

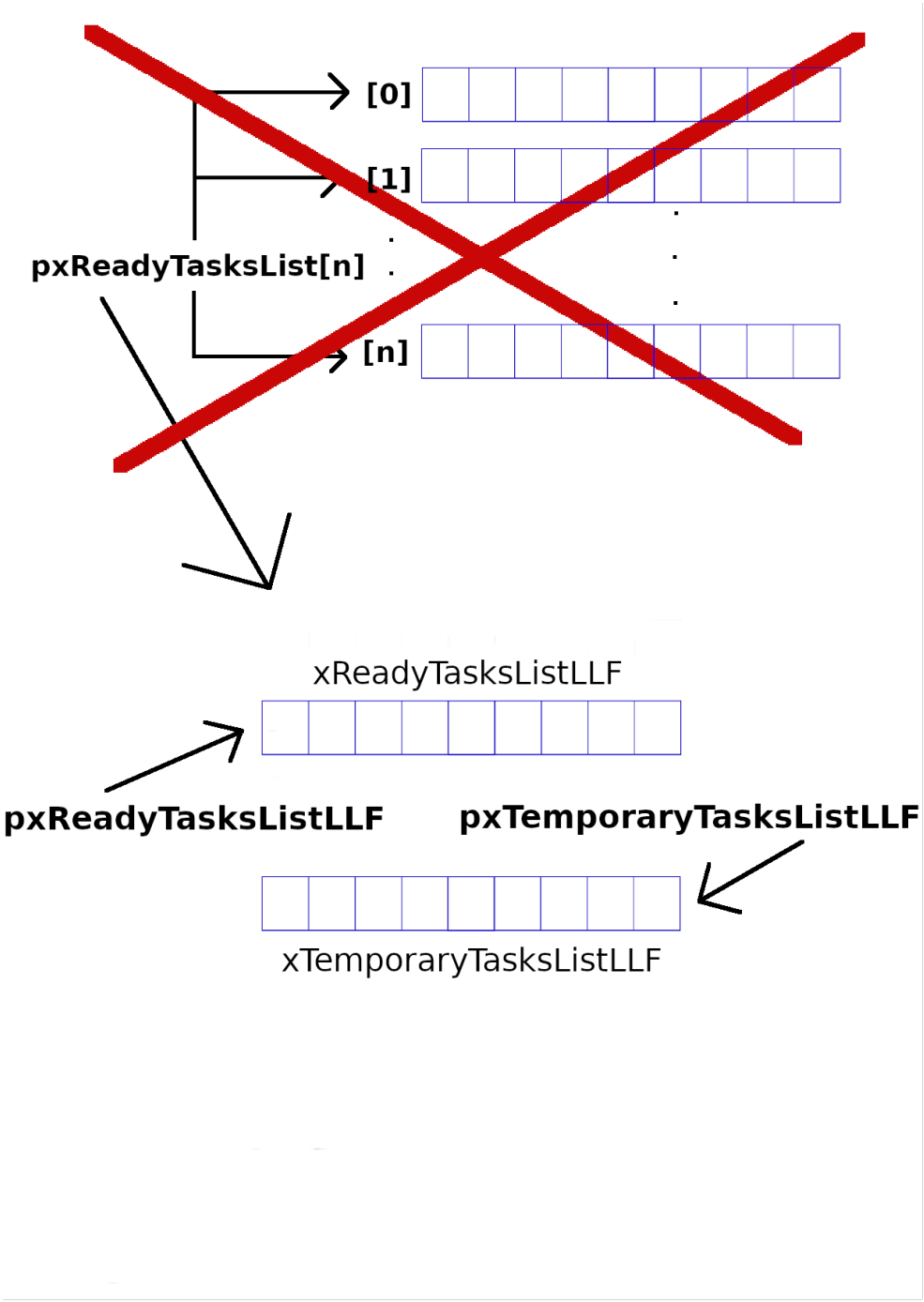
a `xTemporaryTaskListLLF` (obrázek 3.5). V listu `xReadyTaskListLLF` budeme podobně jako u EDF spravovat priority našich úloh. Tento nový list bude obsahovat úlohy seřazené podle naší dynamické priority vzestupně, kde pozice v tomto listu bude reprezentovat prioritu podle nejmenší laxity, a opět bude v první pozici naše momentálně spouštěná úloha. Druhý list nám bude sloužit jako způsob, jak rychle a bez velké režie budeme listy řadit. Zbytek architektury FreeRTOSu je zachován a místy se vyskytují drobné úpravy. Pro nastavení LLF se zadefinuje v *FreeRTOSConfig.h* definice `configUSE_LLF_SCHEDULER`. Ta nám bude určovat, jestli bude použit plánovač LLF. Chování programu není definováno pokud bude LLF i EDF plánovač zapnut najednou. Začneme podobně jako u EDF a to tím, že si zadefinujem naše nové předpokládané struktury listů `xReadyTasksListLLF` a `xTemporaryTasksListLLF` a k nim jejich příslušné ukazatele `pxReadyTasksListLLF` a `pxTemporaryTasksListLLF`.

```
#if ( configUSE_LLF_SCHEDULER == 1 )
    /*< Ready tasks in order by their deadline */
    PRIVILEGED_DATA static List_t xReadyTasksListLLF;
    PRIVILEGED_DATA static List_t xTemporaryTasksListLLF;
    /* Necessary for easy swapping */
    PRIVILEGED_DATA static List_t * volatile pxReadyTasksListLLF;
    PRIVILEGED_DATA static List_t * volatile pxTemporaryTasksListLLF;
#endif
```

