VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MECHANISMY PLÁNOVÁNÍ RT ÚLOH PŘI NEDOSTATKU VÝPOČETNÍCH A ENERGETICKÝCH ZDROJŮ

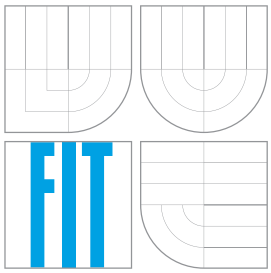
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN POKORNÝ

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MECHANISMY PLÁNOVÁNÍ RT ÚLOH PŘI NEDOSTATKU VÝPOČETNÍCH A ENERGETICKÝCH ZDROJŮ

MECHANISMS FOR SCHEDULING RT TASKS DURING LACK OF COMPUTATIONAL AND ENERGY SOURCES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN POKORNÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOSEF STRNADEL, Ph.D.

BRNO 2012

Abstrakt

Tato práce se zaměřuje na popis mechanismů pro plánování úloh při přetížení a pro snížení spotřeby energie. Každá z částí uvádí tyto mechanismy a důvody pro jejich použití. Mimo jiné jsou popsány i konkrétní algoritmy, které jsou v další části diplomové práce implementovány v systému μC/OS-II a porovnány jejich výkony.

Abstract

This term project deals with the problem of scheduling real-time tasks in overload conditions and techniques for lowering power consumption. Each of these parts features mechanisms and reasons for their using. There are also described specific algorithms, that are implemented in operating system μC/OS-II, and compared in next phase of master's thesis.

Klíčová slova

Real-time systém, model RT úlohy, plánování RT úloh, rate-monotonic, earliest deadline first, nízký příkon, dynamic voltage scaling, dynamic power management, šetření cyklů, nízko energetické prioritní plánování, přetížení, překročení, nejlepší snaha, DASA, LBESA, D^{over}.

Keywords

Real-time system, task model, task scheduling, Rate-Monotonic, Earliest Deadline First, Low-Power, Dynamic Voltage Scaling, Dynamic Power Management, Cycle-Conserving, Low-Power Priority-Based Real-Time Scheduling, Overload, Overrun, Best-Effort, DASA, LBESA, D^{over}.

Citace

Martin Pokorný: Mechanismy plánování RT úloh při nedostatku výpočetních a energetických zdrojů, diplomová práce, Brno, FIT VUT v Brně, 2012

Mechanismy plánování RT úloh při nedostatku výpočetních a energetických zdrojů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana ing. Josefa Strnadela, Ph.D.

.....
Martin Pokorný
18. května 2012

Poděkování

Rád bych poděkoval panu Ing. Josefovi Strnadelovi, Ph.D. za jeho vstřícný přístup a poskytnuté cenné rady při vedení této diplomové práce.

© Martin Pokorný, 2012.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Dnešní svět lidí je řízen počítači. Možná to tak někomu nepřipadá, přesto by si lidé ve vyspělých zemích již nedovedli představit žít bez nich. Můžeme je potkat na každém kroku, počínaje digitálními hodinkami a superpočítači po předpověď počasí konče. Všechny tyto počítače mají více či méně něco společného. V první řadě obsahují řídící, resp. výpočetní jednotku, např. procesor. Dále musí nějakým adekvátním způsobem dokázat reagovat na podněty, které k nim přichází. Aby tyto podněty nezahltily procesor, musí být nějakým způsobem řízeny. Cílem této práce je pro specifické podmínky tyto způsoby shrnout, popsat a vybrané mechanismy porovnat.

Teoretickou část diplomové práce je možné rozdělit do dvou částí. První část, skládající se z druhé a třetí kapitoly, popisuje základní mechanismy real-time systémů. Druhá část je již zaměřena na konkrétní problémy, jejich příčiny a východiska.

Druhá kapitola se zaměřuje na základní pojmy z oblasti real-time systémů. Vysvětluje pojem real-time systém, z jakých částí se skládá a jak je možné je rozdělit. Dále jsou popsány pojmy real-time operační systém a jeho části, jádro a objekty jádra.

V třetí kapitole jsou popsány základní mechanismy plánování úloh v systému, uvedeny některé základní algoritmy pro plánování a také je zaveden model real-time úlohy.

Čtvrtá kapitola se snaží popsat mechanismy plánování úloh se snahou minimalizovat spotřebu energie systému. Jsou popsány důvody a možné způsoby, jak dosáhnout co nejlepších výsledků.

Kapitola pátá je zaměřena na mechanismy, které je možné využít při přetížení systému. Jsou zde uvedeny příčiny vedoucí k přetížení, možná východiska a některé konkrétní algoritmy, které jsou dále v diplomové práci implementovány a porovnávány.

Praktická část se je rozdělena do tří kapitol. Nejprve jsou popsány prostředky, které jsou použity pro implementaci, měření a vyhodnocení zvolených algoritmů. V krátkosti je zde popsána vývojová deska, procesor osazený na vývojové desce a jádro μC/OS-II.

Osmá kapitola se již plně zaměřuje na implementaci vybraných algoritmů do jádra μC/OS-II. Nejprve je popsán princip vytváření úloh, jejich uspávání a buzení a předávání priority. Následně jsou podrobněji popsány jendotlivé implementované algoritmy a jejich pomocné funkce. Poslední podkapitola se zaměřuje na popis implementace logování informací.

Poslední kapitola diplomové práce se zaměřuje na popsání podmínek provádění experimentů, interpretaci výsledků, porovnání implementovaných algoritmů a zhodnocení dosažených výsledků.

V přílohách jsou umístěny diagramy pro lepší pochopení složitějších algoritmů, metriky kódu a paměťové nároky, vizualizace logů ukázkových sad úloh a také stručný manuál.

Kapitola 2

Základní pojmy

2.1 Real-time systém

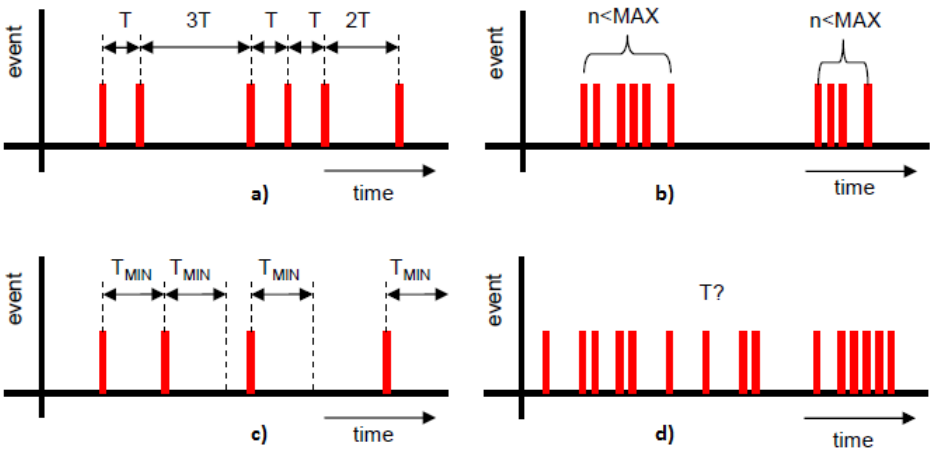
Co je to real-time (dále již jen RT) systém a jaké jsou na něj kladené požadavky? Obecně si můžeme RT systém rozdělit na dvě části. První částí je samotný RT systém a druhou částí je prostředí, ve kterém RT sytém pracuje. Z tohoto prostředí mohou do našeho RT systému přicházet podněty, na které musí náš RT systém adekvátně zareagovat. Těmto podnětům říkáme externí události (dále jen události[1]).

2.1.1 Událost

Události můžeme rozdělit na periodické a aperiodické. Periodické události přicházejí vždy se stejnou prodlevou, to znamená, že od výskytu první události se vyskytne další událost se zpožděním o právě jednu periodu. O aperiodických událostí nevíme přesně, kdy se vyskytnou, avšak i zde může být pozorována jistá vlastnost opakování. Aperiodické události můžeme dále rozdělit takto [15]:

- Nepravidelné [2] - známé, ale v čase se měnící intevaly mezi událostmi.
- Shluky [3] - sekvence událostí libovolně blízko sebe, ale jejich počet je shora ohraničený.
- Pevně ohraničené [4] - sekvence událostí, které jsou od sebe vzdáleny o jistý minimální interval.
- Ohraničené se střední hodnotou [5] - totéž co pevně ohraničené, ale interval je určen střední hodnotou.
- Neohraničené [6] - sekvence událostí, jejichž časové intervaly příchodu je možné určit pouze statisticky. Řídí se potom funkcí hustoty pravděpodobnosti, buď jako vzájemně nezávislý výběr intervalů podléhající společnému rozložení, nebo vzájemně závislé výběry intervalů.

[1] events
[2] irregular
[3] bursty
[4] bounded
[5] bounded average rate
[6] unbounded



Obrázek 2.1: Události – a) nepravidelné, b) shluky, c) pevně ohraničené, d) neohraničené

Systémům, které jsou řízeny příchody externích událostí, říkáme reaktivní [7]. Druhý typ systémů jsou časové systémy, kde chování takovýchto systému je určeno časovými událostmi, většinou periodicky přicházejícími. Jako příklad můžeme uvést běžné operační systémy. V praxi, jelikož RT operační systémy často vycházejí z běžných OS, se setkáme s kombinací časového a reaktivního RT systému.

Jakmile RT systém zachytí událost a převezme ji ke zpracování, neříkáme již, že se jedná o událost, ale o úlohu[8]. Každá úloha v systému je definována základními časovými parametry. Prozatím bude stačit se seznámit jen s časem spuštění úlohy a časovou mezí[9] kladenou na odezvu systému. Na RT systémy jsou kladena tzv. časová omezení[10]. To pro RT systém znamená, že správné fungování nezáleží jen na logických vlastnostech, ale také na vlastnostech časových. Tedy je důležitá správnost výsledku, který nám RT systém poskytne, ale stejně důležité je, zda je výsledek poskytnutý včas.

Definice 2.1.1 Správnost reakcí systému závisí často nejen na logických výsledcích výpočtu, ale také na čase, kdy byly výsledky vyprodukovány. Správné výsledky, které jsou k dispozici příliš pozdě, jsou k ničemu (Stankovic, 1988).

2.1.2 Klasifikace real-time systémů

Jak je uvedeno v definici, tak je velmi důležité, aby byly správné výsledky ve správný čas a na správném místě. Podle toho, co se stane se systémem a jaké následky může mít nedodržení časových omezení na systém a někdy i na jeho okolí, můžeme RT systémy rozdělit. V literatuře zabývající se RT systémy je možné se setkat s rozdělením na hard RT systémy a soft RT systémy [20, 17] (definice převzaty z [19]).

Definice 2.1.2 Hard RT systémy jsou takové systémy, které musí splnit všechny časové meze s téměř nulovým stupněm flexibility. Časové meze musí být splněny, jinak mohou

[7] reactive event-driven
[8] task
[9] deadline
[10] timing constrains

nastat katastrofální následky, které mohou způsobit velké ekonomické, materiální a lidské ztráty. Výsledky získané po časovém termínu mají pro systém buď nulovou užitečnost, anebo s mírou zmeškání rapidně klesá užitečnost pro systém. Nesplnění časových omezení znamená selhání systému.

Definice 2.1.3 Soft RT systém je systém, který musí plnit své časové omezení s určitým stupněm flexibility. Časové meze mohou mít různé úrovně tolerance, a dokonce i statistické rozdělení odezvy s různým stupněm přijatelnosti. V soft RT systémech neznamená nesplnění časových omezení selhání systému, ale v závislosti na aplikaci to může zvýšit zpoždění odezvy systému.

Z definice pro hard RT systémy vyplývá, že byť jedno malé nesplnění časových omezení v systému může způsobit nedozírné následky. Jako příklad můžeme uvést obranný systém navádějící řízené střely. Nebude-li moci naváděcí systém včas vypočítat souřadnice blížící se překážky, tedy výsledky výpočtu budou k dispozici až po časové mezi, nebude již k dispozici dostatečná vzdálenost na to, aby řízená střela mohla změnit dráhu letu. Srážka s překážkou pak muže mít katastrofální následky.

U soft RT systémů nezpůsobí nedodržení časových omezení ztráty na lidských životech, jen vede k dočasné degradaci systému. Jako příklad můžeme použít DVD přehrávač. Úkolem přehrávače je dekódovat obraz, zvuk a patřičně odpovídat na příkazy, zadané uživatelem pomocí ovládání. Množství příkazů, poslaných uživatelem, může přivodit zmeškání některých časových omezení, což může způsobit dočasné zhoršení kvality přehrávaného obrazu či zpoždění zvuku. Jelikož má přehrávač vysokou míru odolnosti proti poruchám, jsou i data získaná pozdě stále užitečná.

Někteří autoři, např. [3], navíc přidávají ještě jednu skupinu, tzv. firm RT systémů. O firm RT systému můžeme říci, že je to mezistupeň mezi hard a soft RT systémy. S hard systémy mají společnou vlastnost, že výsledky došlé pozdě jsou pro systém neužitečné. Na druhou stranu nesplnění časových omezení nepůsobí žádné velké škody. Jako příklad by mohla posloužit nějaká síťová multimediální aplikace, kde paket, který přijde později, je jednoduše přeskočen. Větší počet přeskočených paketů by však mohl způsobit až neúnosnou degradaci sytému.

