Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Implementace vybraných metod pro plánování úloh v RTOS μC/OS-II

Plzeň, 2012 František Svoboda

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. května 2012

František Svoboda

Abstract

Implementation of selected methods for task scheduling in RTOS μC/OS-II This work describes how to change task scheduling in μC/OS-II operating system. This work also shows what is need in Real-time operation system. Scheduling methods are created for the pure code RTOS. The port's examples for (x86 / Renesas microcontroller H8S2633F) are written in C and processor-specific assembly language. The CD-ROM contains source code for the H8S2633F port and implements scheduling algorithm.

Obsah

1 Úvod

μC/OS-II je operační systém reálného času dodávaný ve formě zdrojových kódů programovacího jazyka C, implementovaný na více než 40 hardwarových architekturách. Za jeho vývojem stojí společnost Micriμm Technologies Corporation a je používán v celé řadě aplikací od řídící elektroniky až po letectví. Pro nekomerční a akademické použití nemusí být tento systém licencován.

K implementaci μC/OS-II na konkrétní procesor je nutno upravit jeho konfiguraci a doplnit zdrojové kódy operačního systému o kód specifický pro daný procesor. Takto upravený systém μC/OS-II bývá nazýván portem. Pro vytvoření spustitelného kódu je nutný překladač jazyka C, assembleru a sestavovací program.

Cílem této práce je implementace jiného než standardního plánování procesů operačního systému μC/OS-II 2.86. Standardním se myslí statické nastavení priorit po spuštění systému. Výsledky by měly být ověřeny vhodnými testovacími úlohami, např. na H8S2633F, což je obchodní název jednočipového mikropočítače firmy Renesas Technology Corporation. Očekáváno je lepší využití strojového času procesoru. Dále u případových úloh zaručení splnění mezního času vykonávání oproti stávající implementaci, zároveň mezi implementovanými algoritmy.

- Seznamte se s operačním systémem MicroC/OS-II.
- Prostudujte metody plánování úloh v RTOS. Vyberte metody vhodné pro implementaci v μC/OS-II.
- Navrhněte úpravy plánovače procesů v μC/OS-II tak, aby umožňoval plánování podle zvolených metod.
- Správnou funkčnost plánovače demonstrujte na vhodných příkladech.

2 Real-time systémy

Real-time operační systémy (RTOS) jsou speciálně navrhovány pro splnění přísných časových omezení. Problematiku RTOS si ukážeme na následujícím příkladu z běžného života.

Jako příklad uveďme systém vyhodnocující údaje v automobilu např. systémy ABS, ESR apod. Zde nás ihned napadne, že zpracování údajů od bezpečnostních prvků musí mít aktuální hodnoty. Tedy máme pevně daný časový údaj, do kdy musí být informace vyhodnoceny. Není-li toto splněno, jsou informace bezcenné.

Nejen v automobilech nalezneme systém, který zpracovává obraz před automobilem (k rozeznání dopravních značek). Zde již není tak striktní omezení na dobu odezvy. Uvědomíme-li si, jak zásadní je získání informace o snížené rychlosti. Obraz bývá obvykle zpracováván na vzdálenost 100m, tedy při 90km/h ujedeme při zpoždění jedné sekundy 25m. I po této době máme značku stále před námi a tedy zónu se sníženou rychlostí také.

Ale například při zpoždění 5s bychom byli už v zóně a v lepším případě by nám hrozila pokuta. Máme tedy definován deadline události, ale nemusíme jej tak striktně dodržet jako v případě bezpečnostní výbavy. Systému, kde je povoleno překročení deadlinu o nezbytně nutné zpoždění.

Můžeme říci, že téměř každý řídící systém, je systém reálného času. Pro představu kde se RTOS používá, uveďme některé příklady.

- Automobily - řídící jednotky motoru, bezpečnostní prvky výbavy
- Domácnost - pračky, mikro-vlnné trouby, myčky, kotle na vytápění
- Letectví - zbraňové systémy, navigační systémy, radarový systém
- Řízení výroby - automatizované výrobní linky

Například rozvinuté letecké kontrolní systémy, které používají jeden centrální počítač, se musí vyrovnat s několika letadlovými subsystémy, což vyžaduje operační systém s oddělením času a prostoru pro připojené systémy. Prostorové dělení znamená, že každý úkol je bezpečně izolovaný v paměti počítače, zatímco časové oddělení je vlastně rozdělení výkonného času procesoru.

Toto dělení umožňuje jedinému procesoru vykonávat několik úloh současně, bez rizika jednoho úkolu způsobit problémy v jiných úkolech. Výsledkem tohoto je např. snížení počtu počítačů v letadle a tedy redukovat jeho hmotnost.

Tento druh systémů je vhodné používat tam kde, potřebujeme reflektovat přirozený paralelismus skutečných dějů. Synchronizace a výměna dat mezi úlohami reprezentuje interakci mezi reálnými ději. Oddělení funkcí (co úloha dělá) a časových charakteristik (kdy to dělá) jednotlivých úloh.

Samozřejmě je to prostředník mezi hardwarovou a aplikační vrstvou. Pomáhá programům efektivně využívat hardwarové a systémové prostředky. Vzájemně je od sebe umí oddělit. V několika situacích RTOS jsou přítomny v oblasti embedded systémů, a většinu času nevyžadují zásah uživatelů.

2.1 Historie a současný stav

2.2 Obecné pojmy

V této části uvedeme pouze nejzákladnější vlastnosti.

2.2.1 Doba odezvy

Hlavním parametrem RTOS je včasnost odezvy na vstupní podněty. Reakce na podněty nemusí být nutně za jednotky mikrosekund, aby byl považován za RTOS. Důležité je, že musí dodržet časové meze kladené na dobu jeho odezvy pro účely dané konkrétní RT aplikace.

2.2.2 Determinismus

Systémy reálného času musí být předvídatelné vzhledem ke svému chování v čase. Musí být vždy zaručena maximální doba odezvy, po kterou systém úlohu splní. V opačném případě nastává jeho selhání.

Determinismus není omezen pouze na časové spektrum, ale označuje před-

vídatelnost událostí. Lze tedy v jeho libovolném stavu s příslušnými hodnotami na vstupu jednoznačně určit, jaký bude jeho následující stav a výstup. První případ determinismu se nazývá temporální, druhý událostní. Jinými slovy řečeno, je určitelný v čase a nebo na základě údálosti.

2.3 Klasifikace RTOS

Dělení RT systémů, může být bráno dle řady hledisek. Ve většině literatury jsou děleny podle potřeby dodržet časové požadavky kladené na systém.

2.3.1 Hard

Na úvodním příkladu jsme naznačili časovou závislost výsledků úloh. Podle ní dělíme také na hard real-time a soft real-time. Hard systémy dodržují termín dokončení dávky přesně, pokud v čas nedokončíme úlohu, chování systému obvykle vede ke katastrofálním dopadům. Např. kontrolní a řídící systémy jaderné elektrárny striktně dodržují termín ukončení[1].

2.3.2 Soft

Systémy nedodržující striktně termín ukončení, mohou ”občas”minout deadline. Co znamená ”občas”? Nejčastěji se definuje procentuálním zastoupením, např. 99% bude dodrženo. Nebo se definují funkce v závislosti na čase, kde po termínu ukončení mají klesající tendenci relevantnosti dat. Klesající použitelnost dat a zpoždění, nám zde nevadí.

2.3.3 Firm

Oproti tomu systémy firm mají mez, do které jsou data relevantní, a nevede opoždění termínu ukončení k pádu systému. Pakliže je tato mez překročena, může to mí vliv na stabilitu, ba dokonce na pád systému.

[1] v ang. lit. deadline

3 Přehled RTOS

Na trhu existuje celá řada RTOS a vybrat z nich nemusí být jednoduché. Můžeme vybírat z volně dostupných tak i komerčních. Také můžeme vybírat podle podporovaných platforem, podpoře samotného výrobce či komunity. Většina RTOS má vlastnosti popsané výše. V následující části jsou vyzdvihnuty vlastnosti, kterými se daný výrobce prezentuje. Uvedené systémy jsou byly vybrány podle nejčastějšího zastoupení a jsou minimem toho, co nabízí trh.

3.1 Windows Embeded

Mezi hlavní zbraně systémů postavených na platformě společnosti Microsoft [1] patří podpora svých služeb. Díky tomu se dají ušetřit nemalé výdaje za vývoj nadstandartních požadavků. Využití najde od přenosných ultrazvukových přístrojů, GPS zařízení, bankomatů po využití ve velkých stavebních strojích.

Hlavní doménou těchto systémů je práce s vlastními technologiemi. Zejména pro podporu synchronizace dat, využití profilů, místní služby, reklamní odvětví, business intelligence a line-in-business aplikací, zpracování dat ze zařízení. Zapojení zařízení do jedné platformy přináší například:

- Silverlight pro Windows Embedded
- Internet Explorer Embedded
- Dotykové ovládání
- Media Player
- Office and PDF Viewers
- a další

Mezi podporované platformy patří ARM architektura, x86 procesory, x64 procesory, MIPS a další. Samozřejmostí se pak stává podpora například některých souborových systémů, bluetooth technologie a další. Vývoj aplikací je plně podporován vývojovým prostředí Microsoft Visio.

[1] [WinEM]

3.2 QNX Neutrino

Rozsáhlá podpora POSIX standardů usnadňuje přenositelnost aplikací a rychlou migraci z Linux, Unix, a dalších open source programů. Runtime podpora a sady BSP pro populární chipsety, včetně ARM, MIPS, PowerPC, SH-4, a x86, umožňuje návrhářům vybrat nejlepší hardwarové platformy pro jejich embedded systém, pak se rychle dostat své systémy do provozu.

Díky mikrojádru architektury QNX Neutrino RTOS[2], prakticky jakákoli součást (i low-level ovladač) může selhat, bez poškození jádra nebo jiné součásti. Vytížení procesoru, pokud jsou zdroje omezené, získají procesy dle podílů.

Pokud ovladače zařízení, protokol zásobníku, nebo aplikace selže. Je v QNX Neutrino RTOS zařízeno, tak aby nezkolaboval systém. Ale manažer ukončil úlohu s problémem a obnovil jej (často za několik milisekund bez restartu). Podporované jsou též síťové technologie, mezi něž patří IPv4, IPv6, IPSec, FTP, HTTP, SSH a Telnet. Unikátní transparentní distribuce umožňuje přístup k prostředkům na libovolném uzlu v síti, pro bezproblémové peer-to-peer sdílení zdrojů.

QNX souborové systémy jsou obsluhovány mimo mikrojádro v uživatelské chráněné-paměti. Vývojáři mohou spustit, zastavit, nebo aktualizovat souborové systémy za běhu bez nutnosti restartu. Více souborových systémů může být spuštěno současně a dokonce pracovat naráz s cílem rozšířit navzájem schopnosti.

3.3 Wind River VxWorks

VxWorks[3] byl navržen pro použití v různých multi-jádrových sestavách jako svého času jediný operační systém na symetrickém/asymetrickém multiprocesovém režimu nebo jako host OS na vrcholu Wind River Hypervisor. VxWorks je vysoce škálovatelný RTOS.

VxWorks systémy zahrnují certifikovaný OS pro velmi přísné bezpečnostní normy. VxWorks také nabízí několik úrovní zabezpečení, vysokou odol-

[2][QNXso]
[3][VxWor]

nost RTOS bezpečnosti pro náročné požadavky Common Criteria a National Information Assurance Partnership (NIAP). VxWorks má řešení pro každý aspekt vestavěných systémů komunikace.

S Wind River Tilcon Graphics Suite mají vývojáři k dispozici rychlý návrh grafického uživatelského rozhraní. Podporované jsou mikroprocesory například společností Aitech Defense Systems Inc., AMD, ARM Ltd., Broadcom Corporation, Freescale Semiconductor, Goma Elettronica S.P.A., Hilscher Gesellschaft fur Systemautomation mbH., Integrated Device Technology (IDT), Intel Corporation a mnoho dalších.

3.4 Nucleus

Společnost Mentor Graphics[4] též nabízí RTOS. Silné stránky tohoto systému jsou v rozdělení funkcí, tak aby byly co nejvíce nezávislé a při pádu aplikace nezkolaboval celý systém. Operační systém poskytuje bohaté možnosti pro grafické uživatelské rozhraní. Samozřejmostí je ovládání dotykových panelů.

Nechybí zde podpora nejrůznějších standardních komunikačních rozhraní a protokolů. Mezi něž patří USB, SATA, IDE a mnohé další. Umožňuje také virtualizace některých rozhraní, např. RS232. Mezi síťovými protokoly nechybí IPv4 ani IPv6, součástí jsou přípravy například pro SNTP klienty, SMTP klient/server.

Podpora multimédií je na vysoké úrovni, využívají otevřené, platformě nezávislé API OpenMax. To umožňuje přehrávat standardní typy audio a video formátů. S podporou rozhraní paměťových médií, je spojena podpora souborových systémů FAT12, FAT16, FAT32 a také vlastního Nucleus SAFE pro flash paměti.

Podporované platformy jsou PowerPC, MIPS, ColdFire, Atmel. Stejně jako u většiny předchozích, závisí platforma na překladači zde jazyka C++ pro cílovou platformu.

[4] [Nucle]

3.5 LynxOS RTOS

Komponenty RTOS v LynxOS[5] jsou navrženy pro absolutní determinismus. Tyto předvídatelné reakce jsou zajištěny i v přítomnosti složitých I/O operací, protože jádro systému umožňuje extrémně krátké a rychle obsloužené přerušení.

LynxOS RTOS také používá lineární škálovatelnost, takže zůstává neochvějně deterministický. Toho je využito zejména pro výbornou odezvu například v komunikacích. Schopnosti síťové komunikace tohoto systému jsou taktéž vyspělé, nechybí například IPSec, IPv6, integrovaný firewall, a Quality of Service (QoS). Zahrnuje nejnovější protokoly a možnosti pro vytváření síťových prvků, včetně Gigabit Ethernet, SNMP v1, 2 a 3, směrovací algoritmy jako RIPv2, OSPFv2 a BGP-4.

Vývoj aplikací a úpravy systému jsou doporučovány pro Eclipse, který je de facto šířen jako otevřený vývojový nástroj. Vhodným doplňkem je i debugger SpyKer schopný monitorování všech událostí v systému.

Mezi platformami, které tento systém podporuje, najdeme x86 od desek s procesory s 386 až po Core 2 Duo procesory. Vyjímkou nejsou ani některé procesory AMD a Via Cyrix. Samozřejmě najdeme zástupce další architektury, mezi něž patří nejrůznější procesory PowerPC, některé desky s MIPS mikroprocesory, nebo zástupce od IBM řada 440.

3.6 Linux

Zabývat se schopnostmi operačních systémů založených na Linuxovém jádře je takřka nemožné. A to díky tomu, že pokrývají nepřebernou oblast použití, od jednoduchých síťových prvků, přes mobilní telefony až po serverové stanice. Navíc zde narážíme na nejednotnost distribucí, především v desktopové sféře.