Dále v metodě `prvInitialiseTaskLists()` si inicializujeme naše nové listy následujícím způsobem:

```
#if ( configUSE_LLF_SCHEDULER == 1) // < initialise task ready list
    {
        vListInitialise( &xReadyTasksListLLF );
        vListInitialise( &xTemporaryTasksListLLF );
        pxReadyTasksListLLF = &xReadyTasksListLLF;
        pxTemporaryTasksListLLF = &xTemporaryTasksListLLF;
    }
#endif
```

Tentokrát úprava metody `prvAddTaskToReadyList()` bude vypadat dosti podobně úpravě v EDF plánovači. Zde jako hlavní list je `xReadyTasksListLLF` ve kterém jsou uloženy úlohy řazeny vzestupně podle laxity. Tato funkce má lineární složitost.



Obrázek 3.5: Nový list xReadyTasksListsEDF obsahuje úlohy seřazené vzestupně podle vzrůstající laxity. Na první pozici se nachází úloha s nejnižším laxitou. Druhý list má sloužit pouze řazení úloh. Každý list má přidružený ukazatel pro swapování obsahu.

```
#if ( configUSE_LLF_SCHEDULER == 1)
    #define prvAddTaskToReadyList( pxTCB ) \
        vListInsert( pxReadyTasksListLLF , &( \
                    ( pxTCB )->xStateListItem ) )
#else

    /* ORIGINAL CONTENT
    * Place the task represented by pxTCB into the appropriate
    * ready list for the task.  It is inserted at the end of
    * the list.
    */
    #define prvAddTaskToReadyList( pxTCB )                      \
        traceMOVED_TASK_TO_READY_STATE( pxTCB );                \
        taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );     \
        vListInsertEnd( &( pxReadyTasksLists[                   \
            ( pxTCB )->uxPriority ] ), &(                       \
            ( pxTCB )->xStateListItem ) );                      \
        tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
#endif
```

Následuje deklarování nových proměnných do `prvTaskControlBlock`, které jsou potřeba nově vědět.

```
#if ( configUSE_LLF_SCHEDULER == 1 )
    TickType_t xTaskCapacity;
    volatile TickType_t xTaskExecTime;
#endif
```

`TickType_t xTaskCapacity` nám nově bude určovat dobu nutnou pro výpočet úlohy a `volatile TickType_t xTaskExecTime` proměnná ukládá stav o tom, kolik je potřeba ještě provést výpočtů v dané periodě. Nyní když máme tyto dvě nové proměnné deklarované, je nutno upravit `xTaskPeriodicCreate()`, aby s nimi bylo možné pracovat.

```
BaseType_t xTaskPeriodicCreate( TaskFunction_t pxTaskCode,
                            const char * const pcName,
                            const configSTACK_DEPTH_TYPE usStackDepth,
                            void * const pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t * const pxCreatedTask,
                            TickType_t period,
                            TickType_t capacity){
```

```
        ...


        pxNewTCB->xTaskExecTime = 0;
        pxNewTCB->xTaskPeriod = period;
        pxNewTCB->xTaskCapacity = capacity;
        listSET_LIST_ITEM_VALUE( &( ( pxNewTCB )->xStateListItem ),
                              ( pxNewTCB )->xTaskPeriod - (
                              xTickCount + ( pxNewTCB )->xTaskCapacity ) ;
        rvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t )
                            usStackDepth, pvParameters, xPriority,
                            pxCreatedTask, pxNewTCB, NULL );
        prvAddNewTaskToReadyList( pxNewTCB );
        xReturn = pdPASS;
    } else {
        xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
    }
    return xReturn;
}
```

Dále upravíme probouzení úloh v metodě `xTaskIncrementTick()` a připravíme prostor na umístění kódu řazení úloh. Vše děláme v souladu s definicí laxity $L_i = D_i - (t + c'_i)$.

```
BaseType_t xTaskIncrementTick( void )
{
        ...

        #if ( configUSE_LLF_SCHEDULER == 1 )
            vResortTasksLLF();
            const TickType_t xTemp = (pxCurrentTCB->xTaskExecTime
                            + 1) % (pxCurrentTCB->xTaskCapacity + 1);
            if( listGET_LIST_ITEM_VALUE( &(
                            ( pxCurrentTCB )->xStateListItem ) ) )
                            pxCurrentTCB->xTaskExecTime = xTemp;
            if(pxCurrentTCB !=
                    (TCB_t *)listGET_OWNER_OF_HEAD_ENTRY(
                                    pxReadyTasksListLLF )){
                xSwitchRequired = pdTRUE;
            }
        #endif
```

```
        ...

        #if ( configUSE_LLF_SCHEDULER == 1 )
                        printf("Here: %ld\n",pxTCB->xTaskExecTime);
                        pxTCB->xTaskExecTime = 0;
                        printf("Here: %ld\n",pxTCB->xTaskExecTime);
                        listSET_LIST_ITEM_VALUE( &(
                                ( pxTCB )->xStateListItem ),
                                ( pxTCB )->xTaskPeriod - (
                                ( xTickCount % ( pxTCB )->xTaskPeriod )
                                + ( pxTCB )->xTaskCapacity ) );
        #endif

        ...

}
```