2.1.3 Determinismus

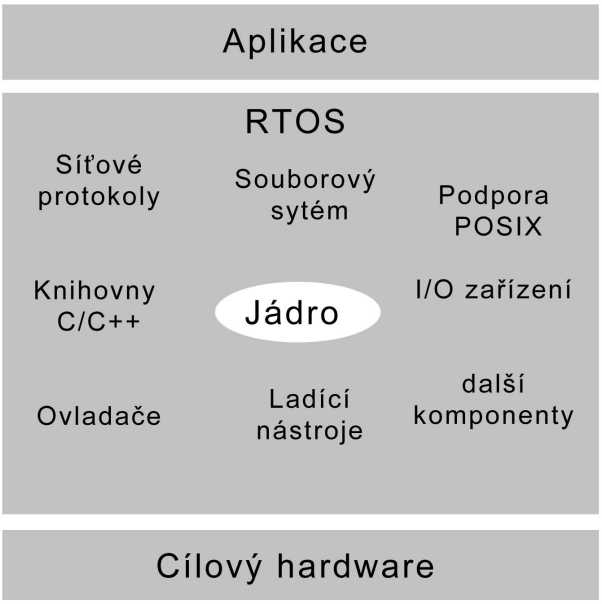
Cílem každého návrhu je, aby navržený systém měl deterministické,tzn. předvídatelné, chování. Systém se chová deterministicky, pokud pro libovolný stav, ve kterém se systém nachází, a libovolnou příchozí externí událost lze jednoznačně určit stav, ve kterém se bude systém následně nacházet, a množinu výstupních odezev. Deterministické systémy je obecně řádově jednodušší navrhnout, ladit a testovat než nedeterministické. Důsledkem toho je snaha, aby všechny RT systémy byly tak deterministické, jak je to jen možné.

2.2 Real-Time operační systém

Real-time operační systém (dále jen RTOS) je program, který plánuje včasné vykonání úloh, řídí systémové zdroje a poskytuje nám pevný základ pro vývoj aplikačního kódu. Aplikační kód může dosahovat různé složitosti, od jednoduchých digitálních stopek, až po komplexní navigační systém letadla. Kvalitní RTOS proto poskytují vysokou míru škálovatelnosti dle požadavků aplikací, ve kterých jsou použity.

2.2.1 Jádro

Každý RTOS má jádro. Jádro je hlavním kontrolním softwarem, který poskytuje minimální logiku, algoritmy plánování a řízení zdrojů. Jednoduché RTOS se mohou skládat jen z tohoto jediného prvku. Komplexnější RTOS však mohou k jádru přidávat ještě další moduly obsahující souborové systémy, síťové protokoly a mnohé další komponenty potřebné pro specifické aplikace, jak je vidět na obrázku.



Obrázek 2.2: Schéma RT systému

Přestože mohou být RTOS více či méně rozšířené podle potřeb aplikace, setkáme se v jádru většiny RTOS se stejnými prvky. Těmito prvky jsou plánovač[11], objekty a služby. Plánovač je součástí každého jádra a poskytuje sadu algoritmů, které úlohám určí čas, kdy budou vykonány. Objekty můžeme popsat jako speciální konstrukce jádra, které pomáhají vývojářům vytvářet aplikace pro RT systémy. Běžnými objekty jádra jsou například úlohy, semafory či fronty zpráv. Služby jsou operace, které jádro provádí nad objekty, nebo obecné operace jako časování, ošetření přerušení a řízení zdrojů.

2.2.2 Plánovač

Jak je již uvedeno výše, srdcem každého jádra je plánovač. Plánovač nám poskytuje algoritmy potřebné pro rozhodnutí, která úloha bude kdy vykonána. Abychom pochopili, jakým způsobem plánovač funguje, je třeba rozebrat ještě některé pojmy.

Nejprve je nutné vědět, co je a co není možné plánovat. Předmětem plánování jsou objekty jádra, která se ucházejí o čas procesoru. Objekt, který těmto kritériím vyhovuje, je úloha. Úlohu můžeme charakterizovat jako nezávislé vlákno vykonávání, které obsahuje posloupnost nezávisle naplánovaných instrukcí. Poznamenejme, že semafory a fronty zpráv,

[11] scheduler

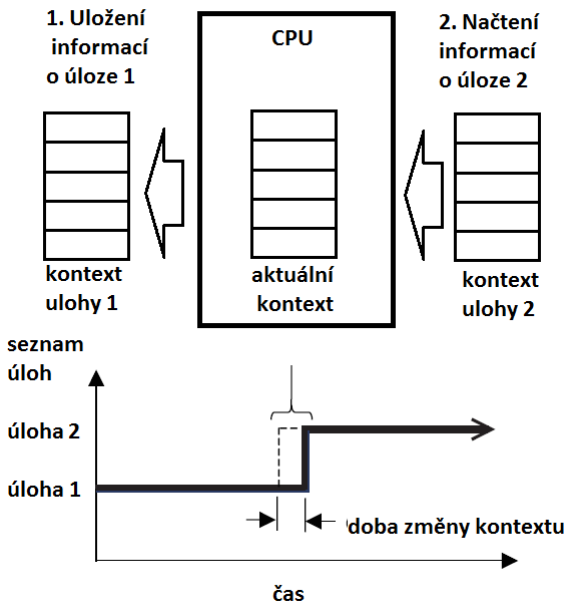
ačkoli jsou objekty jádra, nemohou být plánovatelné. Tyto objekty se používají pro komunikaci a synchronizaci mezi úlohami.

Víceúlohovost

Uvažujme RTOS implementovaný na jednoprocesorové architektuře. V jeden okamžik přijde více požadavků na zpracování úlohy. Jakým způsobem bude možné zpracovat více úloh současně? Vzniklý problém je možné řešit pomocí multitaskingu, nebo-li víceúlohového zpracování. Víceúlohovost RTOS umožňuje obsluhovat více činností v rámci jejich časových omezení. Dostaví-li se více úloh ke zpracování, jeví se jako souběžné zpracování. Ve skutečnosti jádro provádí prokládání úloh podle předem zvoleného plánovacího algoritmu. S počtem úloh potřebných naplánovat rostou i nároky na výkonnost procesoru. Tento fakt je způsoben zvýšením počtu změn mezi kontexty různých úloh.

Změna kontextu

Každá úloha, nacházející se v systému, má svůj kontext. Kontextem se rozumí obsah prvků, které určují aktuální stav procesoru. Jako příklad uveďme obsah programového čítače, obsah příznakového či indexového registru. Změna kontextu nastává, když plánovač přepne[12] z jedné úlohy na jinou. Typický příklad změny kontextu je znázorněný na obrázku 2.3.



Obrázek 2.3: Příklad změny kontextu

Jak je z obrázku zřejmé, musí plánovač nejdříve pozastavit úlohu 1, uložit její kontext, načíst kontext úlohy 2 a spustit ji. Nejen vykonávání úloh, ale i samotná změna kontextu zabere také část času procesoru. Ačkoliv je čas potřebný pro přepnutí z jedné úlohy na druhou ve srovnání s časem vykonávání úloh relativně zanedbatelný, je třeba navrhovat

[12] preempce (preemption)

aplikace tak, aby nedocházelo k častým změnám kontextu. Pokud by k tomu docházelo, způsobovalo by to zbytečné zvýšení režie procesoru.

Dispečer

Velmi důležitou částí plánovače je dispečer[13]. Úkolem dispečera je, aby byl správně změněn kontext a tok řízení výpočtu. V jakémkoliv okamžiku běhu RTOS se tok řízení výpočtu může nacházet v následujících režimech: zpracovávání aplikační úlohy, zpracování přerušení (ISR[14]) a režim jádra. Když přerušení nebo úloha způsobí systémové volání, je tok řízení výpočtu předán jádru a tok vykoná jednu ze svých systémových rutin. Jakmile jádro dokončí vykonávání systémových rutin, je dispečer odpovědný za předání toku řízení některé z aplikačních úloh. To, která úloha bude následně vybrána, není úkolem dispečera, ale plánovacího algoritmu. Úkolem dispečera je přepnout kontext a předat tok řízení úloze.

V závislosti na tom, kdo způsobil systémové volání, mohou nastat různé akce. Je-li systémové volání způsobeno nějakou úlohou, např. je již uvolněný některý ze sdílených zdrojů, předá dispečer řízení jádru, které rozhodne podle plánovacího algoritmu, která úloha poběží jako další. Způsobilo-li systémové volání přerušení ISR, předá také dispečer tok řízení jádru, které musí toto přerušení obsloužit. Jelikož mají HW přerušení vyšší prioritu, nebude při jeho zpracování a současném příchodu dalšího systémového volání dispečer znovu zavolán. Dispečer se použije, až bude celé přerušení dokončeno.

Plánovací algoritmy

Jak již bylo mnohokrát zmíněno, plánovač dle plánovacího algoritmu určuje, která úloha v daný okamžik poběží a jaké sdílené zdroje bude moci využívat. Mohou být použity jak klasické plánovací algoritmy typu FIFO či round-robin, tak speciální algoritmy, které jsou např. popsány v dalších kapitolách.

2.2.3 Důležité vlastnosti RTOS

- Spolehlivost - systém schopný běžet dlouhou dobu bez zásahu člověka. Spolehlivý systém je ten, který běží a nezpůsobuje chyby.
- předpovídatelnost - systém má chování, které je možné matematicky odvodit. Dobrý deterministický systém na stejné požadavky odpoví ve stejném čase jen s nepatrnými odchylkami.
- Výkon - výkon systému závisí jak na hardware, tak i na software. Pro popsání výkonu může být použito mnoho pohledů. Jednou z možností je rychlost, s jakou může systém generovat výsledky založené na přicházejících vstupech. Další možností je rychlost přesunu dat za sekundu, typicky udáváné v bitech za sekundu.
- Kompaktnost - v závislosti na konstrukčních a cenových požadavcích na RT systému musí být požadovaný RTOS malý a účinný. Např. malý levný mobilní telefon s omezenou pamětí nebude potřebovat RTOS s pokročilými funkcemi, které nebude moci využít.

[13] dispatcher
[14] interrupt service routines

- Rozšiřitelnost - RTOS může být použitý pro různě složité aplikace, je tedy vhodné, aby bylo možné přidat k základní funkcionalitě nějaké složitější funkce, případně nepotřebné funkce odstranit.

2.3 Objekty jádra

Objekty jádra jsou konstrukce jádra, které vývojářům poskytují základní prostředky k vytváření aplikací. Tyto objekty můžeme rozdělit do dvou skupin podle toho, zda je tyto objekty možné plánovat, či nikoliv. Mezi plánovatelné objekty patří pouze úlohy. Mezi neplánovatelné patří semafory, fronty zpráv, roury, signály a další. Dále blíže nastíníme funkci jednotlivých objektů. Podrobnější informace o níže popsaných objektech a případně dalších objektech lze najít např. v literatuře[19], ze které bylo čerpáno.

2.3.1 Úlohy

Běžné aplikace jsou typicky navrženy tak, aby běžely sekvenčně a vykonávaly instrukce v předurčeném pořadí. Tento způsob vykonávání programu však není pro RT systémy vhodný, jelikož tyto systémy musí být schopny reagovat na více událostí. Navíc tyto události musí být zpracovány v rámci jejich časových mezí. U těchto systémů potřebujeme souběžné zpracování. Většina RTOS proto poskytují úlohy a služby pro řízení úloh, které ulehčují návrh aplikací pro RTOS.

Úloha je nezávislé vlákno výpočtu, které s ostatními souběžnými úlohami soutěží o výpočetní čas procesoru. Aplikace je rozdělena do více souběžných úloh, aby bylo zaručeno dokončení jednotlivých úloh v rámci jejich časových omezení. Každá úloha v systému je definována sadou parametrů. Mezi těmito parametry může být např. jednoznačný identifikátor úlohy ID, jméno, priorita, rutiny úlohy či stav plánování. Podle druhu implementace jádra se mohou parametry lišit. V některých jádrech se nemusí vyskytovat například priorita.

Každá úloha v systému se může nacházet v jednom z následujících stavů:

- Připravena - úloha je připravena k běhu, ale nemůže běžet, protože je vykonávána jiná úloha.
- Blokována - úloha potřebuje ke svému běhu sdílený zdroj, který není momentálně dostupný. Musí počkat, dokud nebude uvolněn.
- Běžící - úloha, která je právě vykonávána procesorem.

V různých literaturách je možné se setkat i s obsáhlejším rozdělením. Například toto rozdělení může být rozšířeno o stavy odložené[15], nevyřešené[16] a zpožděné[17]. V tomto případě jsou stavy zpožděné a nevyřešené rozdělením stavu blokované.

2.3.2 Semafory

Při běhu mnoha souběžných vláken výpočtu je potřeba zajistit vzájemně výlučný přístup ke sdíleným prostředkům. Pro tyto případy používáme semafor. Semafor je objekt jádra, který může jedna či více úloh získat či uvolnit pro účely synchronizace anebo vzájemného vyloučení.

[15] suspended
[16] pended
[17] delayed

Semafor typicky obsahuje jednoznačný identifikátor, hodnotu a seznam čekajících úloh. Pokud může hodnota nabývat pouze hodnot 0 a 1, říkáme takovému semaforu binární. Do binárního semaforu může vstoupit maximálně jedna úloha. Pokud však může hodnota nabývat více jak 2 hodnot, jedná se o tzv. čítací semafory. V tomto případě může do kritické sekce vstoupit více jak jedna úloha. Pro nastavování hodnot semaforu se používají operace wait a signal. Tyto operace jsou implementovány jako atomické[18].

2.3.3 Fronty zpráv

Fronty zpráv jsou objekty jádra používané ke komunikaci a výměně dat mezi úlohami. Často bývají implementovány jako buffer, případně mohou být uloženy v předem určené části paměti. Odeslaná zpráva čeká na svého příjemce, dokud si ji nevyzvedne. U zpráv je možné určovat priority a tím předběhnout zprávy s nižší prioritou. Fronta zpráv má zpravidla omezenou velikost, prvky označující začátek a konec fronty. Odeslání či přijmutí zpráv neblokuje procesor. Problémy mohou nastat, pokud chceme číst zprávu z prázdné fronty nebo posíláme-li zprávu do fronty plné.

2.3.4 Roury

Podobně jakou fronty zpráv se i roury používají k synchronizaci úloh a výměně dat mezi nimi. Obvykle je roura implementovaná jako příslušenství k jednosměrné výměně dat. Na jedné straně se data zapisují a na druhé čtou, proto jsou využívány dva různé deskriptory. Do roury můžeme zapisovat více úloh souběžně, stejně tak z ní číst. Data proudící rourou nejsou strukturovaná, ale stává se z nich datový proud. Čtení z tohoto datového proudu se provádí metodou FIFO, na rozdíl od fronty zpráv, kde je možné některé zprávy předběhnout. Výhodou oproti frontě zpráv je možnost čekat, dokud se v rouře nevyskytnou nějaká data, čímž dojde k zablokování úlohy.