Pro operační systém Linux existují externí řešení, které jsou v implementovány do jádra systému. Od počátku nebyl Linux navrhován jako Real-Time operační systém. Změnit jeho jádro, které má navrženo fixně prioritní plánování, by bylo přinejmenším nevhodné a komplikované. Z tohoto důvodu

[5] [LyRto]

je možné doplnit tento systém o RT plánování, formou minimálních úprav jádra a dodáním externích modulů plánování.

Fungují zde tak pospolu jak úlohy real-time-ové, tak ne-real-time-ové pospolu. Ne-RT úlohy jsou vykonávány v případě, že nemá být vykonávána RT úloha.

4 μC/OS-II

V této kapitole probereme v rychlosti vlastnosti vybraného real-time operačního systému. Nebudeme se jimi však zabývat dopodrobna, není to cílem práce. Spíše to berme jako základ, který by měl vývojář pracující s μC/OS-II znát.

4.1 Úlohy

Úlohou označujeme programový kód oddělený v čase a paměti, který vykonává požadovanou funkci. Obvykle je úloha implementována jako nekonečná smyčka. Verze 2.00 obsahuje 64 prioritních úrovní, každá může odpovídat právě jedné úloze. Ve skutečnosti jsou některé rezervovány pro systémová vlákna. Nejvyšší čtyři a nejnižší čtyři v pozdějších verzích je možnost využití až 256 priorit. Jádro umožňuje pomocí rezervovaných funkcí měnit priority vláken, označit vlákno za smazané, fakticky jej ale nesmaže.

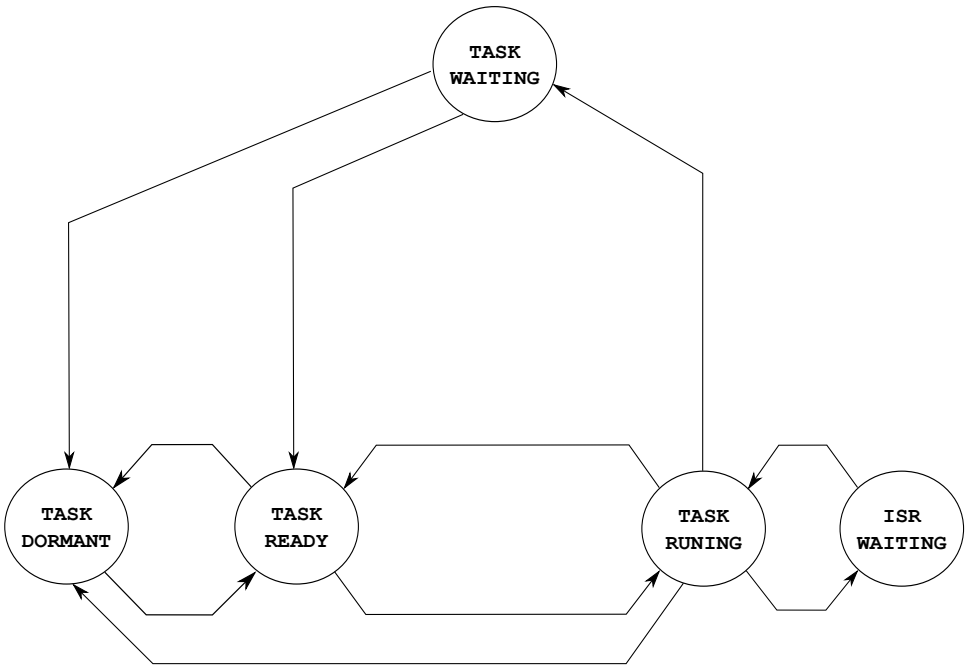
4.1.1 Stavy úloh

Každý úkol při svém běhu se může dostat do několika stavů. Podle nich plánovač určuje, která úloha bude vykonávána. Stavy úloh v μC/OS-II jsou:

- dormant
- ready
- running
- waiting
- interrupted

Dormant odpovídá stavu, kdy je vlákno v paměti, ale není dostupné. Úloha, která je ready, čeká na ukončení právě běžícího vlákna a je připraveny k běhu. Vlákno vlastnící CPU je running. Waiting úkol, čeká na dokončení např. I/O operace, na sdílené zdroje. Interrupt service routine[1] obsluhuje

[1] Interrupted dále jen ISR



Obrázek 4.1: Stavy vláken

přerušení vzniklé nenadálou událostí a přebírá CPU. Na obr. 4.1 jsou stavy znázorněny spolu s přechody mezi nimi.

Obecně nemůžeme říci, že by každý RTOS používal stejné označení, ale jistě přidá či ubere nějaký jiný stav.

4.1.2 Task Control Block

Každé vytvořené vlákno má k sobě přidružený paměťový prostor nazývaný *task control block*. TCB je datová struktura, používaná μC/OS-II k udržení stavu nejen přerušené úlohy. Po vykonání úlohy s vyšší prioritou, návratu k přerušené a opět získat kontrolu nad CPU, tam kde vykonávání opustila. Struktura obsahuje

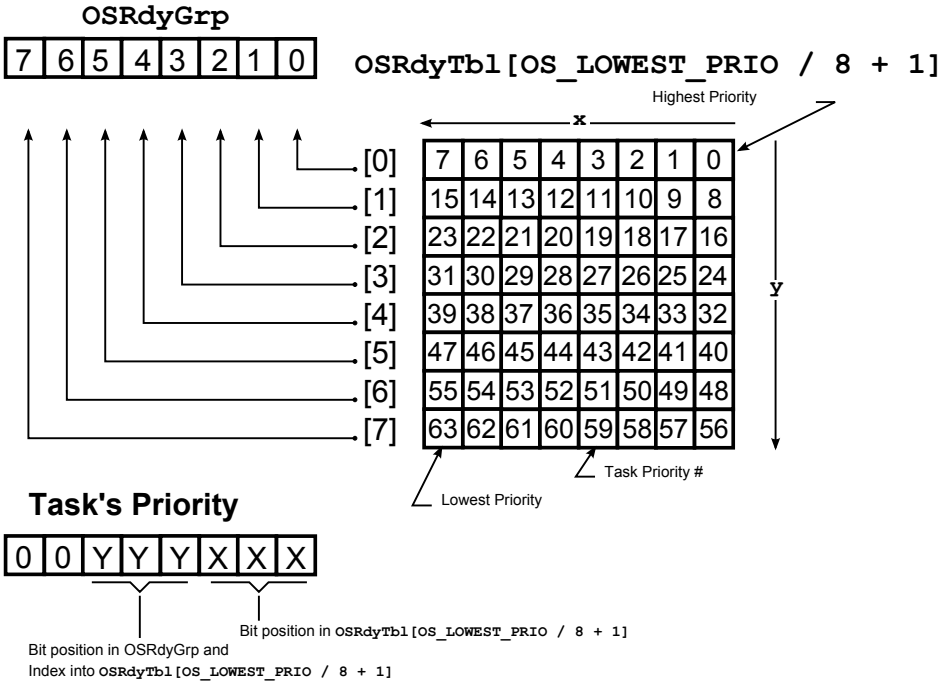
- ukazatel na vrchol zásobníku úlohy
- ukazatel na dno zásobníku úlohy
- velikost zásobníku

- nastavení rozšíření zásobníku
- nastavení ověření zásobníku
- indikaci vyčištění zásobníku
- nastavení použití Floating Point
- identifikátor vlákna
- následníka a předchůdce (TCB)
- ukazatel na Event Control Block
- ukazatel na zprávy vláknu
- ukazatel na zachycené události
- informaci zda bude úloha ready po splnění vybrané události
- informace, zda je úloha odložena na určitý čas
- stav úlohy
- prioritu úlohy
- udaje zrychlující výběr ready úlohy
- indikace smazání úlohy
- ukazatel na uživatelské rozšíření TCB

Po inicializování µC/OS-II jsou všechny TCB v tabulce propojeny jako spojový seznam volných OS_TCB. Velikost tabulky je omezitelná nastavením počtu úloh v souboru *os_cpu.h*. Jiné systémy mohou udržovat další údaje, např. některé se hodí pro dynamické plánování úloh, statistické údaje apod.

4.1.3 Výběr úlohy

Každá úloha ve stavu ready je umístěna v seznamu *ready list*. Položky nesou informace o *OSRdyGrp* a *OSRdyTbl[]*. Priority jsou seskupovány po osmi úlohách ve skupině. Na obr.4.2 je znázorněna struktura ready listu a výběr položky. Bitová maska je v kódu 1 z n nikoli v klasické dvojkové soustavě. Každý bit *OSRdyGrp* reprezentuje svým indexem skupinu, která obsahuje



Obrázek 4.2: Výběr úlohy z Ready listu

vlákno ve stavu ready, a je indexem do *OSRdyTbl[]*. Než je přepnut kontext procesoru je nutné vybrat z ready listu úlohu. V původní implementaci μC/OS-II je doba výběru závislá na počtu úloh a je konstantní. Způsob jak najít ready úlohu je prostý, ale pomalý. Plánovač by musel vždy projít celou OSRdyTabl[], naštěstí je toto vyřešeno statickou lookup tabulkou. Ve které pomocí indexu OSRdyGrp získáme Y souřadnici do OSRdyTbl[] a z indexu OSrdyTbl[Y] získáme X souřadnici. Takto popsaný ready list nelze brát jako obecně používaný v RTOS. Objevují se varianty použití prioritních front, zajištění správné umístění pozice podle priority je otázkou řadících algoritmů, nebo použitím klíčů jako priorit.

Při výběru úloh plánovač může být přerušen. Obsluha a správa přerušení je dostupná i v případě přeplánování. Pouze by se nemělo stát, že přerušení vyvolá opětovné přeplánování. Totiž v případě návratu do plánovače, na stejné místo kde byl přerušen, by mohlo dojít k situaci, že právě vyvolané přerušení-přeplánování změnilo poměry v systému a nemusela by být vybrána správná úloha. K zákazu přerušení systém definuje funkce *OSShedLock()* a *OSShedUnLock()* tyto funkce se musí používat párově, tzn. zamknout a po vybrání úlohy odemknout.

4.1.4 Přepínání kontextu

Po výběru úlohy je nutné uchovat informace o přerušené či pro danou chvíli splněné úloze. K tomu je předdefinováno makro, které musí vývojář portace vytvořit pro každou platformu zvlášť. Měl by se postarat o bezchybné uložení hodnot registrů procesoru. Implementace tohoto makra je záležitostí obsluhy softwarového přerušení. Přenositelnosti a napsání maker pro specifické platformy je záležitost velmi komplikovaná, podrobněji se jí věnuje autor systému ve své knize [MicroII].

4.2 Přerušení

V souvislosti s výběrem úlohy a přepnutím kontextu jsme zmínili přerušení. Interrupt[2] je schopnost procesoru pozastavit právě vykonávaný program a začít vykonávat jiný program (obsluhu přerušení). Začalo se implementovat do procesorů z důvodu obsluhy periferii – přesněji V/V zařízení. Procesor je obvykle mnohem rychlejší než ostatní hardware, a tak kdyby se zabýval pouze obsluhou periferie, byl by v danou chvíli nevyužit a většinu svého procesorového času by strávil ve smyčce čekáním na daný hardware. V dnešní době se přerušení využívá mnohem víc při přepínání procesů.

Přerušení v architekturách se vyvolá dvěma základními způsoby, Software-generated (softwarově) a External-hardware generated (hardwarově), tedy vyvolané vnějším okolím. To se dále rozděluje na NMI[3] nemaskovatelné a na INTR[4]. Softwarová přerušení vznikají v jediném případě, a to když procesor narazí na instrukci přerušení μC/OS-II se toto provádí inkrementováním *OSIntNesting* nebo použitím funkce *OSIntEnter*.

Když jsem v minulém odstavci definoval přerušení vyvolané mimo procesor, nebyla to úplně pravda, protože do hardwarového přerušení se řadí tzv. Internal HW, která vznikají při vnitřní chybě kódu, např. dělení nulou, špatná instrukce. Externě vyvolané přerušení nastává při žádosti ze vstupu (vnější hardware) nebo chybě matematického koprocesoru.

Co se tedy stane, když procesor obdrží žádost o přerušení? Nejprve zkontroluje, jestli se daný signál bude maskovat a pokud ne, přeruší právě

[2]přerušení
[3]Non Maskable Interrupt
[4]Interrupt Request

vykonávaný program a začne vykonávat program obsluhy přerušení (Interrupt Service Routine). Provede se výběr obsluhy podle vektorů přerušení (u většiny microprocesorů), uloží se kontext probíhající úlohy a upozorní se jádro na vstup do obsluhy přerušení. Následně se provede uživatelské obsloužení přerušení. Po jeho ukončení se informuje jádro o výstup z ISR, obnoví kontext CPU a pokračuje se v úloze dál.

V µC/OS-II je opět pouze definována obsluha přerušení. Její implementace závisí též na jednotlivé architektuře.

4.2.1 Periodické přerušení

Je speciální případ přerušení, většinou sloužící k odměřování počtu instrukcí, době vykonávání apod.. Doporučená periody je od deseti do sta opakování za jednu sekundu. Čím častěji jsou vyvolávána přerušení, tím více to vede ke zpomalení systému. Na druhou stranu jsou pravidelná přerušení, která jsou obsloužena hardwarově, např. v µC/OS-II je takto značeno *clock tick*, česky často označované jako hodinové přerušení a je závislé na hardwarovém časovači procesoru.

Vývojář systému má díky tomu k dispozici způsob jak změřit časové údaje. Umožňuje mu také na jednotky času nebo počet hodinový přerušení uspat činnost úlohy. Také to umožňuje držet informaci o době běhu systému, např. při použití 32b čítače jsme schopni pro frekvenci 100Hz odměřit 497 dní v kuse. Nastavení je součástí souboru *os_cfg.h* s nastavením konstanty *OS_TICKS_PER_SEC*.

4.3 Event Control Block

Zatím jsme se věnovali úplným základům a vynechali jsme některé velmi důležité schopnosti. O doplnění funkčnosti systému se stará blok ECB. Tento blok slouží vláknům ke komunikaci mezi sebou, k zajištění konzistentního přístupu ke sdíleným zdrojům pomocí semaforů a front. Pravidla použití ECB jsou prostá, jediný kdo může čekat na signál přes ECB, jsou vlákna, nikoli obslužné rutiny přerušení. Zaznamenávat signály do ECB však můžou jak vlákna tak ISR. Pokud nějaká úloha očekává signál v ECB, měla by být definována čekací doba.

Blok událostí je definován jako struktura obsahující pět údajů. Uvádět zde jak vypadá struktura v paměti, není důležité. Jelikož hlavní část, tedy tabulka komu se signalizuje, je stejná jako tabulka s ready úlohami. Výběr, odebrání a vložení signálu do struktury ECB je totožný s výběrem ready úloh podle priorit.