V následujícím kusu kódu je nově zadefinována funkce `void vResortTasksLLF()`. Tato funkce obsluhuje logiku výpočtu laxity v každém okamžiku, jelikož jsme limitováni FreeRTOSem v tom, jakou maximální dobu můžeme strávit výběrem úkolu, stává se tato funkce delikátním úkolem, jelikož i když to vypadá, že funkce by se dala napsat i úsporněji, její podoba v tomto tvaru je nezbytná. Nad důvodem proč se však v této práci nebudeme zabývat. Tato funkce má kvadratickou složitost.

```
void vResortTasksLLF(){
    List_t * listSwap;
    configASSERT( ( listLIST_IS_EMPTY( pxTemporaryTasksListLLF )
                                        == pdTRUE ) );
    while( listLIST_IS_EMPTY( pxReadyTasksListLLF ) == pdFALSE ){
        TCB_t * pxTCB =
                listGET_OWNER_OF_HEAD_ENTRY( pxReadyTasksListLLF );
        uxListRemove( &( ( pxTCB )->xStateListItem ) );
        const TickType_t xRuned = ( pxTCB )->xTaskCapacity -
                                  ( pxTCB )->xTaskExecTime;
        const TickType_t xTimed =
                    (xTickCount - 1) % ( pxTCB )->xTaskPeriod;
        listSET_LIST_ITEM_VALUE( &( ( pxTCB )->xStateListItem ),
                    ( pxTCB )->xTaskPeriod - ( xRuned + xTimed ) );
        vListInsert( pxTemporaryTasksListLLF, &(
                                    ( pxTCB )->xStateListItem ) );
    }
```

```
    listSwap = pxReadyTasksListLLF;
    pxReadyTasksListLLF = pxTemporaryTasksListLLF;
    pxTemporaryTasksListLLF = listSwap;
}
```

Jelikož toto byl náš poslední nezbytný úkol k funkčnosti plánovače LLF vrátíme se se k němu znovu až v sekci LLF kapitoly Testování.

Kapitola 4

Testování

V této kapitole se zaměříme na testování, porovnání jednotlivých algoritmů. Na následujících příkladech se budeme snažit porozumět praktickému použití plánovačů reálného času v reálném světě. Zkusíme porovnat použití algoritmů dynamických priorit s algoritmem statické priority. V této části se nebudeme zabývat ničím jiným než praktickými příklady, které byly použity na systému reálného času. Systém, na kterém pracujeme, je preemptivní, jinak by nemělo cenu řešit plánování, pokud chceme, aby úlohy opravdu splnili deadline.

4.1 Testování FreeRTOS plánovače

V této sekci si spustíme úlohu, kde oba `configUSE_EDF_SCHEDULER` a `configUSE_LLF_SCHEDULER` jsou nastaveny na nulu.

```
/* Standard includes. */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <assert.h>

/* FreeRTOS kernel includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Local includes. */
//#include "console.h"

#define mainSELECTED_APPLICATION 2
#define projCOVERAGE_TEST 1
```

```
/* This demo uses heap_3.c (the libc provided malloc() and free()). */

/*-----------------------------------------------------------*/
void taskA (void * pvParameters);
void taskB (void * pvParameters);
#if ( configUSE_EDF_SCHEDULER == 1 )
#define TASK_A_PERIOD 8
#define TASK_B_PERIOD 5
#define TASK_A_CAPACITY 2
#define TASK_B_CAPACITY 3
#endif
/*-----------------------------------------------------------*/
}
```

Nejprve si deklarujeme definice period a kapacit úloh `TASK_A_PERIOD` , `TASK_B_PERIOD` , `TASK_A_CAPACITY` a `TASK_B_CAPACITY`.

```
int main( void )
{
    printf("FreeRTOS simulation started!\n");
    printf("Using FreeRTOS scheduler.\n");
    assert( xTaskCreate( taskA, (const char *) "Task A",
                                configMINIMAL_STACK_SIZE,
                                NULL, 2, NULL)  == pdPASS);
    assert( xTaskCreate( taskB, (const char *) "Task B",
                                configMINIMAL_STACK_SIZE,
                                NULL, 3, NULL) == pdPASS );
    vTaskStartScheduler();
    for(;;);
    return 0;
}
/*-----------------------------------------------------------*/
```

Úlohy jsou vytvářeny pomocí již zmíněných metod `xTaskCreate()`, kde si do nich vložíme námi vytvořené definice patřičné priority kde úloha B má větší prioritu. Nakonec spustíme plánovač pomocí metody `vTaskStartScheduler()`. V tento okamžik nám běží úlohy A, B a IDLE.

```
void taskA (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_A_PERIOD;
    volatile int cnt = TASK_A_CAPACITY;
    printf("Starting task A\n");
    for(;;){
```

```
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                --cnt;
                tick = x;
            }
        }
        cnt = TASK_A_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
    }
}
void taskB (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_B_PERIOD;
    volatile int cnt = TASK_B_CAPACITY;
    printf("Starting task B\n");
    for(;;){
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                --cnt;
                tick = x;
            }
        }
        cnt = TASK_B_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
    }
}
#endif
```