2.3.5 Signály

Signály se chovají podobně jako přerušení. Hlavním rozdílem je, že přerušení přichází z vně systém a signál je generován nějakou úlohou uvnitř systému. Stejně jako u normálních přerušení mohou signály přicházet asynchronně. Příchod signálu oznamuje úloze, že se během normálního výpočtu vyskytla nějaká událost, na kterou bude třeba nějakým způsobem zareagovat. Počet a čísla jednotlivých signálů se mohou podle implementace RTOS lišit. Každý signál v systému je spojen s nějakou událostí. Události mohou být vyvolány neúmyslně, jako např. nastala neočekávaná chyba při výpočtu. Také mohou být vyvolány úmyslně, chce-li úloha oznámit něco jiné úloze. Každá úloha může určit, jakým způsobem na příchozí signál zareaguje.

[18] tzn., že provádění těchto operací nemůže být jakkoliv přerušeno

Kapitola 3

Plánovací algoritmy

Mějme množinu úloh, které je potřeba vykonat. Úloha je obvykle charakterizována dobou potřebnou pro běh, časem požadavku na spuštění, časovou mezí úlohy a případně požadavky na zdroje. Běh úlohy může a také nemusí být přerušen podle typu plánování. Hlavním úkolem plánování je určit pořadí, aby byly splněny časové meze všech úloh. Plánování úloh se neřídí jen podle časových mezí, může být dále ovlivněno snahou přístupu ke sdíleným zdrojům či závislostí mezi úlohami, kdy je potřeba, aby byly nejprve dokončeny všechny předcházející úlohy. Celkově můžeme cíle a požadavky na plánování shrnout následovně:

- dodržení všech časových omezení úloh,
- zabránit současnému přístupu ke sdíleným zdrojům,
- snižovat celkový čas potřebný ke změně kontextu, způsobný preempcí,
- zajistit bezpečnost a spolehlivost.

3.1 Klasifikace plánovacích algoritmů

Způsob plánování by měl být pro každý RTOS vhodně zvolen podle vlastností systému a úloh v něm se vyskytujících. Tyto vlastnosti rozdělme následovně:

- soft/hard/firm,
- periodické/aperiodické/sporadické,
- preemptivní/nepreemptivní,
- jednoprocesorové/víceprocesorové,
- fixní/dynamické priority,
- statické/dynamické (on-line/off-line),
- závislé/nezávislé.

Soft, hard, firm

Nároky kladené na úlohy v systému mohou být rozděleny na soft, hard či firm. Tyto vlastnosti byly popsány v odstavci 2.1.2.

Periodické, aperiodické, sporadické

Úlohy, stejně jako události 2.1.1, mohou být podle pravidelnosti jejich spouštění rozděleny na periodické, aperiodické a sporadické.

Periodické úlohy jsou takové úlohy, které pravidelně přicházejí s pevnou periodou. Časové omezení periodických úloh nám říká, že úloha musí být provedena vždy jednou za periodu.

Aperiodické úlohy jsou takové úlohy, které přichází nepravidelně s možným pevným ohraničením mezi příchody 2.1.1. Splnění aperiodické úlohy je podmíněno časovou mezí úlohy.

Sporadické úlohy přichází také nepravidelně, ale se známým pevným ohraničením. Pevné ohraničení je charakteristické minimálním časem mezi příchody dvou po sobě jdoucích úloh. Splnění sporadické úlohy je podmíněno časovou mezí úlohy.

Preemptivní, nepreemptivní

Preemptivní plánování dovoluje, aby právě prováděná úloha byla přerušena. Přerušení znamená odebrání úlohy procesoru a vyhrazení času procesoru jiné naléhavější úloze. Přerušená úloha je pozastavena a až bude později opět vybrána k běhu, bude pokračovat od místa, kde byla pozastavena.

Naopak nepreemptivní plánování nedovoluje úlohu, která je prováděna, přerušit. Jakmile se začne úloha provádět, je možné spustit další úlohu až po dokončení předešlé. U tohoto přístupu plánovaní je těžší plnit časová omezení úloh.

Jednoprocesorové, víceprocesorové

V závislosti na počtu procesoru je potřeba vhodně vybírat plánovací algoritmy. U více procesorů je třeba, aby algoritmy zamezily souběžnému přístupu ke sdíleným zdrojům. Dále se musí určit strategie, dle které budou procesory mezi sebou vzájemně komunikovat, aby nebylo spotřebováno mnoho času procesorů ke komunikaci. Tyto vzniklé problémy naopak není třeba řešit u jednoprocesorových architektur.

Fixní a dynamické priority

V prioritním plánování má každá úloha přiřazenou prioritu. Priority mohou být pevně dané a za žádných okolností se nemění po celou dobu běhu systému. Druhou možností jsou dynamické priority, které se mění za běhu systému.

Statické, dynamické (on-line/off-line)

Aby bylo možné naplánovat úlohu, je třeba mít o úloze nějaké informace. Statické plánování se používá, když víme, kdy která úloha do systému přijde ještě před jeho samotným spuštěním. Nevýhodou tohoto plánování je předpoklad neměnnosti parametru úloh a nepřizpůsobivost ke změnám. Na druhou stranu dynamické plánování nepředpokládá výskyt úloh. Plánování pak probíhá podle výskytu jednotlivých úloh či analýzy vlastností úloh. Výhodou je přizpůsobivost plánu na změny. Nevýhodou pak složitější implementace a nároky na plánování. Můžeme se setkáme i s kombinací dynamického a statického plánování.

Závislé, nezávislé

Uvažujme RT systém, ve kterém může úloha začít, až když dostane výsledky předešlé úlohy. Úloze předávající své výsledky úloze následující říkáme předek. Má-li úloha nějaké předky, nazýváme ji precedenčně závislou úlohou. Další závislost může nastat, bude-li více úloh usilovat o sdílený zdroj. Této závislosti říkáme závislost daná vzájemným vyloučením. Úlohu nazveme nezávislou, pokud nemá žádné předky a neusiluje o žádný sdílený zdroj [23].

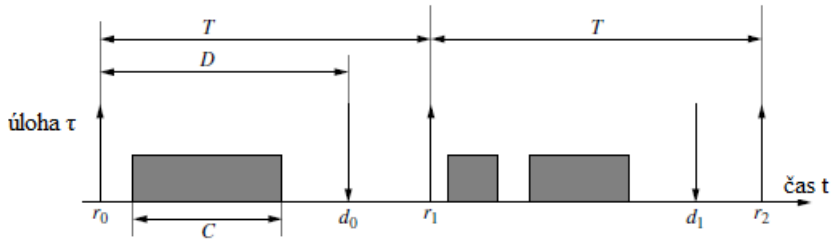
3.2 Model RT úlohy

3.2.1 Základní parametry RT úlohy

Úlohy v RT systému jsou základním prvkem, který je třeba plánovat, aby byla splněna časová omezení každé úlohy. Úlohy mohou mít hard, firm či soft časová omezení. Dále mohou úlohy být periodické, anebo aperiodické, s pevnou, dynamickou, či žádnou prioritou, závislé či nezávislé. Model RT úlohy pak zahrnuje všechny důležité parametry potřebné k plánování. Uveďme základní parametry potřebné k plánování:

- čas obdržení požadavku na spuštění úlohy r_i,
- čas potřebný pro výpočet procesorem bez přerušení C_i,
- absolutní časová mez d_i, je nejzazší čas, kdy musí být úloha dokončena,
- aby nedošlo k chybě sytému,
- relativní časová mez D_i udává rozdíl mezi absolutní časovou mezí d_i a časem
- spuštění r_i: $D_i = d_i - r_i$,
- perioda volání úlohy T (udává se pouze u periodických úloh).

Každou úlohu τ budeme popisovat základními parametry následovně: $\tau(r, C, D, T)$ pro periodické úlohy a $\tau(r, C, D)$ pro úlohy aperiodické. V diagramech bude parametr r, čas příchodu požadavku na spuštění úlohy, označen symbolem $\uparrow$ a parametr d_i, absolutní časová mez úlohy, označen symbolem $\downarrow$. Pokud jsou si parametry úlohy D a T rovny, bude použit symbol $\updownarrow$. Jednotlivé parametry úloh a jejich symboly ilustruje obrázek 3.1.



Obrázek 3.1: Model úlohy

Správné hodnoty základních parametrů úloh jsou pro plánování velmi důležité. Nemohou-li být časové prodlevy způsobené na změny kontextu, obsluhy přerušení, systémových volání

či plánovačem zanedbány, musí být zahrnuty do parametrů příslušné úlohy (obvykle parametr C). Toto je důvodem požadavku, aby jádro RTOS mělo deterministické chování, kde je možné určit doby těchto prodlev.

3.2.2 Dynamické parametry RT úlohy

Kromě základních parametrů je možné o úlohách získat další tzv. dynamické parametry. Tyto parametry rozšiřují informace o úloze, které napomáhají lépe pochopit principy provádění a plánování úloh. Mezi tyto parametry patří:

- čas provádění úlohy s_i je čas, kdy byla úloha předána procesoru k vykonání,
- čas ukončení provádění úlohy f_i je čas, ve kterém byl výpočet úlohy dokončen,
- čas odezvy R_i, je doba mezi časem ukončení úlohy f_i a časem spuštění úlohy r_i: $R_i = f_i - r_i$
- zbytková relativní časová mez úlohy v čase $D(t) = d - t$, udávající, kolik času ještě zbývá do uplynutí časové meze úlohy,
- zbývající čas potřebný k dokončení úlohy v čase $C(t)$,
- relativní doba volnosti úlohy $L = D - C$, udávající maximální dobu prodlevy před začátkem provádění úlohy při neomezeném přístupu k procesoru,
- zbytková relativní doba volnosti úlohy v čase $L(t) = D(t) - C(t)$,
- zbytkové zatížení $CH(t) = \frac{C(t)}{D(t)}$

3.2.3 Množina úloh

Pro účely modelování je RT aplikace reprezentována množinou úloh postupně, či současně spouštěných. Pokud jsou úlohy spuštěny současně, mají stejný čas obdržení požadavku na spuštění r_i. Takovým úlohám říkáme, že jsou ve *fázi*. Níže jsou ještě uvedeny některé významné pojmy, týkající se plánování množin úloh.

Činitel vytížení procesoru U množiny n periodických úloh popsaný vztahem

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{3.1}$$

Činitel zatížení procesoru CH množiny n periodických úloh popsaný vztahem

$$CH = \sum_{i=1}^{n} \frac{C_i}{D_i}. \tag{3.2}$$

3.2.4 Další vlastnosti plánování

Než přejdeme k samotným plánovacím algoritmům, uveďme vlastnosti ještě některých pojmů souvisejících s nimi.

- Plán je produkt plánovače předepisující časování stavových přechodu jednotlivých úloh.

- Plán označíme za přípustný, jsou-li splněny všechny časové meze úloh.
- Množina úloh je plánovatelná, pokud je k ní možné sestavit přípustný plán.
- Plánovací algoritmus označíme za optimální, pokud je schopen vytvořit přípustný plán pro jakoukoliv plánovatelnou množinu úloh.
- Test plánovatelnosti umožňuje rozhodnout, zda množina periodických úloh je, či není plánovatelná.
- Test přijetí umožňuje při dynamickým plánování rozhodnout, zda i po přidání dané úlohy do množiny úloh zůstane množina úloh plánovatelná.

3.3 Základní algoritmy pro plánování úloh

Níže jsou popsány dva velmi často používané algoritmy plánování úloh. Tyto algoritmy jsou často použity jako základ pro složitější algoritmy. Další plánovací algoritmy je možné nalézt např. v [3], [6] či [26].

3.3.1 RM

Rate-Monotonic (dále RM) je jednoduchý algoritmus pro plánování periodických úloh, který každé úloze přiřadí prioritu podle frekvence příchodu požadavků na spuštění úlohy. Úlohy s vyšší frekvencí příchodu (s kratší periodou) budou mít vyšší prioritu. Jelikož se periody nemění, je každé úloze přiřazena priorita, která se již v čase nemění. Algoritmus RM je optimální pro všechny množiny úloh s pevným přiřazením priorit(důkaz je možné nalézt např. v [3, str. 87-89]).

Pro plánování se využívá pouze pevně stanovených priorit a preempce. Naplánovány jsou vždy úlohy s nejvyšší prioritou. Pokud běží úloha A a přijde úloha B s vyšší prioritou, je úloha A pozastavena a spuštěna úloha B. Postačující podmínkou pro plánovatelnost množiny úloh je (odvození je možné nalézt v [6, str. 27-29]):

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n * (2^{1/n} - 1) \tag{3.3}$$

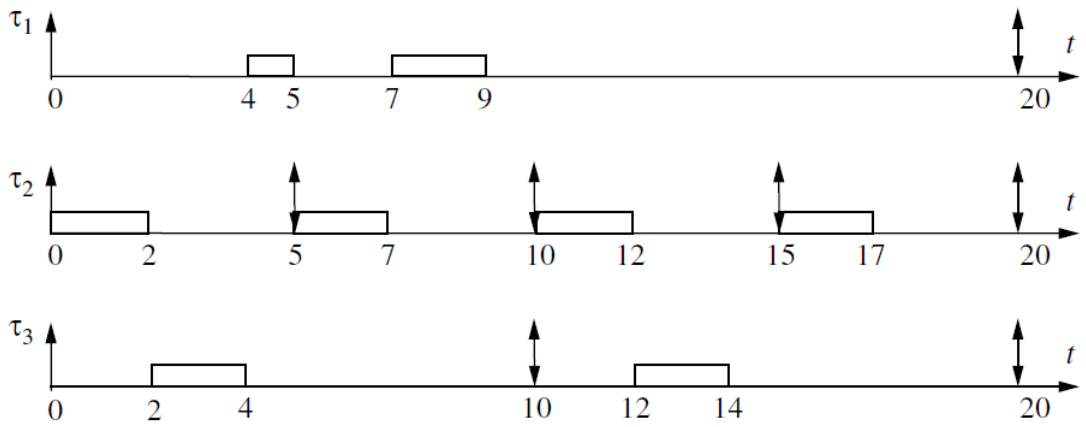
Příklad plánování tří periodických úloh pomocí algoritmu RM je znázorněn na obrázku 3.2:

3.3.2 EDF

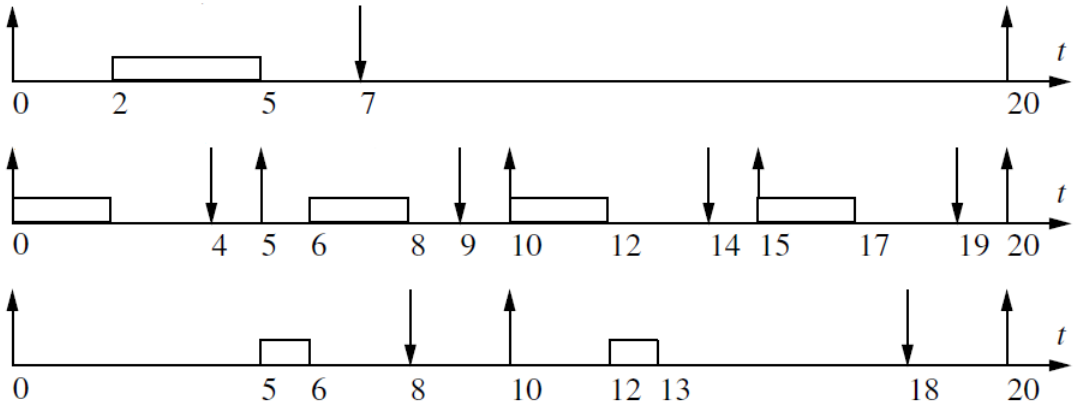
Earliest Deadline First (dále EDF) je nejběžnější algoritmus pro plánování úloh s dynamickou prioritou. Přiřazování priorit úlohám je založeno na zbytkové relativní časové mezi úlohy v čase $D(t)$. Nejvyšší priorita bude přiřazena úloze, které v čase t zbývá nejkratší doba k uplynutí jejich časových mezí. EDF algoritmus je obvykle preemptivní, je však možné setkat se i s nepreemptivní implementací.