- typ události
- čítač v případě semaforu
- ukazatel na frontu nebo zprávu
- tabulku čekatelů
- index do tabulky

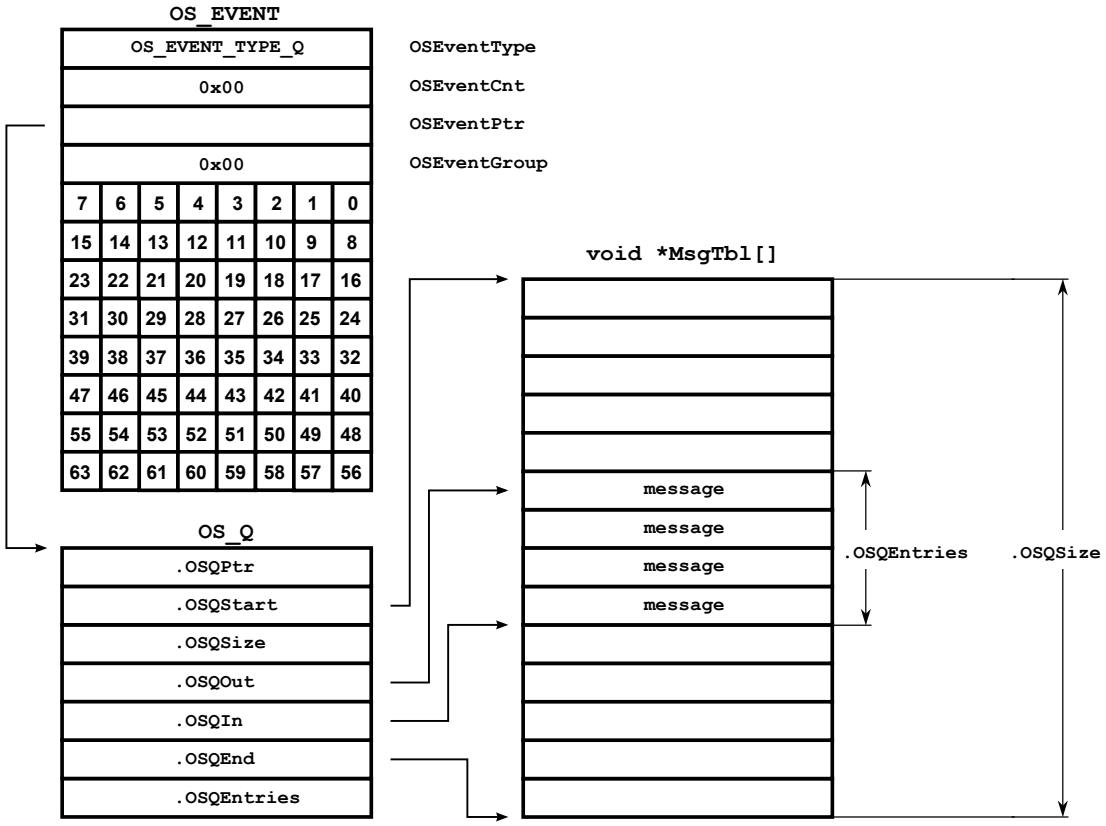
Úloha čekající na signál z ECB je vyjmuta z ready listu a je vložena do wait listu ECB. Bloků ECB je v systému víc stejně jako bloků TCB, jsou organizovány jako spojový seznam, jehož maximální délka je konfigurovatelná.

4.3.1 Speciální funkčnosti

Semafory jsou synchronizační prvky, převážně využité k synchronizaci procesů a samozřejmě i vláken. Semafor je prvek skládající se z celočíselného čítače a operacemi nad semaforem. Hodnota čítače se pohybuje na vymezeném intervalu, kde jedna strana označuje nulový počet přístupů a jedno plný počet přístupů a blokuje úlohy. Zažitá konvence říká, že čítač nabývá hodnot *0 až N*, kde N kde N je kladné celé číslo označující např. omezení velikosti fronty.

Zajímavý je např. semafor binární, tedy s čítačem 0 až N, který se dá využít k vzájemnému vyloučení úloh. Takovýto semafor nazýváme zkráceně *Mutex*.

Další co vývojáři ECB umožňuje je použití zpráv. K dispozici máme možnost předávat jednu zprávu použitím *OS_EVENT_TYPE_MBOX* nebo několik do fronty *OS_EVENT_TYPE_Q*. Obojí předávání stran umožňuje vybírání zpráv pomocí blokování úlohy při prázdné frontě, tak i pouze vybrání bez blokování pokud je fronta prázdná. Rozdíl mezi použitím jedné zprávy a fronty je v použití ukazatele OSEventPtr v OE_EVENT, v případě fronty ukazuje na strukturu *OS_Q*, která udržuje informace o frontě.



Obrázek 4.3: Propojení ECB a OS_Q

- ukazatel na počátek fronty
- velikost fronty
- výstup fronty
- vstup fronty
- ukazatel na další frontu
- počet prvků ve frontě

Propojení struktury ECB s OS_Q je na obr.4.3.

μC/OS-II umožňuje bez výhrad použít výše uvedené synchronizační prvky. Programátorům poskytne funkce pro vytvoření a základní obsluhu. Navíc např. u semaforů umožňuje získat jejich datovou strukturu. Signalizaci úlohám skrze tyto prvky se provádí právě pomocí ECB.

4.4 Event-Flag

Použití ECB je mocný nástroj, místo něj lze v řadě případů použít na paměť a obsluhu méně náročné příznakové signalizování [5]. Signalizace se zde provádí nastavováním sérií bitů, udržujících informace o stavech. Datová struktura *OSFlagGroup* obsahuje

- typ signalizace (flag)
- ukazatel na seznam OSFlagNode
- OSFlagFlags držící status událostí

Struktura *OSFlagNode* určuje, která úloha a na co čeká. Struktury jsou mezi sebou propojené, dají se tak pomocí typu skládat podmínky pro čekající vlákna. Její struktura je

- odkaz na OSFlagGroup
- ukazatel na další OSFlagNode (předka a následovníka)
- ukazatel na TCB čekající úlohy
- vzor signalizace bitů
- typ struktury OR/AND (ALL/ANY)

Vytváření skupiny signalizace mohou jen vlákna, nikdy se nesmí stát, aby se vytvářeli v ISR. Po vytvoření je skupina prázdná a je nutné nastavit její parametry. Seznam podmínek je možné vytvořit připravením struktur. Taktéž signalizováním blokování, když se narazí na funkci *OSFlagPend()*, jejíž parametry jsou data struktury OSFlagNode. Po odblokování úlohy je nutné ji odebrat z listu čekatelů a označit vlákno za ready.

4.5 Správa paměti

Vývojář aplikací v C/C++, který běžně používat dynamické alokování a uvolňování paměti pomocí malloc() a free(), by si měl uvědomit následující. Použití těchto funkcí je velmi nebezpečné, tyto funkce zajišťují velikost

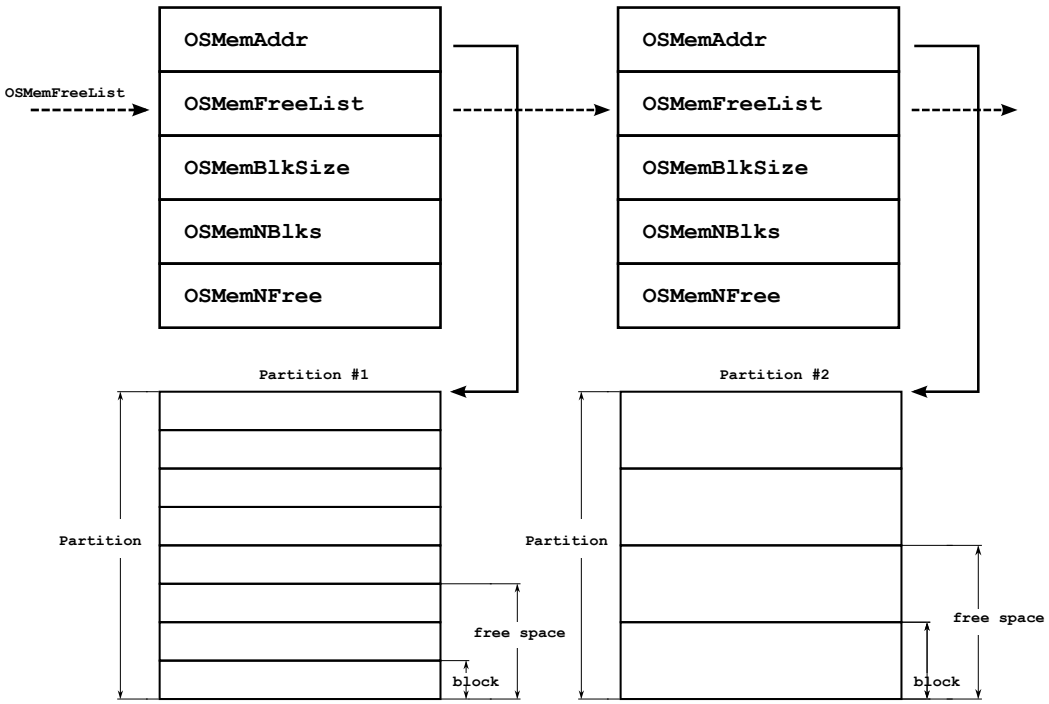
[5] Známější anglické pojmenování Event-Flag

alokované paměti. Co však nezajišťuje, je souvislost oblasti v paměti. Takto alokovaná oblast může být fragmentovaná, to znamená, že oblast není alokovaná v jednom kuse, ale je roztroušena po malých částech do volných oblastí. Navíc k tomu, je doba alokace a dealokace nedeterministická, právě díky fragmentaci.

μC/OS-II poskytuje mechanismus zajišťující deterministický čas vykonání funkcí alokace a uvolnění paměti. Aby toto bylo možné, musí být blok v souvislé části paměti. K tomu slouží *partition* obsahující bloky o zvolené velikosti. Struktura udržující informace se nazývá *memory control block*.

- ukazatel na začátek
- ukazatel na další strukturu
- velikost bloku
- počet bloků
- volné bloky

Na následujícím obrázku 4.4 vidíme náznak propojení MCB ve spojení seznamu a také je zde naznačeno použití bloků a zbylého volného místa.



Obrázek 4.4: Alokace bloků paměti

5 Plánování procesů

Jedná se o základní schopnost jader operačních systémů. V případě RTOS je důraz kladen na časová omezení úloh. V následující části budou vysvětleny důležité pojmy a algoritmus plánování v μC/OS-II. Posléze se dostaneme k pokročilejším, které jsme implementovali.

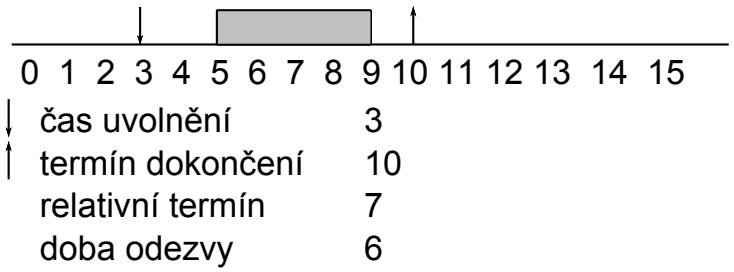
5.1 Preemtivní/Nepremtivní plánování

Pokud je povoleno přerušení úlohy plánovačem, nazýváme plánování preemtivní. Posléze jí byl odebrán čas CPU a byla spuštěna úloha s aktuálně nejvyšší prioritou. Přerušené úloze je nastaven stav připraven a čeká, než bude vybrána k běhu. Preemptivní plánování je možné pouze u preemptivních úloh.

U nepreemptivního plánování nedochází k přerušení spuštěné úlohy plánovačem. Nepřerušení však může vést k chybám v časování, které preemptivní přístup odstraní, což je nevýhodou tohoto přístupu. Takovýto úhel pohledu je z hlediska přístupu k důležitým prostředkům výhodnější pro jednoprocesorové RT architektury, protože nevyžaduje speciální mechanismus přístupu do kritické sekce ani pro řazení úloh do front.

5.2 Prediktivní synchronizace

Tím je myšlena schopnost více procesové komunikace a predikování času. Když proces sdílí prostředky, je velmi často nutná synchronizace kvůli integritě dat. Když nějaký proces používá sdílený zdroj, jiný musí čekat. Tento čas není předem známý. Řešením může být například maximální trvání semaforu, které ale negarantuje přístup kritického procesu k nekritickému. Další používanou technikou je neblokující synchronizace využívající fronty FIFO, tím je deterministicky určen nejhorší čas přístupu ke sdíleným zdrojům.



Obrázek 5.1: Doba dávky

5.3 Situační model

Proces kontroly počítače provádí jeden nebo více kontrolních a monitorovacích funkcí. Každá funkce je asociována s jedním nebo více úkoly. Některé z těchto úkolů jsou prováděny v reakci na události v zařízení nebo sledují systém. Ostatní jsou prováděny v reakci na události v jiných úkolech. Žádný z úkolů nemůže být proveden před zahájením akce, která jej vyvolává. Každý z úkolů musí být dokončen před dobou, nežli je opět vyvolán plánovačem.

5.3.1 Úlohy

Úlohy [1] jsou jednoduché programy, které mají CPU v daný čas pro sebe. Návrhový proces real-time aplikací vyžaduje rozdělení práce do úloh po částech problému. Každý úkol má svou prioritu vykonávání, vlastní hodnoty registrů CPU, vlastní zásobník. Úlohy bývají často realizovány nekonečnou smyčkou a mobou být v různých stavech, příklad viz obr.4.1.

Instancí úlohy je dávka[2]. Dávky potřebují k běhu zdroje, jako je např. CPU, síť, kritická sekce. Často se hardwarové zdroje nazývají "procesory". Časový průběh dávky je na obr.5.1.

Kromě programového kódu události, které určuje programátor, je nutná

[1] v ang. lit. task

[2] v ang. lit. označováno job

znát tyto časové údaje úlohy τ_i:

- r_i - Čas uvolnění dávky připravené k běhu.
- C_i - Termín nejdelšího běhu úlohy
- D_i - Relativní termín dokončení dávky
- T_i - Perioda volání úlohy (pouze pro periodické).

Kvalita plánování úloh závisí na přesnosti výše napsaných údajů. Dále bychom měli počítat s délkou vykonávání operací přepnutí kontextu úlohy, volání jiných obslužných funkcí jádra.

5.3.2 Čistě cyklické

Takto nahlížíme na aplikace, které spouští každou úlohu periodicky. Mají téměř neměnné požadavky na systémové prostředky, např. digitální regulátor, řízení letu a jiné.

5.3.3 Převážně cyklické

Většina úloh je spouštěna periodicky. Systém reaguje na nějaké externí asynchronní události, např. moderní avionické a řídicí systémy.

5.3.4 Asynchronní a predikovatelné

Požadavky na systémové prostředky se mohou během po sobě jdoucích spuštění velmi lišit, ale mají známé meze. Většina úloh není periodická, např. multimediální komunikace, zpracování signálu z radaru, sledování objektů a další.

5.3.5 Asynchronní a nepredikovatelné

Převážně reagují na asynchronní události. Kde obsluha je velmi složitá, např. real-time simulace, real-time řízení atp.