Úlohy jsou spuštěny v nekonečném cyklu, kde v části `while(cnt)` se provádí obsah úlohy. Pokud si úloha odpracuje své, tak vyskočí z cyklu, dá nám o tom vědět a uspí se na danou periodu.

```
void vApplicationTickHook( void )
{
    taskENTER_CRITICAL();
    TaskHandle_t x = xTaskGetCurrentTaskHandle();
    TickType_t a = xTaskGetTickCount();
    printf("TICK : %ld\n\tNyní se zpracovává úloha: %s\n",
                                        a, pcTaskGetName(x));
```

```
FreeRTOS simulation started!
Using FreeRTOS scheduler.
Starting task B
TICK : 1
    Nyní se zpracovává úloha: Task B
TICK : 2
    Nyní se zpracovává úloha: Task B
TICK : 3
    Nyní se zpracovává úloha: Task B
Starting task A
TICK : 4
    Nyní se zpracovává úloha: Task A
TICK : 5
    Nyní se zpracovává úloha: Task A
TICK : 6
    Nyní se zpracovává úloha: Task B
TICK : 7
    Nyní se zpracovává úloha: Task B
TICK : 8
    Nyní se zpracovává úloha: Task B
TICK : 9
    Nyní se zpracovává úloha: Task A
TICK : 10
    Nyní se zpracovává úloha: Task A
TICK : 11
    Nyní se zpracovává úloha: Task B
TICK : 12
    Nyní se zpracovává úloha: Task B
TICK : 13
    Nyní se zpracovává úloha: Task B
TICK : 14
    Nyní se zpracovává úloha: IDLE
TICK : 15
    Nyní se zpracovává úloha: IDLE
TICK : 16
    Nyní se zpracovává úloha: Task B
TICK : 17
    Nyní se zpracovává úloha: Task B
TICK : 18
    Nyní se zpracovává úloha: Task B
TICK : 19
    Nyní se zpracovává úloha: Task A
TICK : 20
    Nyní se zpracovává úloha: Task A
TICK : 21
    Nyní se zpracovává úloha: Task B
TICK : 22
    Nyní se zpracovává úloha: Task B
TICK : 23
    Nyní se zpracovává úloha: Task B
TICK : 24
    Nyní se zpracovává úloha: IDLE
```

Obrázek 4.1: Průběh testu FreeRTOS plánovače

```
    taskEXIT_CRITICAL();
}
```

Zde si v každém tiku zavoláme výpis toho, co se děje, abychom v tom měli jasno. Zde předpokládáme uplatnění priorit úloh z definice FreeRTOS. Tyto úlohy jsou spouštěny tak, že pokud úloha B bude chtít pracovat, bude upřednostněna na úkor úlohy B.
Jak je vidět na obrázku 4.1, tak plánování souhlasí s předpokladem, že B je vždy upřednostněno na úkor A. Tudíž algoritmus FreeRTOS nelze použít pro plánování, kde je kritické, aby úlohy byly splněny v určitý okamžik, neboť jakákoliv úloha může jednoduše zablokovat chod celého zařízení. Proto je nezbytné aby programátor si na tohle dal pozor.

4.2 Testování EDF

Zdě program bude vykonávat úlohy s pomocí plánovače EDF, tudíž ve `FreeRTOSConfig.h` nastavíme `configUSE_EDF_SCHEDULER` na jedničku. Demo aplikace se bude skládat ze dvou úloh A a B, kde pro každou z úloh zadefinujeme její periodu a výpočetní dobu (kapacitu). Výpisy o stavu úloh budou vypsány na standardní výstup. Testovací program vypadá následovně:

```
/* Standard includes. */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <assert.h>

/* FreeRTOS kernel includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Local includes. */
//#include "console.h"

#define mainSELECTED_APPLICATION 2
#define projCOVERAGE_TEST 1

/* This demo uses heap_3.c (the libc provided malloc() and free()). */

/*-----------------------------------------------------------*/
void taskA (void * pvParameters);
void taskB (void * pvParameters);
#if ( configUSE_EDF_SCHEDULER == 1 )
#define TASK_A_PERIOD 8
#define TASK_B_PERIOD 5
#define TASK_A_CAPACITY 2
#define TASK_B_CAPACITY 3
#endif
/*-----------------------------------------------------------*/
}
```

Nejprve si deklarujeme definice period a kapacit úloh `TASK_A_PERIOD` , `TASK_B_PERIOD` , `TASK_A_CAPACITY` a `TASK_B_CAPACITY`. Úlohy jsou vytvářeny pomocí již zmíněných metod `xTaskPeriodicCreate()`, kde si do nich vložíme námi definované definice `TASK_A_PERIOD` a `TASK_B_PERIOD` . Nakonec spustíme plánovač pomocí metody `vTaskStartScheduler()`. V tento okamžik nám běží úlohy A, B a IDLE.

```
int main( void )
{
    printf("FreeRTOS simulation started!\n");
    #if ( configUSE_EDF_SCHEDULER == 1 )
    printf("Using Earliest Deadline First scheduler.\n");
    assert( xTaskPeriodicCreate( taskA, (const char *) "Task A",
                                 configMINIMAL_STACK_SIZE,
                                 NULL, 3, NULL, TASK_A_PERIOD )
                                                    == pdPASS);
    assert( xTaskPeriodicCreate( taskB, (const char *) "Task B",
                                 configMINIMAL_STACK_SIZE,
                                 NULL, 3, NULL, TASK_B_PERIOD )
                                                    == pdPASS );
    vTaskStartScheduler();
    for(;;);
    #endif
    return 0;
}
/*-----------------------------------------------------------*/



#if ( configUSE_EDF_SCHEDULER == 1 )
void taskA (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_A_PERIOD;
    volatile int cnt = TASK_A_CAPACITY;
    printf("Starting task A\n");
    for(;;){
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                --cnt;
                tick = x;
            }
        }
        cnt = TASK_A_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
    }
}
void taskB (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_B_PERIOD;
```