Vzhledem k tomu, že je plánování založeno na parametru $D(t)$, může být algoritmus obecně použit pro plánování jak periodických, tak aperiodických úloh. Množina periodických úloh je plánovatelná tehdy a jen tehdy, je-li činitel vytížení procesoru $U \leq 1$(důkaz v [3, str. 100 a 101]). Pro množinu hybridních úloh je postačující podmínkou $CH \leq 1$ a nutnou podmínkou $U \leq 1$ [6].

Na obrázku 3.3 je znázorněno plánování tří periodických úloh pomocí algoritmu EDF.



Obrázek 3.2: Příklad RM plánování pro úlohy $\tau_1(0, 3, 20, 20)$, $\tau_2(0, 2, 5, 5)$ *a* $\tau_3(0, 2, 10, 10)$



Obrázek 3.3: Příklad EDF plánování pro úlohy $\tau_1(0, 3, 7, 20)$, $\tau_2(0, 2, 4, 5)$ *a* $\tau_3(0, 1, 8, 10)$

Kapitola 4

Nízkopříkonové plánovací mechanismy

Spotřeba energie se stává velmi důležitým faktorem při návrhu moderních vestavěných systémů, zvláště pak u mobilních systému, které pracují s omezeným zdrojem energie, jako jsou baterie. U těchto systémů se proces návrhu systému vyznačuje kompromisem mezi vysokým výkonem a nízkým příkonem. Důraz je pak kladen, aby byly splněny všechny omezení úloh při minimálním příkonu. Snížení příkonu pozitivně ovlivňuje nejen životnost baterii u přenosných zařízení, ale snižuje i potřebné prostředky pro chlazení systému samotného. Techniky návrhu nízkoenergetických obvodů napomáhají snížit spotřebu energie, avšak dynamické řízení příkonu na vyšších úrovních návrhu systému dokáže tuto spotřebu ještě více snížit. Cílem této kapitoly bude popsat vybrané techniky plánování, které umožňují měnit příkon systému a tím i lépe nakládat s energií.

4.1 Příkon a CMOS

Abychom porozuměli návrhu nízkoenergetických systémů, je třeba porozumět fyzickým vlastnostem obvodů, které vedou ke spotřebě energie či výkonové ztrátě.

4.1.1 Příkon

Velká část dnešních číslicových obvodů je implementována pomocí technologie CMOS[1]. Schéma CMOS hradla je znázorněno na 4.1. Ačkoli zde budou popsány vztahy pro CMOS hradla, lze tyto vztahy zobecnit i pro jiné technologie.

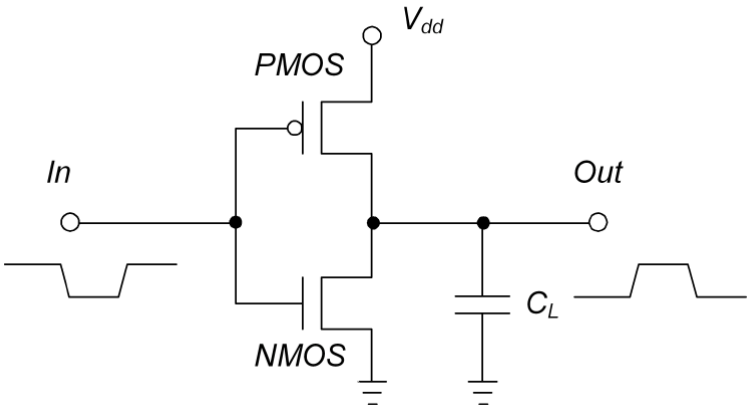
Příkon CMOS hradla můžeme rozdělit do dvou základních složek, statickou a dynamickou:

$$P_{CMOS} = P_{statick} + P_{dynamick} \tag{4.1}$$

Statická složka

V ideálním případě, pokud je hradlo v ustáleném stavu a nevede žádná cesta od zdroje napětí k zemi, je statická složka příkonu hradla CMOS nulová. Takový stav však v praxi není a přes MOS tranzistory vždy uniká nějaké napětí. Tento statický příkon $P_{statick}$ je

[1] Complementary metal-oxide-semiconductor



Obrázek 4.1: Schéma obvodu CMOS

popsán rovnicí 4.2, kde I_{diode} je zpětný diodový proud mezi difúzními oblastmi a substrátem, $I_{subthreshold}$ je ztrátový proud tranzistoru a U_{dd} je napájecí napětí. Velikost $P_{statick}$ je v čase konstantní. Z tohoto důvodu jí nebývá často věnována přílišná pozornost.

$$P_{statick} = (I_{subthreshold} - I_{diode})U_{dd} \tag{4.2}$$

Dynamická složka

Dynamický příkon hradla lze spočítat jako součet složek PSW a PSC.

Složka PSW se nazývá *příkon přepínaných kapacit*. Tento příkon je způsoben nabíjením a vybíjením zátěžových kapacit. Toto nabíjení a vybíjení je způsobeno přechody mezi logickými hodnotami($0 \rightarrow 1$, $1 \rightarrow 0$). Složka PSW tvoří 85-90% dynamického příkonu.

Složka PSC se nazývá *příkon způsobený krátkým spojením*, který je způsoben průchodem zkratového proudu. Tento stav nastává při přechodu z jedné logické hodnoty do druhé, kdy jsou na krátký čas otevřeny oba tranzistory NMOS i PMOS. Složka PSC tvoří zbylých 10 - 15% dynamického příkonu.

Po odvození z rovnic, které je možné nalézt v [28] a zjednodušení, můžeme pro dynamický příkon CMOS obvodu použít rovnici:

$$Pdynamick \sim \alpha f C_L U_{dd}^2, \tag{4.3}$$

kde α je počet překlopení ($0 \rightarrow 1$, $1 \rightarrow 0$) v obvodu v rámci předdefinované časové periody, f je frekvence synchronizačního signálu, C_L je kapacita hradel a U_{dd} je napájecí napětí. Všechny techniky pro snížení spotřeby energie a příkonu se snaží ovlivňovat tyto faktory. Bohužel jsou všechny tyto složky nějakým způsoben provázány a snížení jedné může mít nežádoucí účinky na ostatní složky. To znamená, že snižování příkonu je záležitost, se kterou je třeba pracovat nanejvýš obezřetně.

Vycházíme-li z rovnice, jeví se snížení napájecího napětí U_{dd} jako vhodné místo pro optimalizace, protože jeho závislost je kvadratická. Snížení napětí o polovinu sníží příkon čtyřikrát. Bohužel toto snižování s sebou přináší další úskalí. Pokud se U_{dd} blíží prahovému napětí, zvýší se zpoždění obvodu. Toto je způsobeno delším časem potřebným pro nabíjení a vybíjení výstupní kapacity C_L. Vzhledem k těmto okolnostem je dolní limit pro napájecí

napětí stanoven kolem $2 * U_{dd}$ [12]. Toto snížení výkonu může být zmírněno pracováním s nižším prahovým napětím, avšak tento postup vytváří další možné problémy. Pokud se prahové napětí stane příliš malé, zvýší se úniky přes MOS tranzistory a zároveň se zvýší statický příkon tranzistoru.

4.1.2 Spotřeba energie

Nyní se zaměřme na spotřebu energie E. Ačkoliv nízký příkon a energetická účinnost mají podobné cíle, postupy, kterými jich dosahují jsou rozdílné. Každý výpočet má stanovený nějaký čas, kdy musí být dokončen. Spotřeba energie E bude nižší, pokud čas potřebný pro výpočet bude nižší, anebo pokud bude nižší příkon. V porovnání s příkonem zahrnuje rovnice pro spotřebu energie 4.4 i čas. Pokud bychom snížili příkon a zároveň zvýšili výpočetní čas, mohlo by nakonec místo snížení dojít ke zvýšení spotřebované energie.

$$E = P(t)dt \tag{4.4}$$

Pokud bychom například snížili příkon na polovinu a zároveň zdvojnásobili výpočetní čas pro operaci, nevedlo by to k žádné změně spotřeby energie. Na druhou stranu napájecí napětí působí na horní mez frekvence synchronizačního signálu. Z tohoto důvodu se napětím U_{dd} a frekvencí f zabýváme společně. Poznamenejme, že nižší spotřeba často značí pomalejší systémy. RT plánování a minimalizace energetické spotřeby jsou velmi příbuzné problémy a k dosažení nejlepších výsledků je potřeba je řešit společně.

4.2 Klasifikace nízkopříkonových technik

Přístupy, kterými je možné snížit energetickou náročnost systému, mohou být rozděleny do dvou kategorií: dynamic voltage scaling a dynamic power management (dále jen DVS resp. DPM).

DVS může být použito jak na úrovni návrhu hardwaru, tak na úrovni návrhu softwaru. Jedná se o jednu z nejúčinnějších technik pro snížení spotřeby energie E systémů založených na CMOS, např. VLSI. Snižováním napájecího napětí V_{dd} je možné docílit značné redukce E, jak bylo uvedeno výše. Pokud úloha nevyžaduje pro splnění všech požadavků maximální výkon systému, může být dynamicky snížena frekvence f (a tím i napětí V_{dd}) na nejnižší možnou úroveň, při které budou stále splněny požadavky. Tímto postupem snížíme spotřebu energie, aniž by byly patrné jakékoliv změny v chování systému.

DPM se k problému, jak ušetřit energii, staví zcela odlišným přístupem. Princip je založen na tom, že při běhu úloh v systému se střídá doba provádění úloh s dobou, kdy je procesor nečinný (IDLE). Při nečinnosti odebírá procesor stále stejné množství energie, ačkoliv nevykonává žádnou prospěšnou práci. Úkolem pro DPM je přepnout procesor do tzv. nízkoenergetického [2] stavu, při kterém odebírá podstatně méně energie. Pokud je procesor v tomto stavu, říkáme, že spí. Aby mohl být procesor uspán či vzbuzen, je samozřejmě zapotřebí vynaložit nějaký čas a energii. Problémem při tomto mechanismu je předpovědět, bude-li doba nečinnosti procesoru dostatečně dlouhá pro uspání a vzbuzení procesoru.

[2] low-power

4.3 DVS

Snižováním napětí a s tím spojené i maximální frekvence se DVS snaží docílit snížení spotřeby energie. Podle toho, za jakých okolností k tomu dochází, můžeme DVS rozdělit na intraúlohové a interúlohové (dále jen intra resp. inter). Hlavním rozdílem je, jak algoritmy nakládají s volným časem[3]. IntraDVS algoritmus spotřebuje všech možný volný čas v rámci mezí vykonávání vlastní úlohy, čímž sníží rychlost vykonávání úlohy sama sobě. Naopak InterDVS použije volný čas pro úlohy následující, tedy sníží se rychlost vykonávání úloh následujících.

4.3.1 IntraDVS

Úkolem IntraDVS algoritmu je určit místa v programu úlohy, kde by mělo dojít ke změně napětí. Podle toho, jakým způsobem se tyto místa změn napětí hledají, rozdělujeme IntraDVS do pěti skupin následovně:

Segment based rozděluje úlohu do několika segmentů. Po provedení segmentu se nastaví napětí a frekvence tak, aby byly využity volné časy segmentů, které již byly vykonány. Tento postup byl rozšířen na tzv. spolupracující IntraDVS, kde OS a kompilátor pracují společně. Kompilátor při překladu rozdělí úlohu na segmenty a označí je dočasnými značkami. OS pak mění napětí podle chování systému a těchto značek.

Path based využívá k nalezení míst vhodných pro změnu napětí tok řízení[4] (CFG graf) programu. Tok řízení určí všechny místa v programu, kde budou volné prostředky, a vloží informace o změně napětí již při překladu aplikace. Při tomto způsobu plánování je třeba řešit problémy s předvídáním délky prováděných cyklů a jakým způsobem určit místa, kde by mělo dojít ke změně napětí.

Memory aware využívá k určení míst pro změnu napětí místa, kde je procesor nečinný kvůli blokování externí paměti.

Stochastické IntraDVS využívá pravděpodobností informace o výpočetním čase programu. Tato metoda založena na názoru, že z pohledu spotřeby energie je lepší "začít na pomalé rychlosti a v případě potřeby výpočet zrychlit", než "začít na maximální rychlosti a v případě nalezení volného času výpočet zpomalit".

Hybridní postup odstraňuje omezení všech výše uvedených přístupů. Předešlé přístupy nehledí na systém z globálního hlediska a neberou v úvahu možnost, kdy nastane k souběžnému zpracování. Hybridní mechanismus používá k plánování kombinaci intra a InterDVS algoritmů.