5.3.6 Činitel vytížení CPU

V tomto bodě si ukážeme horní mez využití procesoru. Definujeme (procesorové) vytížení, jako času procesoru strávený nad sadou úloh. Jinými slovy je to také jedna mínus doba nečinnosti procesoru. Tedy $\frac{C_i}{T_i}$ je zlomek času procesoru strávený vykonáváním úkolu i pro n úkolů, faktorem vytížení je:

$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$

Přestože faktor vytížení lze zlepšit zvýšením hodnot C_i, nebo snížením hodnoty T_i. Jsme limitováni horní požadavkem, aby všechny úlohy a jejich lhůty na kritické okamžiky byly splnitelé. Jak je zajímavá dolní hranice vytížení procesoru nechám na vašem uvážení. Naopak velmi zajímavým tématem je tedy "Jak velké musí být vytížení procesoru, aby byla sada úloh plánovatelná?". V průběhu textu se vrátíme k tématu za použití výsledků práce věnující se statickému přidělování priorit [CLLiu], k porovnání nasimulovaných hodnot s teoretickými hodnotami některých algoritmů.

5.4 Obsluha přerušení

Mechanismus plánování obsluhy přerušení je pro RTOS klíčovou vlastností. Přerušovací systém má své vlastní priority, které jsou určeny samotným hardwarem. Priority obsluhy přerušení jsou vyšší než priority uživatelských procesů. Pravidla RTOS určují, že obsluha přerušení musí být tak krátká, jak je to jen možné. Přesto spousta zařízení potřebuje delší dobu k prvádění akcí po skončení přerušení, ty pak přecházejí pod ISR.

5.5 Statické rozvrhování

Někdy označované jako offline plánování. Rozvržení úloh zde probíhá ještě před jejich spuštěním. Vyhodnotí se jak časové tak i datové závislosti. Každý proces je naplánován tak, aby byl vykonáván co nejdříve. Výsledný plán bývá realizován listem plánu, při jeho tvorbě může být použito nejrůznějších heuristik. Nevýhodou tohoto přístupu, která předpokládá stejné parametry úloh a neumožňuje přizpůsobení se změnám.

5.5.1 Rate Monotonic

Mechanismus v literatuře označovaný jako Rate Monotonic je statický algoritmus. Vychází z předpokladu, že úlohy volané nejčastěji mají významnější prioritu. Významnost priority je tedy přímo úměrná kmitočtu, odpovídajícího její periodě. Nepřímo úměrná je tedy statickému parametru T úlohy.

Jestliže není mechanismus schopen zajistit plánovatelnost úloh, jsou řešením problému buď to změny parametrů úloh nebo použití jiného plánovacího algoritmu než RM. Změny parametrů úloh vedou na změny parametrů celého systému, což je většinou nepřijatelné. Příčinou selhání tohoto algoritmu může být skutečnost, že mechanismus RM je dobrý pro množinu úloh s parametry $D = T$ a jejichž priorita se nemění v čase.

5.5.2 Deadline Monotonic

Je mechanismus statického přiřazování priorit nepřímo úměrně hodnotě časové meze označovaný zkratkou DM, popřípadě IDA (Inverse Deadline). Vychází z předpokladu, že úlohy s menší hodnotou časové meze jsou významnější. Významnost úlohy je v tomto případě nepřímo úměrná hodnotě statického parametru D úlohy.

Tento mechanismus je schopný dobře plánovat i úlohy na množině $D < T$, tj. celkově na množině $D <= T$. Tzn., že algoritmem DM jsme schopni naplánovat i úlohy, které nejsou plánovatelné pomocí RM.

5.6 Dynamické rozvrhování

Implementačně složitější je plánování úloh za běhu systému. Kdy dochází k jejich analýzám a plánování podle výsledků, či dle podnětů z okolí. K přeplánování a změně priorit dochází podle parametrů algoritmů. Výhodou tohoto přístupu je právě neustálá schopnost se přizpůsobovat změnám. Tyto metody jsou vhodné zejména pro aperiodické a sporadické úlohy. Z velmi početné množiny jsme vybrali jednodušší EDF a LLF.

5.6.1 Earliest Deadline First

Z posledního statického plánování bychom mohli vyvodit první dynamické. Mechanismus EDF, jak již název naznačuje, určuje priority nepřímo úměrně době zbývající v čase t do uplynutí časové meze. Garantuje, že významnější priorita je v čase t přiřazena těm úlohám, kterým zbývá do uplynutí doby splnění méně času.

5.6.2 Least Laxity First

Algoritmy překonávající vlastnosti EDF samozřejmě existují, jedním z nich je LLF. Tento algoritmus přiřazuje priority úlohám podle volnosti úlohy v čase t. Dobou volnosti je myšleno, na jak dlouho může být úloha odložena, aniž by překročila svou časovou mez. Nevýhodou je častější přepínání kontextu a větší počet preempcí.

6 Implementace

V této kapitole si ukážeme, jak proběhlo doimplementování vybraných plánovacích algoritmů. Snahou bylo co nejvíce využít stávajícího kódu, velikost rozšíření by měla být rovněž přijatelná. Postupně zde probereme úpravy potřebné pro kompilaci kódu, poté zakládání úloh podle modelu, rozšíření task control blocku. Poté se budeme věnovat implementaci algoritmů výběru a nakonec změně priorit a následnému výběru spuštění.

6.1 Modifikace zdrojových kódů

Bez zásahu do originálního kódu se neobejdeme, avšak jediné nad čím bychom úpravy měli provádět, jsou pouze části kódu definující překládané kousky systému. Výkonný kód by měl zůstat beze změn a nezavedeme do něj chyby vedoucí k neočekávanému chování. Po analýze zdrojových kódů, je evidentní, že budeme modifikovat soubory *os_core.c* a přidáme konfigurační makra do *os_cfg.h*.

Ideální by bylo samozřejmě neměnit původní kód vůbec, taková situace v jazyce C není možná. Potřebovali bychom na základě definování rozšíření v konfiguraci přetížení původních funkcí, funkcemi vytvořenými. V programovacím jazyce C, neexistuje přetěžování funkcí a metod, proto abychom zachovali kódovou konvenci, vytvoříme další v souboru *os_cfg.h* makro.

```
                        /* ————— SCHEDULING MANAGEMENT ————— */
#define OS_SCH_TYPE  0  /* Type of task scheduling (0-4)           */
                        /*     MicroC/OS-II scheduling        (0) */
                        /*     Rate monotonic scheduling      (1) */
                        /*     Deadline monotonic scheduling  (2) */
                        /*     Early deadline first scheduling (3) */
                        /*     Least laxity first scheduling  (4) */
```

Makro definuje volbu typu plánování. V celku jednoduše a elegantně jsme získali jednoduché volby mezi původním plánováním a vlastním rozšířením. Nyní je nutné označit makrem funkce, které budeme nově navrhovat, podmínkou zachycenou preprocesorem k podmíněnému překladu.

```
#if OS_SCH_TYPE == #ZVOLENY TYP PLANOVANI
/* zde bude funkce*/
#endif
```

Podrobnější popis přidání zaimplementování podmínek pro kompilaci je součástí přílohy A.

6.2 Možnosti jak vytvořit úlohu

Závislosti na původních funkcích μC/OS-II máme obalené podmínkami překladu. V kapitole 5 jsme si definovali časové parametry úloh. Časové parametry si zde nemůžeme představit jako veličinu času, tedy nezadávají se např. sekundy ani minuty. V rozšíření je označujeme časovým údajem počet ticků systému. V podstatě jsou dva způsoby jak zavést parametry do systému a to

- napsat nové funkce na vytváření úloh s parametry
- využít OSTaskCreateExt

6.2.1 Vlastní funkce na vytváření úloh

Napsat nové funkce pro vytváření úloh není nejvhodnější řešení. Funkce by v podstatě kopírovali funkčnost stávajících, k tomu by zavedli do systému parametry. Jejich přidání by znamenalo vyřešení problému kam a jak nejlépe je zařadit. Zda je přidat např. jako samostatný seznam podle priorit úloh. To by zřejmě nebylo dobré, vedlo by to k zatěžování systému při změně časových parametrů ve spojení se změnou priorit. Z tohoto hlediska se jako lepší řešení jeví rozšíření task control blocku.

6.2.2 Rozšíření TCB

K čemu task control block slouží, jsme si uvedli v kapitole 4. Již v původních kódech je struktura TCB vytvořena tak, aby byly voleny jednotlivé části v závislosti na konfiguraci v os_cfg.h. To mimo jiné dokáže uspořit značnou část využité paměti mikroprocesoru v případě, že nejsou potřeba.

V tomto případě by se více než hodilo přidat další hodnoty do stávajícího TCB.

```
typedef struct os_tcb {
  OS_STK   *OSTCBStkPtr;
    /* Pointer to current top of stack                    */

#if OS_TASK_CREATE_EXT_EN > 0
  void     *OSTCBExtPtr;
    /* Pointer to user definable data for TCB extension */
  OS_STK   *OSTCBStkBottom;
    /* Pointer to bottom of stack                         */
  INT32U    OSTCBStkSize;
    /* Size of task stack (in number of stack elements) */
  INT16U    OSTCBOpt;
    /* Task options as passed by OSTaskCreateExt()       */
  INT16U    OSTCBId;
    /* Task ID (0..65535)                                 */
#endif

  struct os_tcb   *OSTCBNext;
    /* Pointer to next     TCB in the TCB list           */
  struct os_tcb   *OSTCBPrev;
    /* Pointer to previous TCB in the TCB list           */
.
.
.

#if OS_TASK_NAME_SIZE > 1
  INT8U     OSTCBTaskName[OS_TASK_NAME_SIZE];
#endif

/* Extended Scheduling RM, DM, EDF, LLF */
#if OS_SCH_TYPE != 0
  INT16U   OSTCBR;
    /* Arrival time of task */
  INT16U   OSTCBC;
    /* Maximum time task */
  INT16U   OSTCBD;
    /* Task Deadline */
  INT16U   OSTCBT;
    /* Task Period */
  INT16U   OSTCBCur;
    /* Task current time */
#endif
} OS_TCB;
```

V ukázce zdrojového kódu jsou všechny vlastnosti našeho modelu, spuštění úlohy, doba opakování, maximální doba vykonávání a mezní termín splnění. Pro některé algoritmy je vhodné udržovat hodnotu, kolik již úloha strávila času vykonáváním. Všimněme si obalení podmíněným překladem, který tyto informace přidá v případě použití jiného než výchozího plánování.

6.2.3 Využití OSTaskCreateExt

Předchozí varianta počítá s malým, ale přeci jen zásahem do zásadní části systému. Tomu jsme se při implementaci rozšíření chtěli vyhnout. Z toho důvodu jsme zvolili variantu číslo dvě. Využít funkci *OSTaskCreateExt*. Tato funkce je rozšířením *OSTaskCreate* pro zavedení úloh do μC/OS-II. Využít rozšířenou variantu je doporučeno pro práci s běžným systémem kvůli větší variabilitě předávání dat úlohám.

6.2.4 Standartní parametry

task	Ukazatel na výkonný kód úlohy.
pdata	Je ukazatel na data předávané jako parametr úloze. Například vlákno obsluhující asynchronní sériový port, pomocí něj může informovat o rychlosti přenosu, chybách a jiné obsluze portu.
ptos	V tomto parametru je ukazatel na vrchol zásobníku. Do zásobníku se ukládají lokální proměnné, parametry funkcí, návratové adresy a registry CPU pro obnovu po přerušení.
prio	Určuje počáteční, unikátní prioritu vlákna. Čím vyšší číslo obsahuje, tím menší prioritu úloha má.
id	Označení číselným identifikátorem vláken.
pbos	Ukazatel na počátek nebo také dno zásobníku.
stk_size	Velikost zásobníku úlohy ve zvolené šířce, při použití INT8U je to počet bytů, při použití INT16U se jedná o počet dvou bytových bloků atd.
opt	Šestnácti bitové nastavení, z čehož dolních 8 bitů používá μC/OS-II a horních 8 bitů je určeno uživateli.
pext	V této proměnné můžeme mít ukazatel na uživatelskou datovou strukturu. Tomuto pointru bude věnován následující kapitola 6.2.5.

Nastavení řady parametrů je záležitostí konfigurace systému, k tomu dobře poslouží [MicroII]. Ukázkové použití metody pro zavedení úlohy obsahuje B spolu s nastavením a použitím rozšíření.

6.2.5 Parametry modelu úlohy

K zavedení parametrů úlohy použijeme parametr *pext*. Obecně slouží k předávání uživatelsky definovaných oblastí paměti, typicky datových struktur. Nic nám tedy nebrání definovat si vlastní strukturu

```
typedef struct os_sched_ext {
  INT16U          OSTCBStartTask;
  INT16U          OSTCBMaxTime;
  INT16U          OSTCBDeadline;
  INT16U          OSTCBPeriod;
  /*      parameters in time          */
  INT32U          OSTCBLastWork;
  INT32U          OSTCBRan;
  INT32U          OSTCBStartPeriod;
  BOOLEAN         OSTCBDeadlineMiss;
  INT32U          OSTCBFirstDeadline;
  INT16U          OSTCBLeastLaxity;
  /*      user data                   */
  void           *pext;
} OS_SCHED_EXT;
```

V této struktuře udržujeme časové údaje úlohy, k tomu ještě stále můžeme přidat uživatelská data. Důležité je donutit uživatele aby ji použil a zvykl si, že úroveň s uživatelskými daty je o jedno zanoření dále. Časové parametry úloh jsou důležité pro samotné plánovací algoritmy, a proto spoléhají na existenci této struktury. Důležité je, aby i systémové úlohy obsahovaly dodatečné časové parametry.

V μC/OS-II je obvykle několik systémových úloh. Pro funkčnost plánování je nutné určit jakou prioritu přiřadit systémovým vláknům. Situace se usnadňuje tím, že systémové úlohy jsou buď to nejvyšší priority, anebo nejnižší. Uvažme nejprve nejnižší, vlákna *idle_task* a *stat_task*. Mají pevně dané priority, proto můžeme snadno přidat strukturu s časovými parametry. Inicializační funkce musí zjistit dodání správných hodnot, odkaz na správné TCB úlohy je daný systémovými konstantami.

```
INT8U  OSSchedExtInit (void)
{
...
ptcb = OSTCBPrioTbl[OS_TASK_TMR_PRIO];
ptcb->OSTCBExtPtr = (void *)pextT;
...
ptcb = OSTCBPrioTbl[OS_TASK_IDLE_PRIO];
ptcb->OSTCBExtPtr = (void *)pextI;
...
ptcb = OSTCBPrioTbl[OS_TASK_STAT_PRIO];
ptcb->OSTCBExtPtr = (void *)pextS;
}
```

Funkce pamatuje i na aplikace využívající úlohu systémového časovače, který běží s předem stanovenou prioritou. Aby bylo možné časové údaje měřit, musí být zprovozněn systémový čas. Tato část se provádí při portaci systému na vybranou architekturu, proto bylo využito portace na H8S [PJer].