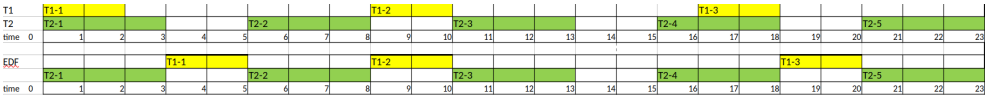
```
    volatile int cnt = TASK_B_CAPACITY;
    printf("Starting task B\n");
    for(;;){
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                --cnt;
                tick = x;
            }
        }
        cnt = TASK_B_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
    }
}
#endif
```

Úlohy jsou spuštěny v nekonečném cyklu, kde v části `while(cnt)` se provádí obsah úlohy. Pokud úloha si odpracuje své tak vyskočí z cyklu, dá nám o tom vědět a uspí se na danou periodu.

```
void vApplicationTickHook( void )
{
#if ( configUSE_EDF_SCHEDULER == 1 )
    taskENTER_CRITICAL();
    TaskHandle_t x = xTaskGetCurrentTaskHandle();
    TickType_t a = xTaskGetTickCount(), b = pcTaskGetPeriod(x);
    printf("TICK : %ld\n\tNyní se zpracovává úloha: %s\n\tTa má periodu: %ld\n"
            "\tPriorita byla spočítána jako: %ld\n",
                                        a, pcTaskGetName(x), b, a + b);

    taskEXIT_CRITICAL();
#endif
```

Zde si v každém tiku zavoláme výpis toho, co se děje, abychom měli přehled. Pro ulehčení porovnávání algoritmů použijeme příklad z ukázky EDF plánování, o kterém víme, že se naplánovat dá, a porovnáme jejich výsledky. Jak je vidět v obrázku 4.3 při porovnání plánování s naším ukázkovým příkladem, vše, co chceme, platí, dokonce i předpokládané rozhodování v čase 17. I zde se ukázalo, že algoritmus zvládl určit prioritu správně. I přesto by si měl programátor dát pozor, aby jeho program splňoval podmínku:

$$\mu = \sum_{i=1}^{N} \frac{c_i}{p_i} \leq 1$$



Obrázek 4.2: Plánování úloh v EDF [BI-SRC]

```
FreeRTOS simulation started!
Using Earliest Deadline First scheduler.
Starting task B
TICK : 1
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 6
TICK : 2
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 7
TICK : 3
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 8
Task B finishes in time: 3
Starting task A
TICK : 4
        Nyní se zpracovává úloha: Task A
        Ta má periodu: 8
        Priorita byla spočítána jako: 12
TICK : 5
        Nyní se zpracovává úloha: Task A
        Ta má periodu: 8
        Priorita byla spočítána jako: 13
TICK : 6
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 11
TICK : 7
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 12
TICK : 8
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 13
TICK : 9
        Nyní se zpracovává úloha: Task A
        Ta má periodu: 8
        Priorita byla spočítána jako: 17
TICK : 10
        Nyní se zpracovává úloha: Task A
        Ta má periodu: 8
        Priorita byla spočítána jako: 18
TICK : 11
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 16
TICK : 12
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 17
TICK : 13
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 18
TICK : 14
        Nyní se zpracovává úloha: IDLE
        Ta má periodu: 268435455
        Priorita byla spočítána jako: 268435469
TICK : 15
        Nyní se zpracovává úloha: IDLE
        Ta má periodu: 268435455
        Priorita byla spočítána jako: 268435470
TICK : 16
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 21
TICK : 17
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 22
TICK : 18
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 23
TICK : 19
        Nyní se zpracovává úloha: Task A
        Ta má periodu: 8
        Priorita byla spočítána jako: 27
TICK : 20
        Nyní se zpracovává úloha: Task A
        Ta má periodu: 8
        Priorita byla spočítána jako: 28
TICK : 21
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 26
TICK : 22
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 27
TICK : 23
        Nyní se zpracovává úloha: Task B
        Ta má periodu: 5
        Priorita byla spočítána jako: 28
TICK : 24
        Nyní se zpracovává úloha: IDLE
        Ta má periodu: 268435455
        Priorita byla spočítána jako: 268435479
```

Obrázek 4.3: Průběh testu EDF

.