4.3.2 InterDVS

Algoritmy InterDVS berou v úvahu, že se v sytému může vyskytovat více úloh, které jsou připraveny ke spuštění. Cílem pak je pro množinu těchto úloh zvolit rychlost tak, aby byla splněna časová omezení všech úloh. Výhodou, vycházející z tohoto přístupu, je možnost rovnoměrného rozložení zatížení napříč úlohami a dobou vykonávání. Důsledkem toho je delší čas strávený v nižších rychlostech procesoru a tím i nižší spotřeba energie.

Změna pohledu v InterDVS na systém jako množinu úloh s sebou přináší krom zvolení rychlosti pro úlohu i rozhodnutí, kdy bude každá z úloh spuštěna. Můžeme tedy říci, že úkolem je přidělení každé úloze času (plánování), ve kterém bude vykonána při nejvýhodnější energetické spotřebě (zvolit nejnižší možnou rychlost).

[3] slack time
[4] control flow

Typicky na úrovni množiny úloh se InterDVS nezabývá, jakým způsobem nakládají úlohy s přiděleným časem procesoru. Předpokládá se, že úlohy si dokáží zvolit optimální rychlost výpočtu. V tomto smyslu se IntraDVS a InterDVS vzájemně doplňují. Ačkoli se může zdát, že kombinace obou systémů by mohla podávat nejlepší výsledky, není tomu vždy tak. Výhodou se jeví, že pokud IntraDVS provede špatné rozhodnutí, může jej InterDVS napravit. Bohužel toto chování s sebou přináší zvýšení režie na plánování, což může ve skutečnosti spotřebovat více energie, než se povede ušetřit.

Klasifikace

Jelikož většina InterDVS algoritmů je založena na prolnutí více technik, není rozdělení do skupin jednoduché. Než rozdělování podle celkového přístupu, rozdělme je podle plánovacích rozhodnutí. Můžeme je tedy dělit v rozsahu od plně statických, kdy veškerá rozhodnutí o plánování a rychlosti jsou provedena off-line, až po plně dynamická, kde se veškeré rozhodnutí dělají až za běhu sytému.

4.4 Statické InterDVS algoritmy

Statické mechanismy jsou často velmi jednoduché. Tyto mechanismy vybírají nejnižší možnou frekvenci, při které bude stále možné splnit časové meze dané množiny úloh. Frekvence je přiřazena staticky a již se dále nemění, pokud nenastanou nějaké změny v množině úloh. Dále si popíšeme některé statické algoritmy.

Statické RM/EDF

Tyto mechanismy přiřadí nejnižší možnou frekvenci procesoru, při které bude množinu úloh ještě možné naplánovat pomocí klasických mechanismů EDF či RM. K nalezení vhodné frekvence f musíme nejdříve nalézt faktor $\alpha(0 < \alpha \leq 1)$, kterým bude frekvence snížena $1/\alpha$. Pro plánování pomocí EDF musí být dále splněná nutná podmínka plánovatelnosti $U \leq 1$. Při použití snížené frekvence pomocí faktoru α je pak nutná podmínka $U \leq \alpha$.

Podobný způsob se použije pro plánovaní pomocí RM. Zde musí být splněna nutná podmínka plánovatelnosti uvedená v[5, str. 47]. Následně je vybrána nejnižší možná frekvence, pro kterou je ještě splněn test plánovatelnosti. Pracovní frekvence je pak upravena na nejbližší možnou, která je podporována procesorem. Má-li procesor m pracovních frekvencí $f_1, f_2, \ldots, f_m$ takových, že $f_1 < f_2 < \cdots < f_m$, je vybrána nejniží frekvence, při které plánování ještě splní časové meze všech úloh [24].

Algoritmus 4.1 popisuje celkové chování statických plánovacích mechanismů jak pro EDF, tak RM. Pomocí těchto metod můžeme snížit příkon, avšak je zde stále mnoho nevyužitého potenciálu. Algoritmy neberou v úvahu, že může některá úloha být vypočtena za nižší čas, než je C. Další statické algoritmy je možné nalézt v [4].

MRS

Maximum Required Speeds (dále jen MRS) metoda využívá statická data o úlohách k tomu, aby vylepšila plán běhu pomocí off-line rozhodnutí. Cílem je dosažení maximálního využití procesoru, ideálně 100%, identifikováním úloh, které je možné spustit pomaleji. Přesněji MRS není samo o sobě plánovací algoritmus, ale postup, podle kterého se vypočítá nejnižší možná pracovní rychlost, při které bude ještě množina úloh pro určitý on-line algoritmus stále plánovatelná. Plánování úloh probíhá za běhu (on-line), avšak rychlost, při které bude

```
EDF_test(α):
   if(C_1/T_1 + ··· + C_n/T_n ≤ α) return true;
   else return false;

RM_test(α):
   if(∀τ_i ∈ {τ_1, ..., τ_n|T_1 ≤ ··· ≤ T_n})
      ⌈T_i/T_1⌉ * C_1 + ··· + ⌈T_i/T_i⌉ * C_i ≤ α * T_i
   return true;
   else return false;

vyber_frekvenci:
   použij nejnižší frekvenci f_i ∈ {f1, ..., f_m|f_1 < ··· < f_m}
   takovou, že RM_test(f_i/f_m) nebo EDF_test(f_i/f_m) je true.
```

Tabulka 4.1: Algoritmy pro statické EDF a RM

úloha spuštěna, je stanovena podle hodnot vypočtených off-line (před spuštěním systému). Tato technika může být použita na řadu klasických plánovacích strategií.

MRS for RM scheduling Budeme-li vycházet z podmínky pro plánování pomocí RM 3.3.1, můžeme určit maximální rychlost, při které bude ještě množina úloh plánovatelná, podle rovnice 4.5:

$$s_{RM_M RS} = \frac{U}{N * (2^{1/N} - 1)} \tag{4.5}$$

Tato podmínka plánovatelnosti bere v úvahu pouze nejhorší možné případy popisující množinu úloh. Bližší analýza, popsaná v [4] či [17], odhalila ještě další možnosti pro prodlužování výpočetního času úloh při splnění jejich časových omezení. Na základě těchto informací můžeme nejen spočítat novou maximální rychlost pro množinu úloh, můžeme dokonce spočítat rychlost vykonávání pro každou úlohu zvlášť. Poznamenejme, že rychlost potřebná pro vykonání úlohy je nepřímo úměrná k faktoru rozpínání. Nalezení maximální potřebné rychlosti je tedy ekvivaletní k nalezení nejmenšího faktoru rozpínání.

Faktor rozpínání vypočítáme iterační metodou, počínaje u úloh s nejvyšší prioritou a pokračujíce k úlohám s prioritou nižší. Dále položme předpoklad, že úlohám τ_i v množině úloh $\tau_{i i=1,\ldots,N}$ je index přiřazován podle jejich priority, která se u RM plánování vypočte jako $1/T_i$. Nechť q ukazuje na poslední úlohu, které byl přiřazen faktor rozpínání a počáteční hodnota je $q = 0$. Každá z úloh $\tau_{q<i\leq N}$ musí být vykonána před jejím bodem plánování S_i definovaným:

$$S_i = \{kT_j | 1 \leq j \leq i \wedge 1 \leq k \leq \lfloor \frac{T_i}{T_j} \rfloor\} \tag{4.6}$$

Rovnice 4.6 platí pouze pro případy, kdy je $T_i = D_i$. Pro množiny úlohy, kde je $T_i \neq D_i$, je třeba změnit posloupnost bodů plánování následovně:

$$S'_i = \{t | t \in S_i \wedge t < D_i\} \cup \{D_i\} \tag{4.7}$$

Úloha τ_i pak splní své časové požadavky, pokud existuje bod plánování $S_{ij} \in S_i$, pro který platí následující vztah:

$$\sum_{1 \leq r \leq q} \alpha_r C_r \lceil \frac{S_{ij}}{T_r} \rceil + \alpha_{ij} \sum_{q < p \leq i} C_p \lceil \frac{S_{ij}}{T_p} \rceil = S_{ij} \tag{4.8}$$

Všimněme si, že úlohy, kterým již byl přiřazen faktor rozpínání, používají α_r, zatímco pro zbylé úlohy je použit faktor rozpínáni α_{ij}, který je závislý na daném bodu plánování. Z energetického hlediska je pro úlohu τ_i dobré mít co největší svůj faktor α_{ij}. To obecně platí pro všechny úlohy. Ve skutečnosti existuje úloha s indexem m taková, že její největší faktor rozpínání je nejmenší mezi všemi faktory rozpínání všech ostatních úloh:

$$\exists m, q < m \leq N, takovemax_{q<j\leq m}(\alpha_{mj}) = min_{q<i\leq n}(max_{q<j\leq i}(\alpha_{ij})) \tag{4.9}$$

Toto tvrzení nemusí nutně platit až do poslední úlohy. Pokud je $q = 0$, úloha nastaví minimální rychlost (frekvenci). Nalezneme-li index m, mohou všechny úlohy s indexem mezi q a m být rovnoměrně nataženy podle rozpínacího faktoru α_m. Rozpínací faktor je přiřazován podle rovnice 4.10.

$$\alpha_r = max_{q<j\leq m}(\alpha_{mj}), r = q + 1 \ldots m \tag{4.10}$$

Tímto krokem je dokončena jedna iterace tohoto algoritmu. Další iterace pokračuje pro $q = m$. Celý proces končí, dosáhne-li q hodnoty N, což znamená, že byly již přiřazeny faktory rozpínání všem úlohám. Maximální požadovaná rychlost pro běh každé z úloh je pak dána vztahem:

$$s_i = \frac{1}{\alpha_i}, i = 1 \ldots N \tag{4.11}$$

4.5 Plánování za běhu (dynamické)

Mechanismy popsané výše neumožňují reagovat na volný čas, vzniklý proměnou délkou vykonávání úlohy. Abychom mohli využít i tyto volné časy, musíme použít plánovací techniky, které mohou za běhu systému upravovat plánování a změny frekvence, resp. napětí procesoru. Nicméně tyto dynamické mechanismy často využívá statické metody, poněvadž mnoho vlastností systému je známých již při návrhu systému. Z tohoto důvodu dynamické DVS metody obvykle provádí jen malé změny v již dříve vytvořeném plánu.

Jak již bylo zmíněno, změny v plánu probíhají při plném chodu systému. Aby byly tyto změny efektivní, je důležité správné fungování dvou základních mechanismů. První se nazývá nalezení a odhad nečinnosti procesoru a využívá se k určení dostupného volného času, který je možné využít pro zpomalení výpočtu úloh. Druhý mechanismus, distribuce volného času k dalším úlohám, určuje, jakou rychlostí bude úloha vykonána.

Změny rychlostí s sebou nesou i nějaké režijní náklady. Z tohoto důvodu je důležité, aby u systémů, ve kterých je mnoho úloh s proměnou délkou vykonávání, byla rozhodnutí o změnách v plánu rychlá a "levná". Pokud by tomu tak nebylo, mohlo by se stát, že prostředky vynaložené na plánování by převýšily získané úspory. Rozhodnutí tedy musí být málo nákladné, rychlé (tvořené obvykle pomocí heuristik) a nesmí negativně ovlivnit chování systému.

4.5.1 Šetření cyklů

Algoritmy využívající šetření cyklů[5] ccRM a ccEDF [24] jsou založené na základních algoritmech RM a EDF. Na rozdíl od statickým berou tyto mechanismy v úvahu, že úlohy mohou být dokončeny dříve, než je nejhorší doba C. Snahou algoritmů je, aby byly tyto časy použity pro další úlohy.

ccEDF

Pro plánování pomocí EDF můžeme použít test plánovatelnosti, uvedený v 4.4, tedy vytížení U musí být menší než maximální frekvence zmenšená faktorem. Skončí-li úloha dříve, můžeme využít získaný čas a přepočítat U. Snížené U je použito, dokud není úloha znovu vyvolána v další periodě. Každá úloha může skončit dříve, avšak systém to neví nikdy dopředu. Z tohoto důvodu může být U přepočten s aktuálním časem v každém bodu plánování (spuštění či dokončení úlohy).