6.3 Implementace algoritmů

V této fázi máme dle konfiguračního souboru možnost vybrat buď'to plánování původní, nebo externí. Abychom jej mohli zavést, bylo nutné v souboru *ucos_ii.c* uvést, o který soubor se jedná.

```
#include <os_sched_ext.c>
```

V nataženém souboru jsou shromážděny zásadní funkční bloky pro rozšířenou funkci plánování. Volbou konfigurace o jeho použití, jsme zajistili zkompilování vlastní, téměř totožné funkce *OS_SchedNew()*. Její smysl spočívá ve vybrání vlákna, které je ve stavu ready a má největší prioritu. Celou tuto metodu jsme přepsali do podoby využívající bloky, překládané na základě zvoleného algoritmu.

Výše uvedená funkce má za úkol projít všechny úlohy. Ověřit zda jsou ve stavu ready, zda nenesou informaci o uspání. Dále musí zajistit aktuální časové údaje a vybrat nejdůležitější úlohu. Ke kontrole parametrů úloh použijeme TCB seznam.

```
OS_TCB     *ptcb;
ptcb = OSTCBList;
while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {
  OS_ENTER_CRITICAL();
  ...
  ptcb = ptcb->OSTCBNext;
  OS_EXIT_CRITICAL();
}
```

V seznamu procházíme jednu úlohu po drohé, z bloku na který máme ukazatel pext, načteme potřebné údaje pro vybraný algoritmus. V zásadě není rozdíl mezi volbou dynamického nebo statického plánování, rozdíly mezi zvoleným algoritmem jsou v kódu spíše kosmetické.

6.3.1 Statické plánování

Rate monotonic scheduling a deadline monotonic scheduling je možno pojmout dvě způsoby. Jednou z možností jak vyřešit výběr úloh, je seřazení úloh. A to při prvním volání plánovače, podle statických parametrů. Nevýhoda tohoto řešení je zcela evidentní. Jednak by se jednalo o počáteční zdržení, díky schopnosti implementovat řadící algoritmy složitosti $NlogN$, jakým je např. řazení haldou. Ale co je v případě zařízení využívajících μC/OS-II důležitější, je potřeba větší paměti k seřazení priorit. Z tohoto důvodu bylo přikročeno k výběru ready úlohy při revizi údajů o průběhu úlohy.

```
// RATE MONOTONIC parameters
if (( ptcb->OSTCBStat | OS_STAT_RDY) == OS_STAT_RDY
   && (ptcb->OSTCBDly == 0 )){
  if (os_task_first == (void *) 0){
    os_task_first = ptcb;
    first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
  }else{
    if ( pext->OSTCBPeriod < first_pext->OSTCBPeriod ){
      os_task_first = ptcb;
      first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
    }
  }
}
...
// DEADLINE MONOTONIC parameters
if (( ptcb->OSTCBStat | OS_STAT_RDY) == OS_STAT_RDY
   && (ptcb->OSTCBDly == 0 )){
  if (os_task_first == (void *) 0){
    os_task_first = ptcb;
    first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
  }else{
    if ( pext->OSTCBDeadline < first_pext->OSTCBDeadline ){
      os_task_first = ptcb;
      first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
    }
  }
}
```

6.3.2 Dynamické plánování

Dynamické plánování s použitím EDF, ukládá do parametrů úloh mezní čas splnění, v závislosti na čase. Jeho hodnota se v čase mění a spočte se $D(t) = Time_{run} + D_{stat}$ $Time_{run}$ je doba, kdy úloha vstupuje do děje a je tedy označena jako ready. Nezbývá nám tedy než určit dobu vstupu do děje. Nejjednodušší a námi použitý způsob je $OS_TIME + T$, kde OS_TIME je aktuální doba strávená během systému a k ní připočtená perioda. Abychom se nepřipravili o první periodu běhu úloh, řekneme, že první doba vstupu, je

právě ta doba, kdy se ji pokoušíme poprvé určit. Abychom co nejméně omezili rychlost systému, jsou časové parametry dopočteny těsně před porovnáním.

```
// EARLY DEADLINE FIRST parameters
if (( ptcb->OSTCBStat | OS_STAT_RDY) == OS_STAT_RDY
   && (ptcb->OSTCBDly == 0 )){
  if (os_task_first == (void *) 0){
    os_task_first = ptcb;
    first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
  }else{
    if ( pext->OSTCBFirstDeadline < first_pext->OSTCBFirstDeadline ){
      os_task_first = ptcb;
      first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
    }
  }
}
...
//LEAST LAXITY FIRST parameters
pext->OSTCBLeastLaxity = pext->OSTCBLeastLaxity - OSTime - pext->OSTCBRan;

if ( (ptcb->OSTCBStat | OS_STAT_RDY) == OS_STAT_RDY
  && (ptcb->OSTCBDly == 0 )){
  if (os_task_first == (void *) 0){
    os_task_first = ptcb;
    first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
  }else{
    if ( pext->OSTCBLeastLaxity < first_pext->OSTCBLeastLaxity ){
      os_task_first = ptcb;
      first_pext = (OS_SCHED_EXT *)(os_task_first->OSTCBExtPtr);
    }
  }
}
```

Obdobně se určují hodnoty pro algoritmus LLF $L(t) = D(t) - C(t)$. Hodnotu deadline v závislosti na čase umíme určit, chybí nám tedy doba, jak dlouho už úloha zpracovávala svůj úkol. To provedeme odečtením aktuální doby běhu systému od doby, kdy bylo vlákno přerušeno. Jak je toto prováděno, bude popsáno v kapitole 6.4. Navíc jsme uživateli dali možnost zjistit, zda některá úloha nestihla svou deadline.

6.3.3 Změna priorit

Nyní jsme ve fázi, kdy umíme zjistit parametry úloh a kdy podle nich jsme schopni určit úlohu ke spuštění. Právě v této části zpracování jsme schopni dodat uživateli informace o minutí deadline. Známe totiž poslední úlohu ve stavu ready a i tu kterou se chystáme spustit.

```
if ( OSTime == 0 ){
  OSTCBHighRdy = os_task_first;
  OSPrioHighRdy = os_task_first->OSTCBPrio;
}else{
```

```
  OS_SchedDataEnd( OSTCBHighRdy );
  OSTCBHighRdy = os_task_first;
  OSPrioHighRdy = os_task_first->OSTCBPrio;
}
OS_SchedDataStart( OSTCBHighRdy );
```

```
void OS_SchedDataStart( OS_TCB *HighRdy ){
  OS_SCHED_EXT     *pext;

  HighRdy->OSTCBStat |= OS_STAT_RDY;
  pext = HighRdy->OSTCBExtPtr;
  pext->OSTCBLastWork = OSTime;
}

void OS_SchedDataEnd( OS_TCB *HighRdy ){
  OS_SCHED_EXT     *pext;

  pext = HighRdy->OSTCBExtPtr;
  pext->OSTCBRan = OSTime - pext->OSTCBLastWork;

  if ( OSTime > pext->OSTCBFirstDeadline ){
    pext->OSTCBDeadlineMiss = OS_TRUE;
  }

  HighRdy->OSTCBStat |= OS_STAT_SUSPEND;
}
```

Stejně jako v případě statického, tak při použití dynamického plánování je situace jasně dána. Není třeba měnit priority v systému. Při průchodu TCB listem vyhledáme nejmenší hodnotu $D(t)$ nebo v případě LLF $L(t)$. Namísto aby byla priorita aktuálně nejdůležitější úlohy vybrána pomocí mapy priorit, vybranou prioritu nastavíme do globální proměnné *OSPrioHighRdy*.

6.4 Sběr parametrů úloh

Změnit dle priority úlohu umíme, zbývá určit data, podle kterých se tento proces děje. Základní parametry, tedy periodu spouštění, vstup do periody, mezní termín a maximální vykonávání úlohy, pro statické i dynamické algoritmy musí zadat vývojář aplikace. K tomuto účelu byla vytvořena inicializační funkce. Její výhodou je hlavně odpadnutí vynulování alokované paměti, na kterou vývojáři často zapomínají.

```
static void OS_SchedPext( OS_SCHED_EXT *pext ,
             INT16U startTask, INT16U maxTime,
             INT16U deadline, INT16U period );
```

Jeden z nejzajímavějších úkolů této práce, je sběr informací pro plánovací algoritmy. Do jaké struktury jsou data sbírána, jsme si uvedli v kapitole

6.2.5. Tato struktura se musí průběžně aktualizovat pro každé vlákno. Je tedy zřejmé, že se data budou sbírat při práci úlohy. Samozřejmě bychom mohli dát toto privilegium uživateli, to by způsobilo několik problémů, jednak není jisté, že by uživatel správně aktualizoval údaje. A hlavně co je důležitější, uživatelská úloha může být náhodně přerušena. Pokud bychom např. sbírali data na začátku, uprostřed a na konci smyčky. Dostávali bychom jen statistické údaje, čím déle by systém fungoval, tím lepší bychom získali.

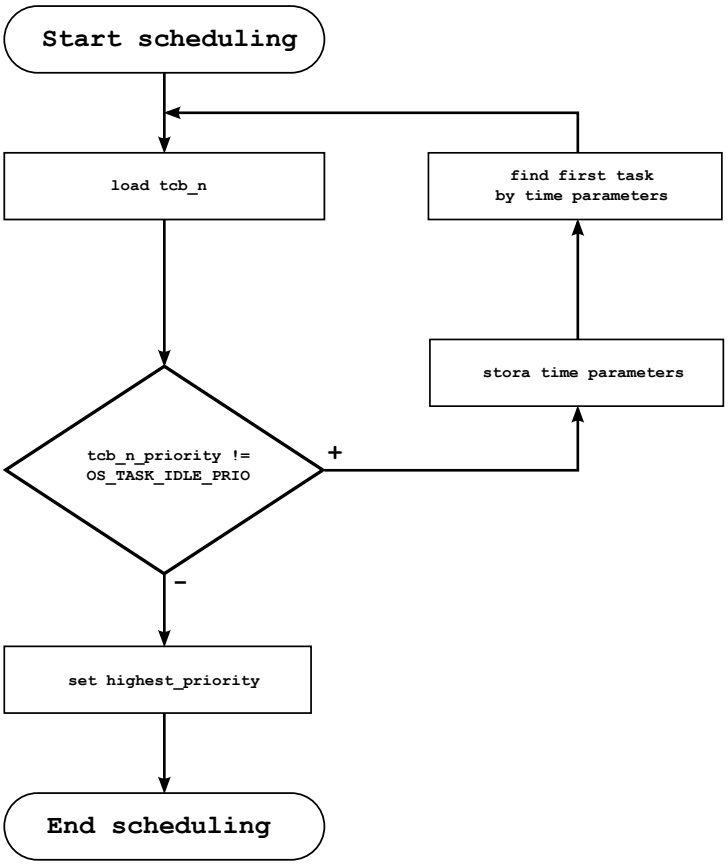
Ač by se jednalo o velmi jednoduchou implementaci, byla by velmi nevhodná. Lepší volbou je implementace do systému. Všimněme si, že po naplánování úloh, dochází k přepnutí kontextu procesoru. To je také ve chvíli, kdy víme, jaká úloha pracovala a jaká bude pracovat. Zde máme bud'to variantu bez zásahu do souboru *os_core.c* nebo s malým zásahem.

Totiž při portaci μC/OS-II se musí vytvářet funkce *OSIntCtxSw*, ta je obvykle napsána v jazyce symbolických adres. Slouží k vytvoření softwarového přerušení pro přepnutí úloh. Spolu s ní se vytváří prototypy dalších deseti uživatelských funkcí, z nichž *OSTaskSwHook* je volána z OSIntCtxSw. Podmínkou by tedy bylo, aby vývojář aplikace vždy do dané funkce umístil naši *OS_ShedDataStart*, která by uložila dynamické parametry přerušené úlohy. Nové úloze by se počáteční hodnoty nemohli nastavit a museli bychom v jedné funkci zjistit prioritu nové úlohy a jí nastavit počáteční parametry. Hlavní nevýhodou by byla nutnost zásahu uživatele.

Naproti tomu stačí mírně upravit soubor os_core.c, přesněji do funkce OS_Sched a také do funkce ukončení obsluhy přerušení *OSIntExit*, přidáním podmínky pro překlad použití dynamického plánování. Podrobnosti modifikace viz příloha A.

```
#if OS_SCH_TYPE == 0
   OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
#else
   OS_SchedDataEnd( OSTCBHighRdy );
   OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
   OS_SchedDataStart( OSTCBHighRdy );
#endif
```

Důležité pro sběr dat je abychom neužívali operátorů násobení a dělení, ač mikroprocesory čím dál častěji obsahují sofistikovanější aritmeticko-logické jednotky, patří stále k nejpomaleji prováděným operacím. Kde by nás to zřejmě velmi svádělo, jsou výpočty aktuálních deadline a period, respektive vstupů úloh do děje. Například vypočtení aktuální deadline $d_act = pext- > OSTCBPeriodStart + pext- > OSTCBDeadline$). Bylo tedy nutné zařídit, aby se vždy správně dopočetl začátek periody a to i v případě, že dojde k



Obrázek 6.1: Pohled na implementované plánování

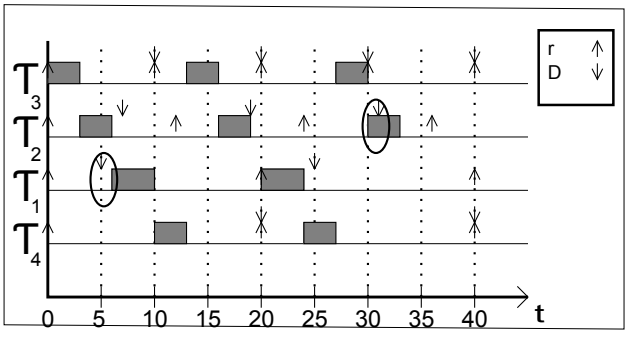
překročení deadline.

Následující vývojový diagram na obr.6.1 statického plánování ukazuje celý cyklus.