4.3 Testování LLF

V této sekci program zaměříme na vykonávání úloh s pomocí plánovače LLF, tudíž ve `FreeRTOSConfig.h` nastavíme `configUSE_LLF_SCHEDULER` na jedničku. Demo aplikace se bude skládat ze tří úloh A, B a C, kde pro každou z úloh zadefinujeme její periodu a výpočetní dobu (kapacitu). Výpisy o stavu úloh budou vypsány na standardní výstup nestandardním způsobem který si popíšeme. Testovací program vypadá následovně:

Nejprve tedy deklarace definic nezbytné pro jednotlivé úlohy. Oko bystrého čtenáře si již rychle povšimlo toho, že úlohy jsou ve všech teste dosti podobné nebo dokonce i stejné. To jim však nebrání plnit jejich úlohu dobře.

```
void taskA (void * pvParameters);
void taskB (void * pvParameters);
void taskC (void * pvParameters);
#if ( configUSE_LLF_SCHEDULER == 1 )
#define TASK_A_PERIOD 7
#define TASK_B_PERIOD 5
#define TASK_C_PERIOD 3
#define TASK_A_CAPACITY 2
#define TASK_B_CAPACITY 2
#define TASK_C_CAPACITY 1
#endif
```

Následně kód, který budou naše tři nové úlohy obsahovat:

```
void taskA (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_A_PERIOD;
    volatile int cnt = TASK_A_CAPACITY;
    printf("Starting task A\nWith capacity: %d\n",cnt);
    for(;;){
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                taskENTER_CRITICAL();
                --cnt;
                tick = x;
                taskEXIT_CRITICAL();
            }
        }
        cnt = TASK_A_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
```

```
    }
}
void taskB (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_B_PERIOD;
    volatile int cnt = TASK_B_CAPACITY;
    printf("Starting task B\nWith capacity: %d\n",cnt);
    for(;;){
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                taskENTER_CRITICAL();
                --cnt;
                tick = x;
                taskEXIT_CRITICAL();
            }
        }
        cnt = TASK_B_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
    }
}
void taskC (void * pvParameters){
    TickType_t lastWakeTick = 0;
    const TickType_t period = TASK_C_PERIOD;
    volatile int cnt = TASK_C_CAPACITY;
    printf("Starting task C\nWith capacity: %d\n",cnt);
    for(;;){
        TickType_t tick = xTaskGetTickCount(),
                   x;
        while(cnt){
            if((x = xTaskGetTickCount()) > tick){
                taskENTER_CRITICAL();
                --cnt;
                tick = x;
                taskEXIT_CRITICAL();
            }
        }
        cnt = TASK_C_CAPACITY;
        xTaskDelayUntil( & lastWakeTick, period );
    }
}
```

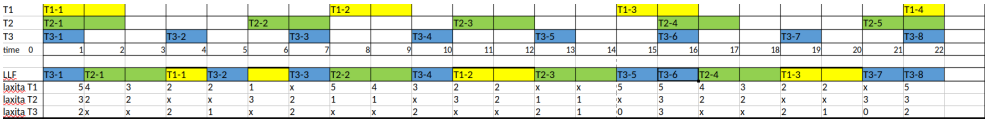
V neposlední části si vytvoříme ve funkci `vApplicationTickHook()` funkce pro výpisy důležitých informací každém tiku programu.

```
...

taskENTER_CRITICAL();
TaskHandle_t x = xTaskGetCurrentTaskHandle();
TickType_t a = xTaskGetTickCount(), b = pcTaskGetCapacity(x),
           c = pcTaskGetPeriod(x), d = pcTaskGetExecTime(x);
printf("TICK : %ld\n\tNyní se zpracovává úloha: %s\n"
        "\tTa má výpočetní kapacitu: %ld\n\tMá periodu: %ld\n"
        "\tV aktuální okamžiku zbývá dopočítat: %ld\n"
        "\tLaxita byla spočítána jako: %ld, a je nejmenší"
        "nebo si mohl plánovač vybrat\n",
        a, pcTaskGetName(x), b, c, b - d, c -
        ( ( ( a - 1 ) % c ) + ( b - d ) ) );
taskEXIT_CRITICAL();

...
```

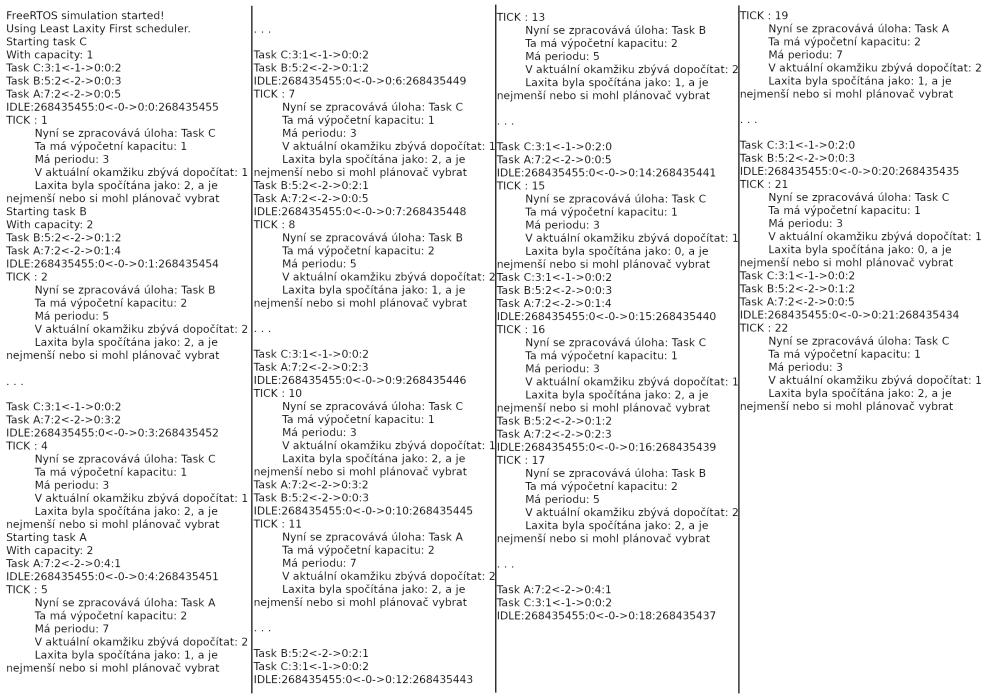
Jak jsem již zmínil o trochu dříve, je čas si vysvětlit jaký, výpis nám dává funkce `printf("%s:%ld:%ld<-%ld->%ld:%ld:%ld\n", ... );` zmíněný v kapitole implementace LLF:

- `%s` - název úlohy
- `:%ld:` - perioda
- `%ld<-` - trvání úlohy
- `%ld` - doba potřebná k dokončení
- `->%ld` - doba kterou již úloha prošla
- `:%ld:` - aktuální tik modulovaný periodou
- `:%ld:` - laxita

Mějme tedy, následující tři úlohy A, B a C kde perioda $p_A = 7$, perioda $p_B = 5$ a perioda $p_C = 3$. Délky jednotlivých úloh jsou: $c_A = 2$, $c_B = 2$ a $c_C = 1$. Tyto úlohy jsou plánovány ve stylu LLF a náš očekávaný výstup vypadá stejně jako na obrázku 4.4.
Po spuštění programu dostáváme následující výpis zobrazený v obrázku 4.5. Porovnáním s naším referenčním řešením dospíváme k závěru, že plánovací algoritmus dělá to, co má, a i když úlohy nejsou plánovány přesně v našem



Obrázek 4.4: Plánování úloh v LLF [BI-SRC]



Obrázek 4.5: Průběh testu LLF

předpokládaném pořadí, výsledek spadá do množiny možných řešení. Odlišnost našich řešení je způsobeno stejnou laxitou více úloh v daný časový okamžik. V porovnání s předchozími algoritmy je tento algoritmus nejkomplexnější, tudíž v našem systému zabere více režijního času, což může být v některých případech značně nežádoucí (např. v systémech, kde je vyžadována rychlost a nehledí se na dodržení deadlinu úloh).

4.4 Závěr testování

Při porovnání všech tří algoritmů docházíme k následujícím závěrům:

- **FreeRTOS plánovač** je základní plánovací algoritmus v systému reálného času. Jeho užití je limitováno víceméně schopnostmi programátora. Plánovač nehledí na deadliny úloh a tudíž nejsou ani definovány. Celý běh programu je omezen na systém statických priorit, které se nastaví na začátku programu nebo se můžou přenastavit kdekoliv v průběhu úlohy. Hlavní výhoda toho systému je rychlost a konstantní složitost vkládání a vybírání úloh. Nevýhody jsou lineární složitost pro přepínání kontextu. Pokud by se přepínání kontextu vzdalo, docílilo by se toho, že systém by přestal být preemptivní a následek by mohlo být nepuštění úloh s větší prioritou k práci. Pro srovnání vůči ostatním algoritmům, budeme brát tento algoritmus jako bod 0. Algoritmus je deterministický.

- **EDF plánovač** je druhý, jednoduchý na implementaci, algoritmus kterým se tato práce zabývá. Hlavní výhodou tohoto systému je uplatnitelnost v prostředí kde je vyžadována minimální přesnost na úlohy, jelikož algoritmu se může dostat do stavu kdy nedokáže dobře naplánovat úlohy a bude docházet k deadlinům. Dalšími výhodami jsou rychlost, nízká komplexita, konstantní složitost přepínání kontextu úloh. Naopak nevýhoda je vyšší riziko nedodržení deadlinu. O proti plánovači FreeRTOS, tak EDF má o 7 řádků kódu méně. Algoritmus není deterministický.

- **LLF plánovač** je poslední z našich třech algoritmů a také nejkomplexnější z nich. Tuto metodou jsme schopni dosáhnout dobrého naplánování úloh v místech, kde například by algoritmus EDF neobstál. Protože se algoritmu může dostat do situací, kde si může vybrat, kterou úlohu zrovna spustí, tak díky tomu, že algoritmu se nedívá do „budoucna“, nedokáže vždy vybrat tak aby uspokojil deadliny všech úloh. Na rozdíl od EDF se tento plánovací algoritmus dá téměř vždy využít až na 100%. Hlavní výhodou je tedy maximální využitelnost procesoru. Naopak nevýhody jsou: vyšší komplexita, je pomalejší z našich vybraných algoritmů, nedokáže vždy určit naplánování tak aby se uspokojily všechny deadliny. O proti plánovači FreeRTOS, tak LLF má o 12 řádků kódu více. Algoritmus není deterministický.