Samotný algoritmus 4.2 je celkem jednoduchý a pracuje následovně. Předpokládáme, že úloha τ_i skončí svůj vypočet po vykonání cc_i cyklů a cc_i je menší než C_i. Protože úloha nepoužije v aktuálním spuštění více než cc_i cyklů, budeme cc_i považovat za nejhorší dobu výpočtu. Se sníženým vytížením U můžeme nyní nalézt potenciálně menší faktor navýšení α pro zbývající plánované úlohy. Samozřejmostí je, že byla-li množina úloh plánovatelná před změnou frekvence, bude i nadále. Při opětovném spuštění úlohy musíme zajistit, aby k výpočtu vytížení bylo opět použito C_i a nikoliv cc_i.

```
vyber_frekvenci():
   použij nejnižší frekvenci  f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
   takovou, že U_1 + ··· + U_n ≤ f_i/f_m

při spuštění_úlohy(τ_i):
   nastav U_i na C_i/T_i;
   vyber_frekvenci();

při dokončení_úlohy(τ_i):
   nastav U_i na cc_i/T_i;
   /* cc_i je aktuální počet cyklů použitý při tomto vykonání úlohy */
   vyber_frekvenci();
```

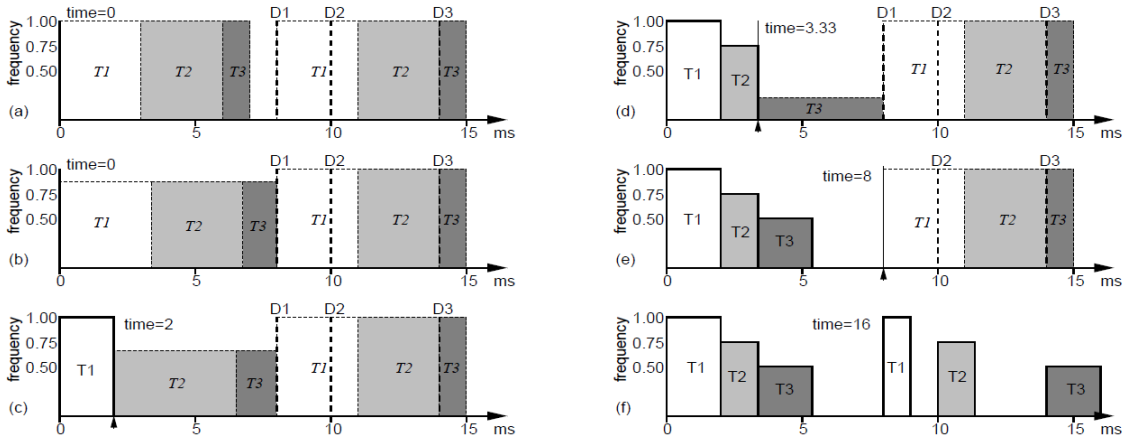
Tabulka 4.2: Algoritmus ccEDF

ccRM

Pro ccRM by mohl být také použit test plánovatelnosti pro RM, ale tento test je značně komplikovanější než pro EDF. Z tohoto důvodu použijeme jiný přístup. Pozorováním byl zjištěn následující fakt. Za předpokladu, že úlohy budou vždy vykonány v nejhorším čase C, statické RM 4.4 může vždy zaručit dodržení časových mezí. Je tedy možné tvrdit, že každá úloha τ_i bude vždy vykonána dříve, anebo nejpozději v nejhorším čase C_i. Časové meze D_i budou pak splněny bez ohlednu na to, jaká bude aktuální frekvence. Snažíme se

[5] cycle-conserving

též vyhnout překročení doby výpočtu nad nejhorší případ doby běhu tak, že každé snížení výpočetních cyklů využitých úlohami může byt použito ke snížení frekvence, resp. napětí. Princip ccRM mechanismu bude vysvětlen pomocí obrázku 4.2. Z počátku začínáme s nej-



Obrázek 4.2: Příklad algoritmu ccRM pro úlohy $\tau_1(0,3,8,8), \tau_2(0,3,10,10), \tau_3(0,1,14,14)$

horším plánem založeným na statickém RM (a), které v tomto případě používá maximální frekvenci. Pro jednoduchost se nedíváme za nejbližší časovou mez systému. Pak se snažíme rozprostřít úlohy, které mají být vykonány, přes celý interval aktuální časové meze (b). Toto poskytne nejnižší možnou hodnotu frekvence. Jelikož frekvencí procesoru je omezený počet, zaokrouhlením na nejbližší možnou dostaneme opět frekvenci = 1. Po vykonání první úlohy opakujeme rozprostírání zbývajících úloh k nejbližší časové mezi (c), což má za následek snížení frekvence, protože úloha τ_1 byla dokončena dříve. Opakování postupu pro každý bod plánování vyústí v konečný plán (f).

Ačkoliv se postup zdá celkem jednoduchý, algoritmus 4.3 je poněkud složitější kvůli několika čítačům, které se musí udržovat. Algoritmus musí sledovat zbylé cykly pro výpočet c_left_i pro každou úlohu τ_i. Když je úloha τ_i spuštěna je c_left_i nastavena na C_i. Pak určíme postup, který by udělal statický RM mechanismus v nejhorším případě pro nejbližší časovou mez libovolné úlohy v sytému. Získáme s_j udávající počet cyklů k další časové mezi a s_m udávající maximální frekvenci. Cykly s_j jsou přiřazovány úlohám shodně s přednostním pořadím RM. S každou úlohou τ_i přijímáme příspěvek $d_i \leq c\_left_i$ odpovídající počtu cyklů, které mohou být vykonány nad interval daný statickým RM. Pokud vykonáme nejméně d_i cyklů u každé úlohy τ_i (nebo pokud je úloha dokončena) pro časovou mez příští úlohy, udržíme krok s nejhorším možným případem a můžeme nastavit rychlost provádění použitím součtu d hodnot. Jak jsou úlohy vykonávány, jejich c_left a d jsou snižovány. Když je úloha τ_i dokončena, c_left_i a d_i jsou nastaveny na hodnotu 0 a frekvence je změněna. Protože jsou použita přechodná kritéria k výběru frekvence, zaručuje algoritmus splnění časových mezí všech úloh. Nevýhodou algoritmu mohou být režijní náklady způsobené častými změnami frekvence (při spuštění a dokončení úlohy), přesto je možné dosáhnout významných úspor energie bez negativních ovlivnění systému.

```
předpokládáme, že f_i je frekvence nastavená statickým algoritmem

vyber_frekvenci():
   nastav s_m = maximum_cyklů_do_další_časové_meze();
   použij nejnižší frekvenci f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
   takovou, že (d_1 + ··· + d_n)/s_m ≤ f_i/f_m

při spuštění_úlohy(τ_i):
   nastav c_left_i = C_i;
   nastav s_m = maximum_cyklů_do_další_časové_meze();
   nastav s_j = s_m * f_i/f_m;
   přiděl_cykly(s_j);
   vyber_frekvenci();

při dokončení_úlohy(τ_i):
   nastav c_left_i = 0;
   nastav d_i = 0;
   vyber_frekvenci();

během vykonávání_úlohy(τ_i):
   sniž c_left_i a d_i;

přiděl_cykly(k):
   for i = 1 to n, τ_i ∈ {τ_1, ..., τ_n | T_1, ..., T_n}
                    /*úlohy seřazené dle periody*/
       if (c_left_i < k )
             nastav d_i = c_left_i;
             nastav k = k − c_left_i;
       else
             nastav d_i = k;
             nastav k = 0;
```

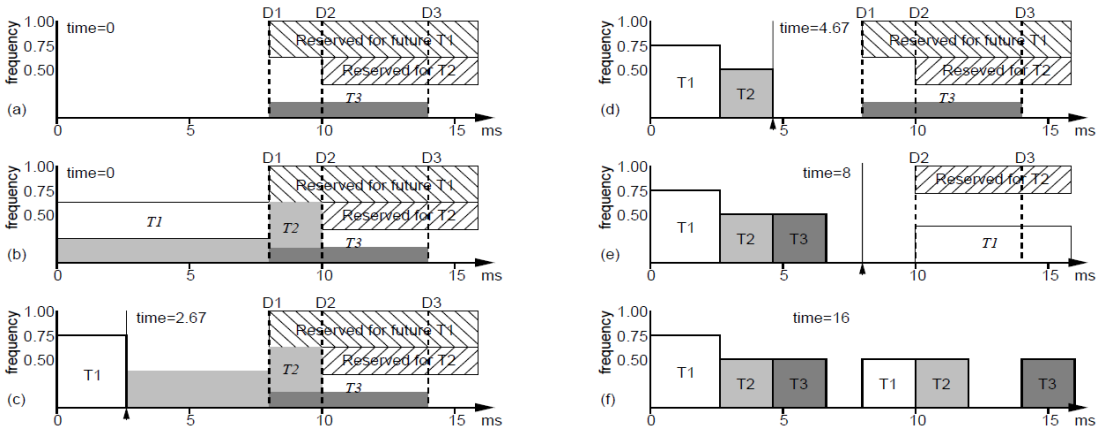
Tabulka 4.3: Algoritmus ccRM

4.5.2 laEDF

Snahou dopředného EDF[6] (dále laEDF)[24] je dosáhnout větších úspor energie použitím dopředné techniky. Cílem je určit, kolik výpočtů bude v budoucnu potřeba a pozdržet vykonání úloh jak jen to bude možné. Dále je snahou algoritmu nastavit frekvenci na nejnižší možnou, při které budou splněny časové meze úloh, aniž by byly ohroženy časové meze úloh budoucích. Toto chování může způsobit spouštění úloh s nižší prioritou při maximální frekvenci, aby byly zaručeny jejich časové meze. Nicméně mají-li úlohy sklony k tomu, že jsou často vykonány dříve než je jejich nejhorší doba C, nebude pro vykonání pozdržených úloh potřeba maximální frekvence. Použitím této metody můžeme tedy dosáhnout práce při nízkých frekvencích, při uspokojení časových mezí všech úloh.

Princip algoritmu opět vysvětlíme pomocí obrázku 4.3. Jak již bylo zmíněno, cílem je pozdržet vykonání úloh až k nejbližší časové mezi v systému (D_1) tak, aby bylo možné pracovat na nejnižší frekvenci. Každé úloze v plánu přidělíme nejhorší čas potřebný k provedení

[6] look-Ahead EDF



Obrázek 4.3: Příklad algoritmu laEDF pro úlohy $\tau_1(0, 3, 8, 8), \tau_2(0, 3, 10, 10), \tau_3(0, 1, 14, 14)$

úlohy C, v pořadí od úlohy s nejdelší časovou mezí (na obrázku D_3). Pak rozprostřeme vykonávání úlohy τ_3 mezi časové meze D_1 a D_3, s ohledem na omezení rezervovaných kapacit v budoucnu vyvolaných ostatních úloh(a). Tento krok opakujeme pro úlohu τ_2. Tato úloha nemůže byt dokončena v čase mezi D_1 a D_2, který zbyl po alokaci času pro τ_3 a budoucí vyvolání τ_1. Nějaký čas pro vykonání τ_2 a celé úlohy τ_1, je tedy vymezen před D_1(b). Práci přidělenou před D_1 použijeme k určením pracovní frekvence. Jakmile je úloha τ_1 dokončena v kratším čase než C_1, opakujme celý postup k nalezení nižší frekvence. Níže je popsán celý algoritmus laEDF 4.4.

4.5.3 Nízko energetického prioritního plánování

Základním principem nízko energetického prioritního plánování(dále lpps[7]) je, že si udržuje dvě fronty úloh [25]: čekací a připravené. Připravená fronta obsahuje úlohy, které jsou připraveny k běhu, seřazené podle jejich priority. Úloze prováděné procesorem říkáme aktivní. Čekací fronta obsahuje úlohy, které již běžely a nyní čekají na další periodu jejich spuštění. Zde jsou úlohy řazeny dle času, kdy bude úloha opět spuštěna. Jeli aktuální čas stejný s časem spuštění některé úlohy, je úloha přesunuta do fronty připravených úloh. Plánovač následně zkontroluje, zda-li nemá úloha ve frontě připravených úloh vyšší prioritu než aktivní úloha. Plánovač lpps může být implementován nepatrnou modifikací tradičního plánovače, protože je závislý pouze na informacích dostupných skrz fronty.

Dále je potřeba řídit změny frekvence provádění. Při spuštění je frekvence nastavena na maximu. Je-li potřeba provést více jak dvě úlohy, pracuje se stále na maximální frekvenci. Pokud zbývá jíž jen jedna úloha, tzn. ta aktivní, použije se algoritmus MRS pro určení rychlosti vykonávání úlohy, tak aby byl využita doba mezi aktuálním časem a časem, kdy bude přesunuta úloha z čekací fronty. Nenachází-li se žádná úloha ve frontě čekajících úloh, je procesor přepnut do režimu nízkého výkonu. Dále je nastaven časovač, který probudí procesor před příchodem další úlohy. Nastavení času vzbuzení by mělo brát v úvahu dobu potřebnou k přechodu procesoru na vyšší výkon.

Dle způsobu určování priorit je možné tuto metodu použit pro standardní plánovací mechanismy EDF či RM, případně se mohou vyskytovat i další modifikace běžných algo-

[7]Low-Power Priority-Based Real-Time Scheduling

```
vyber_frekvenci(x):
   použij nejnižší frekvenci f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
   takovou, že x ≤ f_i/f_m

při spuštění_úlohy(τ_i):
   nastav c_left_i = C_i;
   odlož();

při dokončení_úlohy(τ_i):
   nastav c_left_i = 0;
   odlož();

během vykonávání_úlohy(τ_i):
   sniž c_left_i;

odlož():
   nastav U = C_1/T1 + ··· + C_n/Tn;
   nastav s = 0 ;
   for i = 1 to n, τ_i ∈ {τ_1, ..., τ_n | D_1 ≥ ··· ≥ T_n}
                  /*poznámka: obrácené pořadí úloh jak u EDF*/
       nastav U = U − C_i/Ti;
       nastav x = max(0, c_left_i − (1 − U)(D_i − D_n));
       nastav U = U + (c_left_i − x)/(D_i − D_n);
       nastav s = s + x;
   vyber_frekvenci(s/(D_n − aktuln_as));
```

Tabulka 4.4: Algoritmus laEDF

ritmů.

4.5.4 DRA

Algoritmus DRA[8] porovnává aktuální historii vykonávání s plánem P_{stat} vytvořeným pomocí statické metody [1], např MRS. Spuštěním úlohy dříve než očekával plán P_{stat}, získáme čas, který je využit k bezpečnému zpomalení vykonávání další úlohy. Jednoduchost tohoto návrhu s využitím předstihu úloh před P_{stat} a slepé snižování frekvence procesoru, bohužel není ve všech ohledech bezpečné.

K tomu aby byla i nadále zaručena proveditelnost plánu a účinnost algoritmu, je vytvořena tzv. α-fronta, která napomáhá vypočítat předstih úloh, když jsou vykonány. Fronta obsahuje informace o všech úlohách, které mohou běžet. Tyto informace získává od statického plánu P_{stat}.

Statický plán je tvořen pomocí metody EDF*, která je stejná jako EDF s tím rozdílem, že úlohám se stejnou periodou jsou priority přiřazeny dle metody FIFO. V α-frontě jsou pak úlohy také řazeny tímto způsobem.

Vykonáním úlohy dříve než se čekalo, zůstane nevyužitý čas rem. Jelikož dokončená úloha má ve frontě stále nejvyšší prioritu, musí být odstraněna. Pak může být naplánována další úloha v pořadí. Tato úloha může použít nevyužitý čas rem, celý či část, k tomu,

[8] Dynamic Reclaiming Algorithm

aby zpomalila rychlost vykonávání. K nastavení počáteční rychlosti vykonávání všech úloh je použit alg. MRS. Pro změnu rychlosti, je již pak použit jednoduchý alg., který upraví rychlost dle požadavků časových mezí.