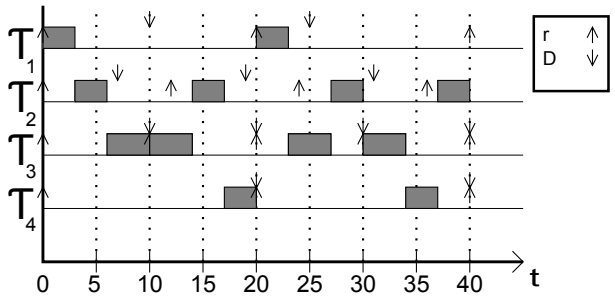
7 Zhodnocení řešení

Ještě než se přesuneme k porovnání práce, ukažme si jednoduchý příklad. Uvažme plánování Rate Monotocnic mechanismem, pouhé čtyři úlohy s parametry první úlohy $\tau_0(r_0, C_0, D_0, T_0) = \tau_0(0, 3, 5, 20)$, druhé $\tau_1(r_1, C_1, D_1, T_1) = \tau_1(0, 3, 7, 12)$, třetí $\tau_2(r_2, C_2, D_2, T_2) = \tau_2(0, 4, 10, 10)$ a čtvrté $\tau_3(r_3, C_3, D_3, T_3) = \tau_3(0, 3, 20, 20)$. Podle definice RM budou úlohy seřazeny prioritně od nejvýznamější po nejméně důležitou v pořadí $\tau_2, \tau_1, \tau_0, \tau_3$ nebo $\tau_2, \tau_1, \tau_3, \tau_0$. Plán generovaný na základě priorit je vyobrazen na obrázku 7.1, kde v čase $t = 0$ vzniknout požadavky na současné vyvolání všech čtyř úloh. Protože úloze τ_2 je přiřazena nejvýznamnější priorita, poběží jako první právě tato úloha. Zbylé úlohy přejdou do čekajícího stavu. Po dokončení úlohy τ_2 v čase $t = 0 + C_2 = 4$ může začít běžet úloha τ_1, která ze zbývajících má nejvyšší prioritu. Poté již může běžet úloha τ_3 nebo τ_0. Avšak bez ohledu na pořadí posledních dvou jmenovaných úloh, překročí τ_0 svou časovou mez již v čase $t = 5$, zatímco spuštění úlohy je podle přiřazených priorit mechanismem RM nejdříve v čase $t = 7$.

V případech není-li mechanismus RM schopen zajistit plánovatelnost množiny úloh RTOS, jak je tomu výše, vzniká problém řešitelný dvěma základními způsoby. První způsob spočívá ve změně parametrů RT úloh s cílem zajistit plánovatelnost dané množiny s modifikovanými parametry. Tento způsob je však nepřijatelný, jednali bychom zcela nelogicky proti původním požadavkům na takovýto systém. Zcela jistě by hrozilo jejich nesplnění a hodlám si tvrdit i dost nevyzpytatelné chování. Druhé řešení spočívá v použití jiného způsobu plánování, algoritmus RM optimálně plánuje na množině úloh RT, mezi jejichž parametry platí vztah $D = T$ a jejichž priorita se v čase nemění.



Obrázek 7.1: RM plán



Obrázek 7.2: DM plán

Když bychom na uvedené úlohy vzaly mechanismus DM, zjistili bychom, že jsou jím plánovatelné. Tento algoritmus je pro nás však okrajovou záležitostí, uvedeme alespoň výsledný plán na 7.1. Zopakujme alespoň, že plánování je nepřímo úměrné parametru D.

Neměňme podmínky a naplánujme úlohy pomocí algoritmu EDF. Tento způsob plánování přepíná mezi úlohami přímo úměrně času zbývajícímu do deadline. Vynechme zdlouhavý a jednoduchý výpočet aktuálních priorit každé úlohy a výsledné hodnoty ukazuje tabulka 7.Tabulka EDF plán

	r	C	D	T	t	0	1	2	3	4	5	6	7	8
τ_0	0	3	5	20	•	1_5	1_4	1_3	4_{21}	4_{20}	4_{19}	4_{18}	4_{17}	4_{16}
τ_1	0	3	7	12	•	2_7	2_6	2_5	1_4	1_3	1_2	3_{12}	3_{11}	3_{10}
τ_2	0	4	10	10	•	3_{10}	3_9	3_8	2_7	2_6	2_5	1_4	1_3	1_2
τ_3	0	3	20	20	•	4_{20}	4_{19}	4_{18}	3_{17}	3_{16}	3_{15}	2_{14}	2_{13}	2_{12}

Tabulka EDF plán

	t	9	10	11	12	13	14	15	16	17	18	19	20
τ_0	•	4_{15}	4_{14}	4_{13}	4_{12}	4_{11}	4_{10}	3_9	2_8	2_7	2_6	2_5	1_4
τ_1	•	3_9	3_8	3_7	2_6	1_5	1_4	1_{15}	3_{14}	3_{13}	3_{12}	3_{11}	3_{10}
τ_2	•	1_1	2_{10}	2_9	3_8	3_7	3_6	2_5	1_4	1_3	1_2	1_1	2_{10}
τ_3	•	2_{11}	1_{10}	1_9	1_8	2_7	2_6	4_5	4_{24}	4_{23}	4_{22}	4_{21}	4_{20}

Tabulka EDF plán

7.1 Testovací úlohy

Pro účely testování vznikly modelové úlohy. Nevykonávají žádnou funkci, pouze demonstrují okolnosti běhu systému. Mohli bychom si představit, že

simulujeme interakci čtyř úloh. Úloha tedy nevykonává nic, na začátku si inicializuje časové parametry. Spustí čekání podle stanovené maximální doby běhu. Poté odečte údaje o skončení a předá řízení OS. Vytváření dalších deadline a začátků period jsou v rozšíření záležitostí plánovače. Pouze pro původní řešení se musí úlohy transformovat do podoby následující

```
void task0(void* pdata)
{
OS_SCHED_EXT     *pext;

for(;;)
{

  pext = (OS_SCHED_EXT *) pdata;
  OS_SchedDataStart( OSTCBHighRdy );
  pext->OSTCBFirstDeadline =  pext->OSTCBStartPeriod +  pext->OSTCBDeadline;
  Puts( STRING_MESSAGE );

  if (pext->OSTCBDeadlineMiss == OS_TRUE){
        Puts( STRING_MESSAGE );
  }else{
    Puts( STRING_MESSAGE );
  }
  while (  pext->OSTCBRan <= pext->OSTCBMaxTime ){
  }
  pext->OSTCBRan = 0;
  if ( pext->OSTCBFirstDeadline < OSTime ){
    pext->OSTCBFirstDeadline =  (INT32U)pext->OSTCBStartPeriod +
pext->OSTCBDeadline;
  }
  while ( (pext->OSTCBStartPeriod  )< OSTime ){
    pext->OSTCBStartPeriod =  pext->OSTCBStartPeriod +
(INT32U)pext->OSTCBPeriod;
  }
  OS_SchedDataEnd( OSTCBHighRdy );
  OSTimeDly( pext->OSTCBStartPeriod - OSTime );/

  }
}
```

Abychom mohli otestovat funkci originálního plánovače, museli jsme implementovat do uživatelské funkce *App_TimeTickHook()* způsob probouzení vláken pro periodické úlohy. Ten kontroluje dobu uplynutí úlohy od uspání.

```
OS_TCB *ptcb;

ptcb = OSTCBList;

while (ptcb->OSTCBPrio != OS_TASK_IDLE_PRIO) {

  OS_ENTER_CRITICAL();
  pext = (OS_SCHED_EXT *) ptcb->OSTCBExtPtr;
  if ( pext->OSTCBStartPeriod <= OSTime && ptcb->OSTCBDly == 0){
    OSTaskResume(ptcb->OSTCBPrio);
  }

  ptcb = ptcb->OSTCBNext;
  OS_EXIT_CRITICAL();
```

```
}
```

Na příkladu použití standardního plánovače, je v době OSTime = 18 máme poslední tick na vykonání 3 tickové úlohy. Po dokončení oznámíme miss. V dalším průběhu se hodnoty miss objevují čím dál častěji. V testování jsme použili schopnost reagovat na minutí kritické meze a simulovali jsme její opravu. Aby se musela kontrolovat po každém spuštění úlohy znovu.

```
Start   -> Task 2    OSTIME =Hx0AE
Start   -> Task 3    OSTIME =0x0B2
Start   -> Task 1    OSTIME =0x0B8
Start   -> Task 2    OSTIME =0x0BB
Start   -> Task 0    OSTIME =0x0BD
       -> MISS
Start   -> Task 1    OSTIME =0x0C5
Start   -> Task 2    OSTIME =0x0C9
       -> MISS
Start   -> Task 3    OSTIME =0x0CD
       -> MISS
Start   -> Task 0    OSTIME =0x0D2
       -> MISS
Start   -> Task 1    OSTIME =0x0D5
Start   -> Task 2    OSTIME =0x0D8
Start   -> Task 1    OSTIME =0x0E2
Start   -> Task 2    OSTIME =0x0E5
Start   -> Task 0    OSTIME =0x0E7
       -> MISS
Start   -> Task 3    OSTIME =0x0EC
Start   -> Task 1    OSTIME =0x0F1
Start   -> Task 2    OSTIME =0x0F4
       -> MISS
Start   -> Task 0    OSTIME =0x0FC
       -> MISS
Start   -> Task 1    OSTIME =0x0FF
Start   -> Task 2    OSTIME =0x102
Start   -> Task 3    OSTIME =0x106
Start   -> Task 1    OSTIME =0x10C
Start   -> Task 2    OSTIME =0x10F
       -> MISS
Start   -> Task 0    OSTIME =0x111
       -> MISS
Start   -> Task 1    OSTIME =0x119
Start   -> Task 3    OSTIME =0x11C
       -> MISS
Start   -> Task 2    OSTIME =0x11D
Start   -> Task 0    OSTIME =0x126
       -> MISS
Start   -> Task 1    OSTIME =0x129
Start   -> Task 2    OSTIME =0x12C
Start   -> Task 3    OSTIME =0x135
Start   -> Task 1    OSTIME =0x136
Start   -> Task 2    OSTIME =0x139
       -> MISS
Start   -> Task 0    OSTIME =0x13B
       -> MISS
```

Uvedené počáteční hodnoty stačí k usouzení, že původní plánování nijak neusnadňuje práci vývojáře. Všechny parametry musí dobře zvážit a sám určit priority v posloupnosti, tak aby se vlákna správně vykonávala.

Na další ukázce viz obr.7.3 vidíme selhání statických algoritmů. Vezměme parametry úloh z úvodního příkladu. Teoreticky jsme si ukázali, že statické algoritmy selžou. Až algoritmus EDF je schopný vybrat vhodnou úlohu a plán dokončit bez ohrožení stability celku. Priority jsou pro úlohy τ_0 až τ_3 zadávány v pořadí $11a14$. Na původním algoritmu nemáme šanci ze systému zjistit, zda se úloha stihla vykonat v čas. Leda bychom si zavedli vlastní počítání v taskách. Z tohoto důvodu bylo umožněno udržovat údaje o úloze i standardní implementaci. Navíc spouštět úlohy periodicky znamená, aby si uživatel nastavil na jak dlouho je úloha suspendována. Proto jsme pro otestování původního plánování doimplementovali uživatelskou metodu *OS-TaskIdleHook()*. V níž jsme úlohy po uplynutí uspání opět aktivovali.

Malá změna ve výběru a hned jsme schopni splnit mnohem větší část úloh. I přesto však dochází k situacím, že nejsme schopni zaručit splnění úloh.

V následující části na obr.7.4 máme porovnání implementovaných dynamických algoritmů. Na prvním dynamickém algoritmu jsme si ověřili, zda jsme teoretické hodnoty spočetli do tabulky 7.Tabulka EDF plán správně. Na druhém vidíme jedinečnost přístupu ke zpracování a častější přepínání kontextu mezi úlohami. Algoritmus least laxity first vybírá úlohy ve správném pořadí, tak aby nedocházelo k miss. Kvůli častému přepínání kontextu algoritmu LLF, museli být vstupy pro simulátor doplněny i do souboru s rozšířením.

Ačkoli jsme počáteční hodnoty spočetli správně, dochází k minutí deadline u obou algoritmů, které nejsme téměř schopni odhalit jinak, nežli simulováním. Zvýše uvedených výstupů vyplývají následující povinnosti vývojáře.

- Dokonale analyzovat použitou harwarovou platformu
- Analyzovat časové parametry vytvořených úloh
- Ověřit simulacemi funkčnost
- Upravit kritická místa
- Případně schopně reagovat na kritická místa

Rate Monotonic scheduling

```
Start   - > Task 0    OSTIME =0x0AF
       - > MISS
Start   - > Task 1    OSTIME =0x0B0
       - > MISS
Start   - > Task 2    OSTIME =0x0B4
Start   - > Task 1    OSTIME =0x0BC
       - > MISS
Start   - > Task 2    OSTIME =0x0BE
Start   - > Task 3    OSTIME =0x0C3
Start   - > Task 2    OSTIME =0x0C8
Start   - > Task 1    OSTIME =0x0CC
       - > MISS
Start   - > Task 0    OSTIME =0x0CF
       - > MISS
Start   - > Task 2    OSTIME =0x0D2
Start   - > Task 3    OSTIME =0x0D7
Start   - > Task 1    OSTIME =0x0D8
Start   - > Task 2    OSTIME =0x0DC
Start   - > Task 1    OSTIME =0x0E4
Start   - > Task 2    OSTIME =0x0E6
Start   - > Task 0    OSTIME =0x0EB
       - > MISS
Start   - > Task 2    OSTIME =0x0F0
Start   - > Task 1    OSTIME =0x0F4
Start   - > Task 3    OSTIME =0x0F7
Start   - > Task 2    OSTIME =0x0FA
Start   - > Task 0    OSTIME =0x0FF
       - > MISS
Start   - > Task 1    OSTIME =0x100
Start   - > Task 2    OSTIME =0x104
Start   - > Task 1    OSTIME =0x10C
Start   - > Task 2    OSTIME =0x10E
Start   - > Task 3    OSTIME =0x113
Start   - > Task 2    OSTIME =0x118
Start   - > Task 1    OSTIME =0x11C
       - > MISS
Start   - > Task 0    OSTIME =0x11F
       - > MISS
Start   - > Task 2    OSTIME =0x122
Start   - > Task 3    OSTIME =0x127
Start   - > Task 1    OSTIME =0x128
       - > MISS
Start   - > Task 2    OSTIME =0x12C
Start   - > Task 1    OSTIME =0x134
       - > MISS
Start   - > Task 2    OSTIME =0x136
Start   - > Task 0    OSTIME =0x13B
       - > MISS
```

Deadline Monotonic scheduling

```
Start   - > Task 0    OSTIME =0x0A0
Start   - > Task 1    OSTIME =0x0A8
Start   - > Task 3    OSTIME =0x0AB
Start   - > Task 2    OSTIME =0x0AC
Start   - > Task 0    OSTIME =0x0B4
Start   - > Task 1    OSTIME =0x0B7
Start   - > Task 2    OSTIME =0x0BA
Start   - > Task 1    OSTIME =0x0C3
Start   - > Task 2    OSTIME =0x0C6
Start   - > Task 0    OSTIME =0x0C8
Start   - > Task 3    OSTIME =0x0CD
Start   - > Task 1    OSTIME =0x0CF
Start   - > Task 2    OSTIME =0x0D3
Start   - > Task 1    OSTIME =0x0DB
Start   - > Task 0    OSTIME =0x0DC
Start   - > Task 2    OSTIME =0x0E1
Start   - > Task 3    OSTIME =0x0E8
Start   - > Task 1    OSTIME =0x0EA
       - > MISS
Start   - > Task 2    OSTIME =0x0ED
Start   - > Task 0    OSTIME =0x0F0
Start   - > Task 1    OSTIME =0x0F6
       - > MISS
Start   - > Task 2    OSTIME =0x0FA
Start   - > Task 1    OSTIME =0x102
       - > MISS
Start   - > Task 0    OSTIME =0x104
Start   - > Task 2    OSTIME =0x108
Start   - > Task 3    OSTIME =0x10C
Start   - > Task 1    OSTIME =0x111
       - > MISS
Start   - > Task 2    OSTIME =0x114
Start   - > Task 0    OSTIME =0x118
Start   - > Task 1    OSTIME =0x11D
       - > MISS
Start   - > Task 3    OSTIME =0x120
Start   - > Task 2    OSTIME =0x121
Start   - > Task 1    OSTIME =0x129
       - > MISS
Start   - > Task 0    OSTIME =0x12C
Start   - > Task 2    OSTIME =0x12F
Start   - > Task 1    OSTIME =0x138
       - > MISS
Start   - > Task 2    OSTIME =0x13B
Start   - > Task 3    OSTIME =0x13F
Start   - > Task 0    OSTIME =0x140
Start   - > Task 1    OSTIME =0x144
Start   - > Task 2    OSTIME =0x147
```