Závěr

Tato práce se věnovala plánovacím algoritmům systémů reálného času s dynamickou prioritou z předmětu BI-SRC, k němuž byla práce vytvořena jako studijní pomůcka. Dynamické algoritmy Earliest Deadline First a Least Laxity First byly implementovány do předmětem používaného systému reálného času FreeRTOS. Aby byla implementace možná, jsou kapitoly věnovány problematice daných plánovacích algoritmů a nejvíce právě struktuře otevřeného systému reálného času FreeRTOS.

V sekcích kapitoly 2 byly vysvětleny a ukázány vybrané plánovací algoritmy, jejichž implementace do FreeRTOS jsou detailně popsány v příslušných sekcích kapitoly 3, kde jsou zvýrazněny nejdůležitější části kódu. V neposlední řadě jsou vybrané algoritmy vystaveny testování vzhledem k vybraným testovacím úlohám.

Nejlepší volbou z implementovaných algoritmů je algoritmus dodávaný FreeRTOS, i přestože algoritmus neřeší problém dodržování deadlinu úloh (programátor je schopen tuto funkčnost doplnit). Na rozdíl, od ostatních zde představených plánovacích algoritmů, tento má konstantní složitost pro vkládání a odebírání úloh, naopak při změně kontextu má lineární složitost. Pro mnoho úloh má jádro tohoto plánovače nejmenší overhead. Výsledky simulace ukázaly, že řešení plánovače FreeRTOS jsou v mnoha případech lepší volbou pro plánování úloh, než EDF a LLF. Práce tudíž splňuje zadání a proto ji považuji za splněnou.

Téma se dá dále rozvíjet a doplnit například o:

- **WCET:** Doplnit práci, aby FreeRTOS mohl určovat WCET (čas který daná úloha potřebuje pro svůj výpočet v nejhorším případě). Výpočet WCET je netriviální problém, který vyžaduje kód pro analýzu běhu programu [7].
- **Moduly plánovače:** Vytvořit nové možnosti plánování ve formě modulů. jedná se o odstranění plánovacích algoritmů z `task.c` a vložení jich do samostatných souborů, které se budou vkládat dle potřeb programátora.

Předpokládáme, že výsledky této práce budou použity v předmětu BI-SRC. Například ji aplikovat na úlohu č.4 (Zybo). Následně pozorovat, jak se nově algoritmus zachová a jestli zvládne úlohy naplánovat a podobně.

Literatura

[1] Kopetz, H.: *Real-time systems: design principles for distributed embedded applications.*, *Real-time systems*, ročník 2. Boston: Springer: Springer Publishing, 2011.

[2] Barry, R.: *Mastering the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd., 2016. Dostupné z: `https://freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf`

[3] Amazon Web Services: *The FreeRTOS™Reference Manual*. 2017. Dostupné z: `https://freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf`

[4] deadline first scheduling, E.: Earliest deadline first scheduling — Wikipedia, The Free Encyclopedia. Dostupné z `https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling`, 2021, [Online].

[5] Kubátová, H.: Systémy reálného času - Dynamické plánování [přednáška]. Praha: FIT ČVUT. [online], 2020. Dostupné z: `https://courses.fit.cvut.cz/BI-SRC/media/lectures/07/src-7-sched-2.pdf`

[6] slack time scheduling, L.: Least slack time scheduling — Wikipedia, The Free Encyclopedia. Dostupné z `https://en.wikipedia.org/wiki/Least_slack_time_scheduling`, 2019, [Online].

[7] Engblom, J.; Ermedahl, A.; Sjödin, M.; aj.: Worst-case execution-time analysis for embedded real-time systems. *STTT*, ročník 4, 08 2003: s. 437–455, doi:10.1007/s100090100054.

[8] Mattihalli, C.: Designing and Implementing of Earliest Deadline First Scheduling Algorithm on Standard Linux. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on*

Cyber, Physical and Social Computing, 2010, s. 901–906, doi:10.1109/GreenCom-CPSCom.2010.82.

[9] microcontrollerslab.com: LEAST LAXITY FIRST (LLF) SCHEDULING ALGORITHM. [online]. Dostupné z: `https://microcontrollerslab.com/least-laxity-first-llf`

[10] Chattopadhyay, S.; Tresina, M.; Narayan, S.: Worst Case Execution Time Analysis of Automotive Software. *Procedia Engineering*, ročník 30, 12 2012: s. 983–988, doi:10.1016/j.proeng.2012.01.954.

Příloha **A**

Seznam použitých zkratek

CPU Central processing unit

EDF Earliest deadline first

ISR Interrupt service routine

LLF Least laxity first

LTS Long term support

PC Program counter

RTOS Real time operating system

RMS Rate monotonic scheduling

SP Stack pointer

TCB Task control block

WCET Worst case execution time

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `build`.. adresář se spustitelnou formou implementace (vzniká a zaniká při spuštění make)
  - `impl` ..... zdrojové kódy implementace
  - `doc` ..... umístění zdrojové formy práce
    - `BP_Zápotocký_Josef_2021.tex`... zdrojová forma práce ve formátu LaTeX
- `doc` ..... text práce
  - `BP_Zápotocký_Josef_2021.pdf` ..... text práce ve formátu PDF
  - `Prezentace_BP.pdf` ..... prezentace práce ve formátu pdf