Aggressive-DRA

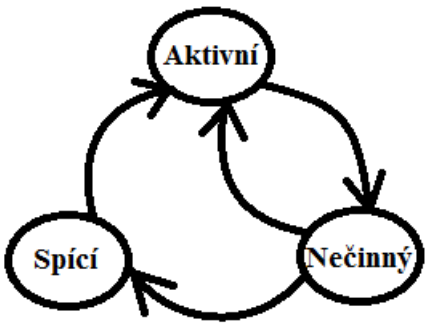
Tento algoritmus vychází z DRA, tedy používá jak frontu tak statické EDF*. Velkým rozdílem je předpoklad, že aktuální i budoucí úlohy boudou pravděpodobně potřebovat menší čas k vykonání než je nejhorší případ. Z tohoto důvodu může za jistých podmínek přijmout agresivní přístup založený na snižování rychlosti běžících úloh. Pokud by však nastal pesimistický scénář, měl by být algoritmus připraven zvýšit rychlost, tak aby bylo zajištěno splnění časových mezí všech úloh.

Důležité je, aby agresivní změny rychlosti nenarušily proveditelnost plánu. Z tohoto důvodu je třeba stanovit úroveň agresivity, tzn. jak mnoho je možné snížit rychlost. Toho je docíleno určením hranic snižování rychlosti.

Změny rychlostí mohou probíhat následovně. Máme-li n úloh připravených k běhu a všechny úlohy musí být dokončeny před příchodem úlohy $n + 1$, pak může být vykonávání např. úlohy 1 zpomaleno, zatímco vykonání ostatních úloh bude zrychleno. Navíc mohou být ještě použity nevyužité časy. Toto chování však nesmí ohrozit proveditelnost plánu. Detailní popis lagoritmů DRA a AGR včetně pseudokódu je možné nalézt v [1].

4.6 DPM

Problém snižovaní spotřeby energie v systému se DPM[9] metody snaží řešit přechodem do nízkoenergetických stavů při nečinnosti. Těchto stavů může být obecně mnoho a každý stav je charakterizován příkonem a výkonem. Takovýto systém můžeme pak modelovat jako energetický konečný automat. Na obrázku 4.4 jsou znázorněný tři stavy - aktivní, nečinný (IDLE) a spící (sleep). Často má zařízení nejen více aktivních stavů, ale i stavů nečinnosti a spánku. Např. procesor Intel Core i7 má šest aktivních stavů, čtyři nečinné a tři, resp. pět stavů spánku (včetně stavu při vypnutém zařízení a při aktivním provozu).



Obrázek 4.4: Ilustrace konečného automatu DPM

Přechod z jednoho stavu do druhého stojí nějaký čas a energii. Obvykle nízký příkon

[9] Dynamic Power Management

znamená nízký výkon a delší dobu změny (přechodu). Přechod z nečinného stavu do aktivního je tedy rychlejší než ze spánku. Pokud je komponenta umístěna do nízkoenergetického stavu, např. spánku, je komponenta nedostupná po dobu strávenou v tomto stavu a navíc po dobu potřebnou k přepnutí mezi stavy. Změny mezi stavy jsou řízeny správcem napájení (dále PM[10]), který sleduje vytížení systému a rozhoduje, kdy k přechodům dojde. PM dělá rozhodnutí podle zvolené politiky.

Práce PM se skládá z následujících úloh:

- sledování a modelování příchozích žádostí s cílem předpovídat výskyt delších nečinných stavů(IDLE),
- tvarování provozu, tzn. vyrovnávání provozu tak, aby byly vytvořeny větší IDLE stavy, které umožní uspat zařízení na delší dobu,
- rozhodnutí kdy a do kterého stavu spánku bude zařízení přepnuto,
- rozhodnutí kdy by mělo být zařízení opět probuzeno.

Dále si blíže popíšme DPM politiky, které se k těmto úkolům staví různými způsoby.

4.6.1 Heuristické politiky

Většina dnes používaných metod se zaměřuje jen na rozhodnutí, kdy by měl být systém uspán. Cena přechodu mezi aktivním stavem a spánkem je obvykle kritériem toho, jak moc agresivní přístup bude použit. Pokud je cena přepnutí nepatrná či ještě v únosných mezích, používá se v komerčních implementacích metoda přepnutí do spánku hned, jakmile je to možné.

U systému, kde je již cena přechodu vyšší a okamžité uspání při nečinnosti by narušovalo plynulé používaní, je často použita *politika časového limitu*. Při této metodě se zařízení přepne do spánku, až vyprší časový limit, který udává jak dlouho má systém čekat v nečinnosti, než bude uspán. Časový limit muže být pevně daný anebo nastavitelný uživatelem. Příklad může být nastavení doby nečinnosti, za kterou bude vypnuta obrazovka notebooku. Při tomto přístupu je důležité správně zvolit časový limit, jelikož příliš krátký může způsobit velmi časté uspání na krátkou dobu, což může být dražší než zůstat v nečinném stavu. Např. nastavení nízkého limitu pro pevný disk může znatelně ovlivnit výkon systému. Na druhou stranu dlouhý limit může zbytečně spotřebovávat energii v nečinnosti a následně přejít ke spánku těsně před příchodem nových úloh, což by se také prodražilo.

Nevýhody pevně stanoveného časového limitu se snaží vylepšit nové mechanismy, které upravují délku limitu podle zatížení systému. Může k tomu být využito např. *strojové učení spolu s modely založenými na ekonomické analýze*.

Další možností je využít *prediktivní politiky*, která se snaží nejen předpovídat délku další nečinnosti studováním distribuce příchodů požadavků, ale snaží se také s jistou přesností přepínat do spánku s co nejmenší dobou nečinnosti. Navíc některé prediktivní politiky vzbudí zařízení, očekávají-li příchod požadavku na obsluhu. Pokud jsou tyto predikce správné, poskytuje tato politika řešení bez zbytečné režie. Avšak je-li predikce špatná, možné náklady mohou být obrovské. Kvalita předpovědi doby nečinnosti je podstatou těchto politik.

Jak politiky s časovým limitem, tak ani prediktivní politiky nemohou zaručit optimální řešení. Toto nám mohou poskytnout až stochastické metody, avšak jen v rámci omezených předpokladů stanovených již při vyvíjení systému.

[10] power manager

4.6.2 Stochastické politiky

Stochastické politiky mohou být rozděleny na řízené událostmi anebo časem. Jsou založené na předpokladu, že distribuční modely jsou nezávislé na historii (někdy jsou i závislé). Dále je možné je ještě rozdělit na neměnné, tzn. vždy je použita stejná politika a měnné, tzn. politika se časem mění. Ať se jedná o kterýkoliv způsob, může být *optimálnost* zaručena, jsou-li algoritmy založeny na modelu Markovových řetězcích např. algoritmy discrete-timeMarkov decision processes (DTMDP), continuous-time Markov decision process (CTMDP) a time-indexed semi-Markov decision process (TISMDP)

4.6.3 DPM v operačních systémech

Politiky řízení spotřeby jsou většinou implementovány jako součást OS. Jako nejznámější uveďme ACPI[11], které je použito v OS windows. Nicméně tato metoda neprovádí přímo politiky, ale poskytuje jen prostředky pro DPM prováděné na vyšších úrovních systému. Tyto metody avšak neprovádí žádné tvarování provozu. Jejich hlavním cílem je jen zhodnotit, zda v době nečinnosti přepnout do spánku či nikoliv. Záleží pak na implementaci OS jak s těmito informacemi naloží.

[11] Advanced Configuration and Power Interface

Kapitola 5

Plánovací mechanismy při přetížení

Tato kapitola se zabývá problémem plánování RT úloh při přetížení systému, tzn. v kritických situacích, ve kterých je požadovaný výpočetní výkon množinou úloh vyšší něž kapacita procesoru. Z tohoto důvodu nemohou být všechny úlohy dokončeny v rámci jejich časových mezí. Okolnosti vedoucí k přetížení se mohou vyskytnout z různých příčin, uveďme některé z nich:

- Špatný návrh systému - systém je navržen a analyzován na základě pesimistických předpokladů. Takový systém může dobře fungovat za normálních okolností, avšak při špičkovém zatížení může dojít ke zhroucení systému.
- Souběžný příchod událostí - i když je systém řádně navržen, může při souběžném příchodu několika neočekávaných událostí dojít ke zvýšení zatížení procesoru přes tolerovaný práh.
- Selhání vstupních zařízení - chyby např. v senzorech mohou někdy generovat neobvyklé sekvence přerušení, které zasytí procesor či zdrží vykonávání úloh, což vede k překročení jejich časových mezí.
- Nepředvídatelné střídání podmínek v okolním prostředí může vyvolat výpočetní požadavek na procesor vyšší, než může procesor zvládnout.
- Výjimky OS - neobvyklá kombinace dat může v některých případech vyvolat výjimku jádra, která spustí rutinu jádra s vysokou prioritou, což může zapříčinit zpoždění výpočtu úlohy.

5.1 Vytížení

V kapitole 3.2 byly zmíněny vztahy pro výpočet zatížení resp. vytížení procesoru pro množinu n periodických úloh. Je-li $U > 1$ nemůže být množina úloh plánovatelná žádným algoritmem. Pro obecnou množinu úloh, které mohou dynamicky přicházet se zatížení systému mění při každém příchodu požadavku na spuštění úlohy. Tomuto požadavku je potřeba přizpůsobit i vztah pro vytížení U. Praktický se ukázal vztah pro výpočet *okamžitého zatížení* $U(t)$ navržený autory Buttazzem a Stankovicem.

Podle této metody je vytížení vypočítané ve všech intervalech aktuálního času t a každé časové meze aktivované úlohy d_i. Všechny intervaly potřebné k výpočtu jsou dány vztahem

$[t, d_1], [t, d_2], \ldots, [t, d_n]$. Pro každý interval $[t, d_i]$ vypočteme částečné vytížení $U_i(t)$ pro první úlohu i následovně:

$$U_i(t) = \rho_i(t) = \frac{\sum_{d_k \leq d_i} c_k(t)}{(d_i - t)}, \tag{5.1}$$

kde $c_k(t)$ značí zbývající čas potřebný k dokončení úlohy τ_k s časovou mezí menší nebo rovnou d_i. Celkové vytížení v čase t je pak následující:

$$\rho(t) = max_i \rho_i(t). \tag{5.2}$$

5.2 Přetížení a překročení

Mluvíme-li o výpočetním zatížení procesoru, musíme rozlišovat mezi pojmy *přetížení* a *překročení*.

O přetíženém systému hovoříme, pokud výpočetní čas požadovaný množinou úloh v určitém časovém intervalu je větší než dostupný výpočetní čas procesoru ve stejném intervalu.

Úloha překročila, pokud přesáhla své očekávané vytížení. Překročení mohou nastat, buď při příchodu požadavku na spuštění další úlohy dříve než bylo předpokládáno (aktivační překročení), nebo překročením očekávané doby výpočtu úlohy (výpočetní překročení).

Jak je vidět, tak přetížení je stav související se procesorem a překročení souvisí s instancí jedné úlohy. Překročení úlohy nemusí bezpodmínečně vyústit v přetížení systému. Nicméně velké neočekávané překročení úloh či řada takovýchto překročení může zapříčinit nepředvídatelné účinky na systém, pokud nebudou vhodným způsobem řízeny. Přetížení můžeme rozlišovat na přechodné a permanentní.

- Přechodné přetížení znamená, že přetížení nastane jen na omezenou dobu. Průměrné zatížení systému je tedy menší nebo rovno jedné, ale maximální zatížení je větší jak jedna.
- Permanentní přetížení znamená, že přetížení nastane na nepředvídatelně dlouhou dobu. Průměrné zatížení systému je větší než jedna.

V RT systémech může být přechodné přetížení způsobené řadou překročení, anebo nárazovým příchodem aperiodických požadavků. Permanentní přetížení typicky nastává v systémech s periodickým příchodem požadavku, pokud celkové zatížení procesoru překročí hodnotu jedna.

5.3 Permanentní přetížení periodického systému

Tento typ přetížení nastane, je-li celkové vytížení systému $U > 1$. Nadměrné zatížení může nastat z následujících příčin:

- špatné odhadnutí maximální požadované doby potřebné k vykonání úlohy,
- neočekávaná aktivace nových periodických úloh,
- zvýšení rychlosti aktivace úlohy z důvodu reakce na nějakou změnu v okolním prostředí.

Tyto situace vedou k hromadění výpočetních činnosti do systémových front. Fronty se mohou zvětšovat pokud přetížení přetrvá čas odezvy úloh může růst až k neurčitu. K obsluze těchto situací vyskytujících se v periodických systémech mohou být použity v podstatě tyto tři metody jak snížit vytížení:

- Přeskakování úloh[1] se snaží snížit celkové vytížení systému vhodným přeskakováním vykonání některých periodických úloh takovým způsobem, že každá úloha má zaručený minimální počet výpočtů úlohy v rámci jejich časových omezení.
- Přizpůsobení periody[2] se snaží snížit vytížení rozšiřováním period úloh na vhodné hodnoty tak, že celkové pracovní zatížení zůstane pod požadovaným prahem.
- Přizpůsobení obsluhy[3] se snaží snížit vytížení snižováním výpočetních požadavků úloh využitím předvídatelnosti s kvalitou obsluhy. Tato metoda lze použít jen u úloh, které byly navrženy tak ,že mohou rozhodovat mezi výkonem a výpočetními nároky. Např. rozhodnutí jak přesné budou vypočtené výsledky. Snížením přesnosti je pak možné zkrátit dobu výpočtu.