Obrázek 7.3: Statické algoritmy

Early Deadline First

```
Start   - > Task 1    OSTIME =0x0AF
       - > MISS
Start   - > Task 2    OSTIME =0x0B9
Start   - > Task 0    OSTIME =0x0BA
       - > MISS
Start   - > Task 1    OSTIME =0x0BD
       - > MISS
Start   - > Task 3    OSTIME =0x0C0
Start   - > Task 1    OSTIME =0x0C9
       - > MISS
Start   - > Task 2    OSTIME =0x0CC
Start   - > Task 0    OSTIME =0x0CE
       - > MISS
Start   - > Task 3    OSTIME =0x0D4
Start   - > Task 1    OSTIME =0x0D5
       - > MISS
Start   - > Task 2    OSTIME =0x0DA
Start   - > Task 1    OSTIME =0x0E1
       - > MISS
Start   - > Task 0    OSTIME =0x0E4
       - > MISS
Start   - > Task 2    OSTIME =0x0E7
Start   - > Task 3    OSTIME =0x0EB
Start   - > Task 1    OSTIME =0x0ED
       - > MISS
Start   - > Task 2    OSTIME =0x0F7
Start   - > Task 0    OSTIME =0x0F8
       - > MISS
Start   - > Task 1    OSTIME =0x0FB
       - > MISS
Start   - > Task 3    OSTIME =0x102
Start   - > Task 1    OSTIME =0x107
       - > MISS
Start   - > Task 2    OSTIME =0x109
Start   - > Task 0    OSTIME =0x10C
       - > MISS
Start   - > Task 3    OSTIME =0x116
Start   - > Task 1    OSTIME =0x119
       - > MISS
Start   - > Task 2    OSTIME =0x11C
Start   - > Task 0    OSTIME =0x120
       - > MISS
Start   - > Task 1    OSTIME =0x125
       - > MISS
Start   - > Task 2    OSTIME =0x129
Start   - > Task 1    OSTIME =0x131
       - > MISS
```

Least Laxity First

```
Start   - > Task 1    OSTIME =0x0A7
       - > MISS
Start   - > Task 2    OSTIME =0x0AB
Start   - > Task 1    OSTIME =0x0B3
       - > MISS
Start   - > Task 0    OSTIME =0x0B6
Start   - > Task 2    OSTIME =0x0B9
Start   - > Task 3    OSTIME =0x0BD
Start   - > Task 1    OSTIME =0x0BF
       - > MISS
 Back   -> Task 3
Start   - > Task 2    OSTIME =0x0C3
 Back   -> Task 3
Start   - > Task 0    OSTIME =0x0CA
Start   - > Task 1    OSTIME =0x0CD
       - > MISS
Start   - > Task 2    OSTIME =0x0D0
Start   - > Task 3    OSTIME =0x0D8
Start   - > Task 1    OSTIME =0x0D9
Start   - > Task 2    OSTIME =0x0DC
Start   - > Task 0    OSTIME =0x0DE
 Back   -> Task 2
 Back   -> Task 3
Start   - > Task 1    OSTIME =0x0E5
 Back   -> Task 3
Start   - > Task 2    OSTIME =0x0E9
Start   - > Task 1    OSTIME =0x0F1
Start   - > Task 0    OSTIME =0x0F2
 Back   -> Task 1
Start   - > Task 2    OSTIME =0x0F7
Start   - > Task 3    OSTIME =0x0FB
Start   - > Task 1    OSTIME =0x100
Start   - > Task 2    OSTIME =0x103
Start   - > Task 0    OSTIME =0x106
 Back   -> Task 2
Start   - > Task 1    OSTIME =0x10C
Start   - > Task 3    OSTIME =0x10F
Start   - > Task 2    OSTIME =0x110
 Back   -> Task 3
Start   - > Task 1    OSTIME =0x118
Start   - > Task 0    OSTIME =0x11B
Start   - > Task 2    OSTIME =0x11E
Start   - > Task 1    OSTIME =0x124
Start   - > Task 3    OSTIME =0x127
Start   - > Task 2    OSTIME =0x128
 Back   -> Task 3
Start   - > Task 0    OSTIME =0x12F
       - > MISS
```

Obrázek 7.4: Dynamické algoritmy

- **!**Nezapomenout na přerušení, které kritické hodnoty ještě zhorší

τ_0	$(0, 1, 25, 50)$	$(0, 1, 10, 20)$	$(0, 8, 25, 50)$	$(0, 13, 42, 45)$
τ_1	$(0, 2, 40, 50)$	$(0, 2, 10, 18)$	$(0, 12, 40, 50)$	$(0, 12, 47, 50)$
τ_2	$(0, 1, 35, 50)$	$(0, 1, 10, 10)$	$(0, 6, 35, 50)$	$(0, 12, 47, 48)$
τ_3	$(0, 1, 30, 50)$	$(0, 2, 18, 20)$	$(0, 4, 30, 50)$	$(0, 11, 47, 47)$
ORIG	OK	MISS	MISS	MISS
RM	OK	OK	MISS	MISS
DM	OK	OK	OK	MISS
EDF	OK	OK	OK	MISS
LLF	OK	OK	OK	MISS

Vybrané úlohy

Najít hranici kde jeden algoritmus přestává fungovat a jiný to ještě zvládne, se povedlo jen pro statický rate monotonic. Díky řazení podle period, není ani u velmi prostých úloh zajistit správný běh. Pro zbývající algoritmy je situace mnohem lepší, z pozorování v tabulce 7.1.Vybrané úlohy lze usoudit, že úspěšnost roste s klesajícím poměrem $\frac{C}{T}$ a $\frac{C}{D}$. To vše je navíc ovlivněno počtem úloh. Za jistou mez plánovatelnosti bychom mohli označit $\sum C_i \leq D_{\min}$.

Všechny simulované úlohy probíhaly alespoň do času OSTime 0x200, kde již byla velká pravděpodobnost, že budeme schopni rozpoznat problémové úlohy. Aplikace k otestování jsou na přiloženém CD v adresáři *\function_workspace\ WorkSpace\IntrApp*.

7.2 Nevýhody řešení

Podíváme-li se na časové parametry úloh, musíme si uvědomit, že hodnoty musí být v souladu s časovými parametry systému. Např. úloha $\tau_1(0, 3, 7, 12)$ hodnoty udávají počet tiků. Kolik instrukcí za jeden tik se provede, je závislé na platformě, často i na optimalizaci překladačem.

Kdybychom hodnoty z úvodního příkladu použili v praxi na nějakém méně výkonné platformě. Mohl by systém být nestabilní a nepředvídatelný. Souvislost hledejme ve správném nastavení počtu tiků za sekundu. Zejména z důvodu, že vyhledání úlohy s nejvyšší prioritou, po té přepnout kontext procesoru by trvalo n-krát déle, než úloha podle parametrů od vývojáře.

Pokud by úloha byla opravdu prováděla svou činnost, jak uvedl. Bylo by vhodné uvažovat přinejmenším o použití původního plánování. Přenáší se tak zodpovědnost na vývojáře používající rozšíření, který musí velmi dobře znát podmínky použití.

V předchozí části jsme zmínili, že doba vyhledání úlohy nejvyšší priority prodlužuje dobu mezi přepnutím kontextu. Sběr dat pro přeplánování by mohl být přesunut do rukou vývojáře, aby jej vhodně využíval ve funkci *OSTaskSwHook*, ta je volána jako uživatelská při přepnutí kontextu. Sám vývojář by si určil jak často sbírat data. Samotný algoritmus sběru informací je složitosti $O(N)$.

Původní myšlenka seřazovat a nově přiřazovat priority všem úlohám, byla při dokončování práce zavrhnuta. Zejména z důvodu značného zvětšení režie jádra systému oproti vlastní funkci úloh. Problém by spočíval v řadícím algoritmu, at' bychom vybrali jakýkoli jednoduše implementovatelný v jazyce C, nejlépe takový jenž nevyžaduje větší dodatečnou paměť, byl by složitosti $O(NlogN)$.

Rychlost výběru úlohy se tedy snížila, v původním řešení se provádělo 20 instrukcí včetně návratových. V obou případech díky přidání inicializace vzrostl počet instrukcí o $OS_{NS}YS_{T}ASK$, kterým se musí dodat ideální časové parametry. Samotnému plánování vzrostl počet instrukcí n-násobně na hodnotu $N \cdot 60 + 50$, v závislosti na počtu úloh, nejvíce jej ovlivňuje zaznamenávání správných údajů. Výběr priority přesto zůstal stále složitosti $O(N)$.

7.3 Výhody řešení

Zakomponováním našich plánovacích metod odpadá nutnost vývojáře přiřadit priority úlohám, tak by plán splnily. Zejména díky dynamickým algoritmům, je množina splněných úloh větší nežli původní plánovatelná množina. Zadání práce považuji za splněné, pouze se mohla větší část věnovat simulacím a jejich porovnáním původního výběru úloh s rozšířením.

V souvislosti s algoritmem least laxity first, bývají obvykle konstruovány formou vyzývání úloh. Režii další úlohy v systému jsme se vyhnuli díky kontrole priorit při návratu z přerušení. Navíc jsme připravili model odpočtu časových údajů i pro jiné aplikace, který můžeme použít i bez zapnutého rozšíření.

Pokud známe velmi dobře použitou hardwarovou platformu a jsme schopni správně určit okolnosti děje. Dosáhneme nejlepších výsledků plánovače při použití algoritmu LLF. Nepříjemnou daní je častější počet přepnutí kontextu. Další výhoda je, že plánovač je deterministický. Při shodě parametrů totiž nehrozí, že pokaždé vybere jinou úlohu. Vybere totiž tu úlohu, která byla do TCB zadána dřív. Systém μC/OS-II se zapracovanými algoritmy, je schopný reagovat na události opoždění zpracování úlohy. Je tedy na uživateli jak se k tomu postaví.

8 Závěr

Tato práce se věnuje nejběžnějším plánovacím algoritmům Real-time operačním systémům. Statické algoritmy rate monotonic a deadline monotonic spolu s dynamickými early deadline first a least laxity jsou implementovány do RTOS μC/OS-II. Aby bylo implementování možné, jsou úvodní kapitoly věnovány úvodu do problematiky samotných RT systémům a nejvíce právě μC/OS-II.

V kapitole 5 jsou vysvětleny vytvořené plánovací algoritmy. Jejichž implementace je popsána v následující kapitole 6, spolu s popisem důležitých bloků zdrojové kódu. V předposlední kapitole 7 jsou shrnuty připomínky k realizaci. Dále jsou zde výsledky simulací, spolu s modelovou testovací úlohou. Kvůli níž bylo udržování údajů o úlohách poskytnuto i původní implementaci.

Nejlepší volbou z implementovaných algoritmů je výběr stupně volnosti úlohy. Ale jen v případě, že jsme schopni tento údaj odečíst. K tomu důležitá dokonalá znalost architektury mikroprocesoru. V ostatních případech jsou doborou volbou výběru úloh jak podle statické hodnoty deadline, tak i její závislosti na čase. Výsledky simulace ukázaly, že implementované rozšíření je lepší volbou pro výběr úloh. Práce splňuje všechny body zadání, a proto ji považuji za splněnou.

Literatura

[MicroII] Jean J. Labrosse: *µC/OS-II The Real-Time Kernel* CMP Books, San Francisco, CA • New York, NY • Lawrence, KS, 2002. ISBN: 1-57820-103-9

[CRTOS] Rafael V. Aroca, Glauco Caurin: *A Real Time Operating Systems (RTOS) Comparison*Laboratório de Mecatrônica EESC – Universidade de São Paulo, 2009

[Sojka] M. Sojka: *Programování systémů reálného času* přednášky ČVUT, Praha, září 2010

[CLLiu] C.L. Liu, Jamse W. Layland: *Scheduling Algorituhms for Multiprogramming in Hard-Real-Time Enviroment* Project MAC, MIT, CIT, 1973

[WinEM] Microsoft: *Windows Embedded home page*. (říjen 2011). `Informace naleznete na http://www.microsoft.com/cze/windowsembedded/`

[Nucle] Mentor Graphics: *Mentor Graphics home page*. (říjen 2011). `Informace naleznete na http://www.mentor.com/`

[QNXso] QNX Software: *QNX home page*. (říjen 2011). `Informace naleznete na http://www.qnx.com/`

[VxWor] Wind River: *Wind River home page*. (říjen 2011). `Informace naleznete na http://www.windriver.com/`

[LyRto] LynuxWorks: *LynuxWorks home page*. (říjen 2011). `Informace naleznete na http://www.lynuxworks.com/`

[Ren] Renesas Electronics Corporation : *Renesas home page*. `Použitý software a řadu informací naleznete na http://www.renesas.com/`

[KPIT] KPIT Tools : *KPIT GNU Tools home page.*
`Stránky použitého překladače http://www.kpitgnutools.com`

[Dud] Dr. Ing. Karel Dudáček : *Work page K.Dudáčka.*
`Návody k prostředí HEW v češt-`
`ině naleznete na http://www.http://home.zcu.cz/~dudacek/`

[PJer] Petr Jerman : *Použití operačního systému MicroC/OS-II na procesoru H8S.*
`Port MicroC/OS-II pro procesory H8S 2600`

[DIEDFonSL] Channamallikarjuna Mattihalli : *Designing and Implementing of Earliest Deadline First scheduling algorithm on Standard Linux.*
`2010 IEEE/ACM`

[JStr] Josef Strnadel : *Návrh časově kritických systémů I-IV.*
`V časopise AUTOMA vyšlo v letech 2010-2011`

[EDFlk] Dario Faggioli, Fabio Checconi, Michael Trimarchi, Claudio Scordino : *An EDF scheduling class for the Linux kernel.*
`European Commission under the ACTORS project (FP7-`
`ICT-216586).`

Použité zkratky

ABS	Antiblockiersystem
ARM	Advanced RISC Machine
BGP	Border Gateway Protocol
BSP	Board Support Packages
CPU	Central Processing Unit
ECB	Event Control Block
ESR	Electronic Stability Control
FAT	File Allocation Table
FIFO	First In First Out
FTP	File Transfer Protocol
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
HW	Hardware
IDE	Integrated Drive Electronics
INTR	Interrupt Request
IP	Internet Protocol
IPSec	IP security
ISR	Interrupt Service Routine
MCB	Memory Control Block
MIPS	Microprocessor without Interlocked Pipeline Stages
NMI	Non Maskable Interrupt
OS	Operační systém
OSPF	Open Shortest Path First
POSIX	akronym pro Portable Operating System Interface
RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
RT	Real-Time
RTOS	Real-time Operační Systém
SATA	Serial Advanced Technology Attachment
SH	SuperH (typ ARM procesoru)
SMTP	Simple Mail Transfer Protocol
SNTP	Simple Network Time Protocol
SSH	Secure Shell
TCB	Task Control Block
USB	Universal Serial Bus

Seznam obrázků

Seznam tabulek

A Vývojová prostředí a nastavení μC/OS-II

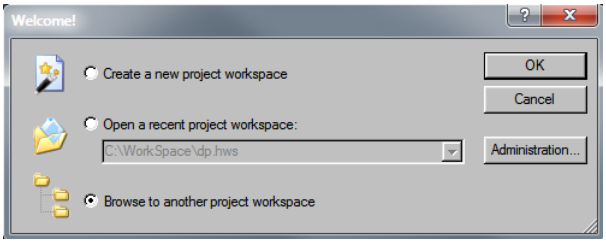
A Nebudeme se věnovat standardním možnostem nastavení systému. Pouze si ukážeme jak využít přidané vlastnosti systému. Jak je správně nakonfigurovat. K tomu nepotřebujeme žádné vývojové prostředí, stačí nám libovolný textový editor. Po té co nakonfigurujeme překládané části systému, ukážeme si jak na instalovaném prostředí upravený kód přeložit. Zmiňované materiály jsou umístěné na přiloženém záznamovém mediu.

A.1 Vývojové prostředí

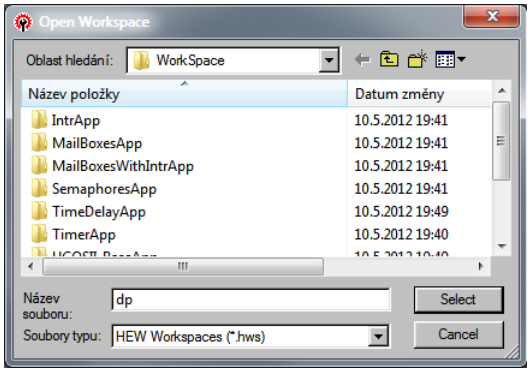
K vývoji aplikace μC/OS-II bylo použito vývojového prostředí High-performance Embedded Workshop verze 4.08.00.011 dostupné na mediu, *\software\h8v7000_ev.exe*, kde jsou dostupné také rozdílové aktulizace. Novější jsou ke stažení z hlavních stránek [Ren]. Ke stažení je nutné být zaregistrován. Postup instalace nebudeme popisovat, je velmi intuitivní a verzi od verze se mírně mění. Na zmíněných internetových stránkách naleznete také manuály k vývojovému kitu EVB2633F.

K prostředí dále nainstalujeme GNUH8 Windows Tool Chain(ELF), *\software\GNUH8v1003-ELF.exe*, nejlépe v nejnovější verzi, ke stažení jsou opět po registraci na [KPIT]. Při instalaci je vyžadován aktivační kód zaslaný emailem. Po naistalování je překladač integrován do prostředí HEW. Ke stažení jsou zde k dispozici též manuály k překladači a assembleru.

HEW obvykle ukládá pracovní složku jako *c:\Workspace*. Doporučil bych strukturu složek zachovat, stačí tedy z CD zkopírovat adresář *Workspace*. Ten obsahuje port pro použitý kit, port je složen z pár ukázkových aplikací z [PJer]. Po spuštění HEW se nás aplikace zeptá, zda chceme začít nový projekt, nebo otevřít ve vybraném umístění, viz. obr.A.1. Zvolíme možnost vybrat a v následujícím okně Obr.A.2 vybereme soubor *dp.hew*. Pravděpodobně se nás ještě v závislosti na verzi HEW zeptá, zda chceme projekt transformovat do nové podoby. Pokud se to stane zvolíme yes. Po načtení projektu bude vývojové prostředí odpovídat zhruba obr.A.3. V levé části je přehled souborů projektu a závislostí souborů mezi sebou.

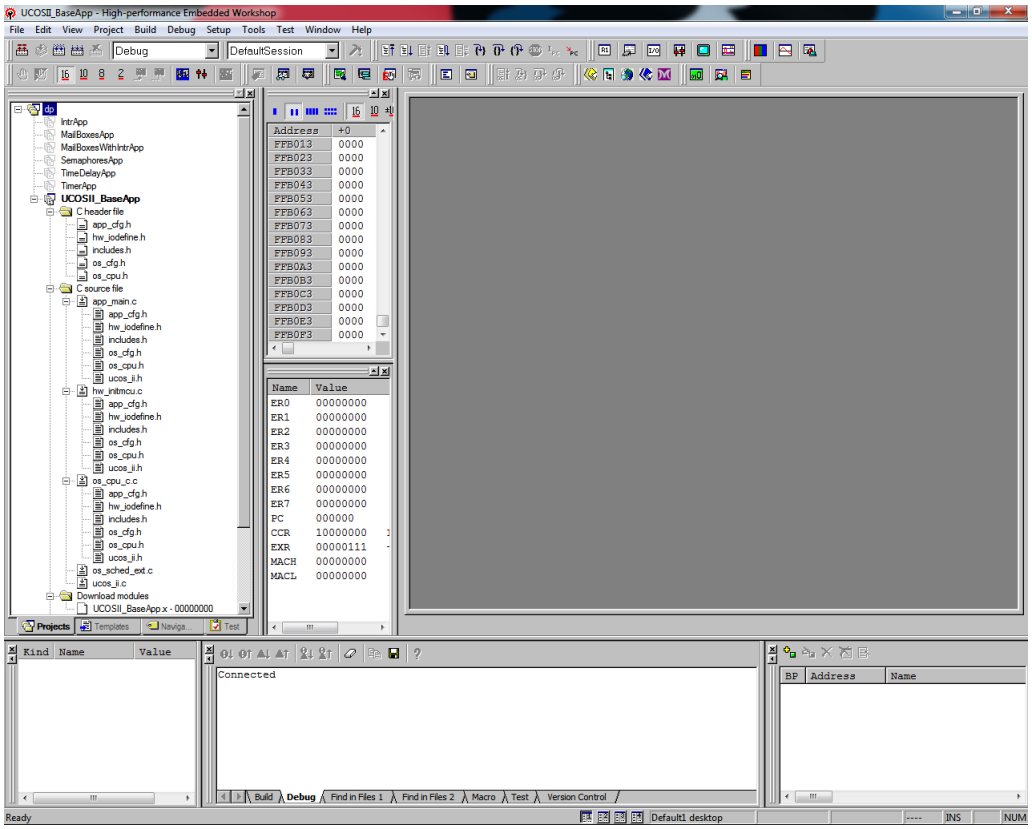


Obrázek A.1: První spuštění HEW

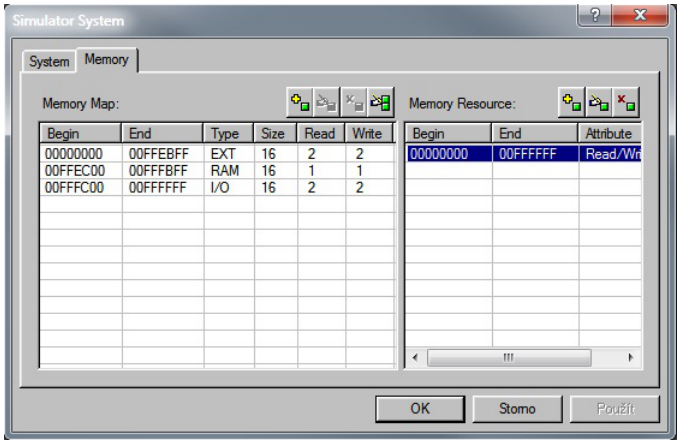


Obrázek A.2: Otevření projektu

Výchozí instalace a nastavení překladače jsou funkce schopné. Jediné co zbývá nastavit pro simulátor kitu je adresní prostor simulátoru. Simulátor procesoru H8S/2600 má nastaven rozsah adres simulované paměti tak, aby vyhovoval obvyklým úlohám. Pokud je to nutné (simulátor hlásí chybu Memory Access Error(Address:H'xxxxxxxx)), lze rozsah paměti upravit v okně, které se otevře příkazem Setup - Simulator - Memory Resource. Záznamy v tabulce Memory Resource lze upravit tak, aby paměť pokrývala celý fyzický adresní prostor procesoru H8S, tj. 16 MB - viz následující obrázek A.4. Krom nastavení paměti, je dobré ověřit, odkud se nahrává přeložená aplikace a o jaký typ simulátoru se jedná. Toto nastaveni najdeme v menu *DebugSettings*, na záložce target vybereme H8S/2600A Simulátor. Dolní části okna je výpis nahrávaných souborů, pokud zde žádný není, přidáme jej tlačítkem Add. Pokud se zde nachází, zkontrolujeme, že se jedná a správné umístění. Správné je ve výchozím nastavení, když je zde nastaven přeložený soubor ze složky otevřeného projektu.



Obrázek A.3: Vývojové prostředí HEW



Obrázek A.4: Adresní prostor simulátoru HEW

B Sestavení konfigurace

B Pokud potřebujeme rozšířit jinou portaci, či vzorový μC/OS-II, potřebujeme lehce upravit zdrojové kódy. Postup je to jednoduchý, pro verzi systému 2.86 bude platit následující postup. Upozornil bych, že v budoucích verzích se budou lišit zejména uvedené řádky, kde se změny provádí.

Systém jako takový dodržuje jistou organizační strukturu, popsanou v [MicroII]. Zde se dozvíme, že soubory jádra systému jsou drženy v jedné složce a ostatní, závislé k portaci jsou ve složce jiné. Proto do složky se soubory jádra zkopírujeme soubory *os_sched_ext.c* a *os_sched_ext.h*, ty se nacházejí ve složce *\os_sched_ext*.

Dále musíme upravit původní zdrojové kódy. Začněme od *ucosii.c*, zde k seznamu vkládaných souborů přidáme

```
#include <os_sched_ext.c>
```

Stejnou řádku přidáme do souboru *includes.h*, který používáme pro vývoj aplikací a importu dodatečných knihoven. Do souboru *ucosii.h* je dobré umístit kontrolu použití rozšíření, tato podmínka při překladu ověří, zda je použito vytváření úloh s externími daty. Doporučuji, pokud přidáváme podmínky překladu, které nevybírají přímo kousky kódu, umístit na konec souborů.

```
#if OS_SCH_TYPE > 0 && OS_TASK_CREATE_EXT_EN < 1
  #error STRING ABOUT EXPRESION
#endif

#if OS_SCH_TYPE == 1 || OS_SCHE_TYPE == 2

  #if OS_MAX_TASKS > (OS_LOWEST_PRIO / 2)
    #error STRING ABOUT EXPRESION
  #endif

#endif
```

Všimněme si, že jsme přidali novou konstantu překladu, tu přidáme na konec souboru *os_cfg.h*.

```
#define OS_SCH_TYPE      typ_planovani
```

Typ plánování nahradíme číslem

- 0 Původní statické plánování

- 1 Externí RMA
- 2 Externí DMA
- 3 Externí EDF
- 4 Externí LLF

Nyní budeme muset upravit soubor *os_core.c*, musíme zajistit, aby se při překladu a volbě externích plánovacích algoritmů nepřeložila funkce *OS_SchedNew()*. Na řádku před komentář (1630) umístíme

```
#if OS_SCH_TYPE == 0
```

a za konec funkce, resp. před komentář (1670) následující funkce

```
#endif
```

Dále pokračujme úpravami ve funkci *OSIntExit()*, zde je místo změny kontextu procesoru. Řádek s *OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];* (664) nahradíme za

```
#if OS_SCH_TYPE == 0
  OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
#endif
```

To samé provedeme ve funkci *OS_Shed()* na řádku (1620) a také ve funkci *OSStart()*(790). Po těchto úpravách je systém použitelný pro další vývoj portace a funkční aplikace. Pokud se vám shodují verze μC/OS-II a nedělali jste žádné úpravy v souboru os_core.c lze jej překopírovat již upravený ze složky *\function_workspace\WorkSpace\UCOSIIBaseApp\MicroCOSIIv286*

Jak používat rozšíření Musíme dodržovat pravidla a doporučení, které stanovuje původní implementace. Úloha je napsána jako nekonečná smyčka, která neobsahuje volání return. Aby docházelo k přepínání úloh systémem, musí být uvnitř smyčky volána jedna z následujících funkcí

- OSMboxPend()
- OSFlagPend()
- OSMutexPend()
- OSQPend()

- OSSemPend()
- OSTimeDly()
- OSTimyDlyHMSM()
- OSTaskSuspend()
- OSTaskDel()

Priority, které uživatel zadá nebudou respektovány s výhradami. Tím je myšleno, že uživatelem zadané priority musí být unikátní a nesmí být z rezervovaných priorit. Jen pro úplnost jsou rezervovány OS_LOWEST_PRIO, OS_LOWEST_PRIO - 1 a naopak nejvyšší priority 0 a 1.

Abychom mohli použít rozšíření o plánování statickými nebo dynamickými algoritmy, musí být při portaci vytvořena obsluha čítače a časovače, bez něj nemá plánovač jak odměřovat údaje. Dále musíme vytvořit strukturu *OS_SCHED_EXT*. A vyplnit následující parametry vztažené k modelu úlohy viz obr.5.1

- OSTCBStartTask - odpovídá parametru r
- OSTCBMaxTime - odpovídá parametru C
- OSTCBDeadline - odpovídá parametru D
- OSTCBPeriod - odpovídá parametru T

Pokud potřebuje uživatel předávat data, k čemuž původně *pext* sloužil, musíme upravit její zpracování a posunout se o jeden pointer dál na *OS_SCHED_EXT->pext*. Takto připravená data předáme funkci *OSTaskCreateExt()*. Ještě před spuštěním multitaskingu, je nutné inicializovat externí scheduler a to voláním funkce *OSSchedExtInit()*.Toto zdůrazňujeme, kvůli neinicializované paměti plánovače, kde pak dochází k porušení pravidel adresace paměti. Pokud jsme provedli všechny kroky z předchozí kapitoly, máme upravený systém připraven k použití.

Ke snadnějšímu zadávání parametrů úloh lze využít vytvořenou funkci, které se předá ukazatel na strukturu s parametry úlohy

```
static void OS_SchedPext( OS_SCHED_EXT *pext , INT16U startTask,
                INT16U maxTime, INT16U deadline, INT16U period );
```

Funkce zajistí inicializaci údajů potřebných pro správný běh. Spolu s touto funkcí byl vytvořen model úlohy, na který je možné různě upravit a použít. Samozřejmostí je možnost použití standardního plánování rozšířeného o sběr informací tasků. Tento model naleznete v adresáři *\function_workspace\WorkSpace\IntrApp*.