5.4 Přechodné přetížení způsobené řadou překročení

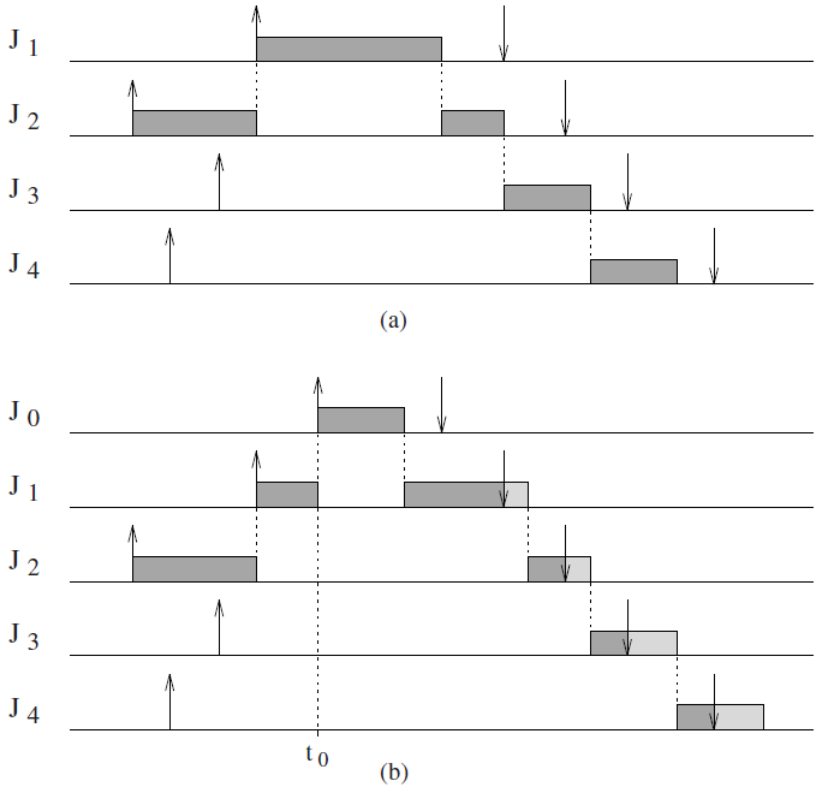
Nyní se zaměřme na obsluhu přechodných přetížení způsobená překročením úlohou, která musela být vykonávána déle, či byla-li aktivována častěji, než se očekávalo. Tato situace může nastat, pokud některý parametr úlohy byl špatně nastaven, anebo byl-li systém záměrně navržen pro příliš optimistické předpoklady za účelem dosažení vyššího průměrného vytížení.

Špatné zacházení s překročením může způsobit vážné problémy RT systému, ohrozit vykonávání kritických úloha a způsobit náhlé snížení výkonu. Aby se zabránilo překročení, které představuje časové ztráty vzniklé vykonáním úloh, musí systém rozhodnout, jestli přeruší vykonávání aktuální úlohy zapříčiňující překročení, anebo ji nechá pokračovat s nižší prioritou. První způsob není bezpečný, jelikož ukončená úloha se mohla nacházet v kritické sekci a ve sdílených zdrojích zanechat nekonzistentní data. Druhý způsob je mnohem lepší, jelikož míra ovlivnění ostatních úloh překročením, může být řízena přiřazováním priorit chybným úlohám, aby provedly zbylý výpočet. Obecný přístup pro toto řešení je implementován jako rezervace zdrojů[4].

5.5 Přechodné přetížení způsobené aperiodickými úlohami

Tento typ přetížení je typický pro událostmi řízené systémy skládající se z mnoha aperiodických úloh vyvolaných externími událostmi. Není-li sytém navržen, aby zvládal enormní příchody událostí, mohou efekty přetížení vyústit ve vážné problémy s řízením sytému. Experimenty bylo zjištěno, že např. výkon EDF může rapidně klesnout během přetížení. V případě, ve kterém příchod nové úlohy může zapříčinit, že všechny předcházející úlohy zmeškají své časové meze. Tento jev nazývaný *domino efekt*, který je znázorněn na obr.5.1 a znázorňuje plánování úloh dle EDF normálně(a) a po příchodu aperiodické úlohy (b).

[1] job skipping
[2] period adaptation
[3] service adaptation
[4] resource reservation



Obrázek 5.1: Domino efekt při plánování pomocí EDF

5.6 Zvládání přetížení způsobené aperiodickými úlohami

Máme množinu aperiodických úloh, kde každá úloha má základní parametry $\tau(r, C, D)$. V jistý okamžik přijde více požadavku na spuštění úloh, čímž může vzniknout přetížení. Pro takovéto dynamické systémy neexistuju žádný algoritmus, který může zaručit proveditelný plán pro množinu takovýchto úloh. Jelikož jedna či více úlohy mohou zmeškat své časové meze, je vhodné, aby se zpozdily úlohy, které jsou *méně důležité*. Tento výběr povede pouze k omezenému provozu a nikoliv k selhání systému. Bohužel základní parametry v sobě nenesou žádnou informaco o tom, do jaké míry je úloha důležitá pro uspokojivý chod systému.

Pro příklad mějme dvě úlohy. První úloha pouze jednou za vteřinu obnoví zobrazení hodin na obrazovce. Druhá úloha kontroluje každou vteřinu teplotu v reaktoru. Je jasné že zpoždění či dokonce přeskočení obnovení zobrazení hodin nebude mít nijak velký vliv na systém. Na druhou stranu zpoždění měření teploty může vyústit až v tragické následky.

Aby plánovací algoritmus neudělal chybu je tedy nutné přiřadit úlohám nějakou *hodnotu*, která bude odpovídat jejich důležitosti v sytému. Tato hodnota přiřazená úloze musí odrážet její význam[5] s ohledem na ostatní úlohy v množině. Jakým způsobem je hodnota přiřazená, je závislé na konkrétní aplikaci. Například může být hodnota stanovena podle:

- doby potřebné pro výpočet C;

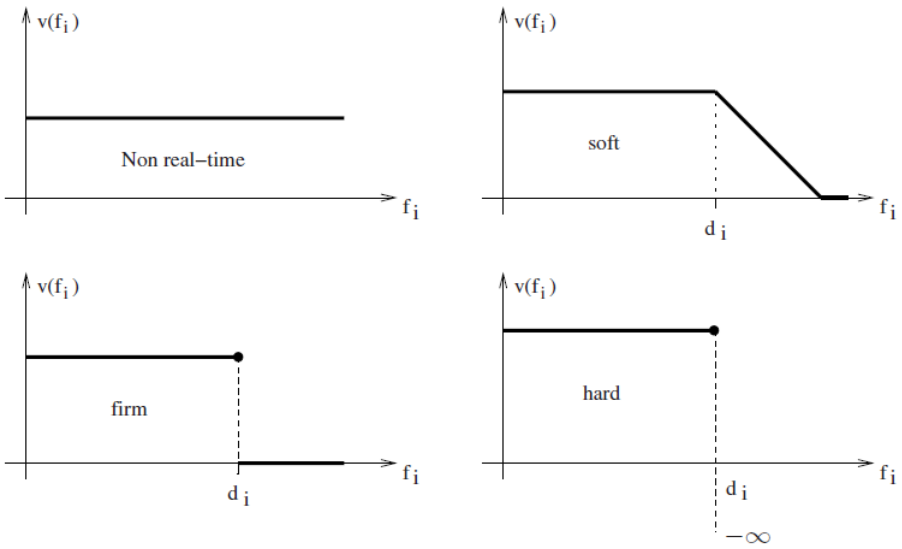
[5] tzn. jaký užitek či zisk pro systém bude mít vykonání úlohy

- libovolného celého čísla, které odráží důležitost úlohy;
- poměru mezi celým číslem a dobou potřebnou k výpočtu, atp.

Tomuto poměru se říká *hodnotová hustota*.

5.6.1 Užitková funkce

V RT systémech je aktuální hodnota úlohy závislá na čase, z tohoto důvodu lze důležitost úlohy lépe popsat pomocí *užitkové funkce*[6], pro základní čtyři typy úloh včetně ne real-time úloh. Ne RT úlohy, které nemají časová omezení, mají stálý nízký užtek, vždy když jsou dokončeny. Hard úlohy mají užitek pro systém, jsou-li dokončeny v rámci časových mezi. Pokud není úloha dokončena včas, může být užitek pro systém považován jako mínus nekonečno. Soft úlohy přináší užitek pro sytém, i když nejsou dokončeny včas, tento užitek se s rostoucím časem snižuje. V neposlední řadě firm úlohy sice při zmeškání časových mezí neohrožují systém, avšak pak nepřináší užitek. Hodnoty užitkových funkcí v čase jsou znázorněny v obr 5.2.



Obrázek 5.2: Užitkové funkce základních typů úloh

K měření výkonnosti plánovacích algoritmů muže být použit součet hodnot všech úlohových funkcí užitečnosti, které jsou vypočteny po dokončení každé úlohy. Definujme agregující zisk (kumulativní hodnotu) Γ_A pro plánovacího algoritmu A následovně:

$$\Gamma_A = \sum_{i=1}^{n} v(f_i), \tag{5.3}$$

kde A je optimální algoritmus, právě když Γ_A je maximální.

[6] utility function - funkce zisku

5.6.2 Model RT úloh pro přetížení

Pro plánování úloh při přetížení musíme upravit model úloh uvedený v 3.2. Dále tedy uvažujme množinu úloh $T = \tau_i = (C_i, D_i, V_i), i = 1 \dots n$, kde C_i a D_i je stejné a parametr V_i je hodnota získaná pokud úloha τ_i skončí před D_i. Maximální *agregující zisk* dosažitelný na množině úloh je roven součtu všech hodnot V_i takto:

$$\Gamma_{max} = \sum_{i=1}^{n} V_i. \tag{5.4}$$

Jelikož při přetížení nejsou některé úlohy dokončeny, nemůže být této maximální hodnoty nikdy dosaženo. Nechť Γ^* je maximální agregující zisk, kterého je možné dosáhnout libovolným plánovacím algoritmem. Pak výkonnost algoritmu A lze měřit porovnáním hodnot Γ_A s Γ^*.

5.6.3 Optimální algoritmus

K porovnávání on-line plánovacích algoritmů potřebujeme analyzovat jejich vlastnosti a výkon při přetížení. Víme, že pro plánování za normálních podmínek existují optimální algoritmy podle kterých je možné srovnávat. Bohužel pro plánování úloh při přetížení takový algoritmus není možné nalézt, jelikož by musel umět dopředu přesně určit, kdy a v jakém pořadí budou úlohy přicházet. Plánovací algoritmus s jasnovideckými vlastnostmi lze sestrojit pouze teoreticky. Ačkoli je optimální algoritmus jen čistě teoretický, může být použitý jako referenční model, aby bylo možné vyhodnotit výkon on-line plánovacích algoritmů při přetížení.

5.6.4 Faktor soutěžení

Pro množinu úloh a plánovací algoritmus nad ním reprezentuje agregační zisk míru výkonnosti právě pro tuto množinu. Chceme-li charakterizovat nějaký algoritmus s ohledem na nejhorší možné podmínky, potřebujeme vypočítat minimální agregační zisk, kterého je možné dosáhnout na libovolné množině úloh. Pro měření nejhorší výkonnosti algoritmu A lze použít faktor soutěžení uvedený Baruahem.

Definice 5.6.1 Plánovací algoritmus A má faktor soutěžení φ_i, právě když může zaručit agregační zisk $\Gamma_i \geq \varphi_A \Gamma^*$, kde Γ^* je agregační zisk dosažený optimálním teoretickým plánovacím algoritmem.

Z definice 5.6.1 lze vypozorovat, že faktor soutěžení je reálné číslo $\varphi_A \in [0, 1]$. Má-li algoritmus A faktor soutěžení φ_A, znamená to, že A může dosáhnout agregačního zisku Γ_A nejméně φ_A krát agregační zisk teoretického optimálního algoritmu Γ^*.

Další důležitý teoretický výsledek nalezený Burahem je, že faktor soutěžení každého algoritmu je shora ohraničen. Tento teorém říká, v systémech s faktorem zatížení $\rho > 2$ a úlohami s důležitostí úměrnou jejich době běhu neexistuje on-line algoritmus A schopný garantovat faktor soutěžení $\varphi_A > 0,25$. Nicméně tento teorém vychází z velmi omezujících podmínek kladených na systém. V praxi tyto podmínky nemusí být splněny, proto má teorém spíše teoretickou platnost. Důkaz teorému je možné nalézt zde [3].

5.6.5 Klasifikace mechanismů

Podle způsobu rozhodování, předvídání a řízení přetížení, můžeme plánovací algoritmy používané při přetížení rozdělit do tří skupin následovně:

- Algoritmy s nejlepší snahou[7], které nepoužívají žádnou predikci přetížení. Příchozí nová úloha je vždy přijata a zařazena do fronty úloh připravených k běhu. Výkonnost systému může být řízena pouze vhodným přiřazováním priority tak, že přidává úlohám hodnotu.

- Algoritmy s testem přijetí, které řídí vstup nových úloh provedením testu zaručujícího plánovatelnost. Kdykoliv přijde nová úloha do systému, záruční rutina ověří plánovatelnost množiny úloha založené na předpokladech nejhorších možných případů. Pokud je nová množina úloh plánovatelná, je úloha přijata do fronty, jinak je odmítnuta.

- Robustní algoritmy oddělují časové omezení od důležitosti pomocí dvou různých přístupů. Jeden pro přijetí úlohy a druhý pro odmítnutí úlohy. Kdykoliv přijde nová úloha, test přijetí ověří plánovatelnost nové množiny při nejhorších možných podmínkách. Je-li množina úloh plánovatelná, nová úloha je zařazena do fronty. Pokud není nalezen plánovatelný rozvrh, je podle přístupu jedna či více úloh odmítnuto tak, aby bylo dosaženo maximálního agregačního zisku proveditelných úloh.

Test přijetí

Test přijetí můžeme chápat jako filtr, který řídí zatížení systému a snaží se ho držet pod hodnotou jedna. Jakmile je úloha přijata algoritmus zaručuje, že bude dokončena v rámci časových mezí. Test však nebere v úvahu důležitost úlohy, což může při přechodném přetížení vest k odmítnutí úlohy s vysokou důležitostí. Toto může za jistých okolností vést ke špatným výkonům v rámci agregačního zisku. Tento problém řeší robustní algoritmy reklamačním mechanismem. Odmítnuté úlohy jsou dočasně ponechány v čekací frontě. Dokončí-li některá z úloh svůj výpočet dříve než je nehorší doba běhu C, může být získaný volný čas použitý k naplánovaní některé z odmítnutých úloh.

5.6.6 Algoritmy s nejlepší snahou

Níže stručně popíšeme velmi často používané algoritmy s nejlepší snahou. Konkrétně se jedná o DASA[8] a LBESA[9] [21]. Oba postupy se snaží vytvořit plán, který bude poskytovat co největší celkový užitek, resp. agregační zisk. Než přejdeme k samotným principům definujme váhu užitku[10] VU jako podíl užitku úlohy τ_i a relativní časové meze $D_i(t)$.

$$VU_i = V_i/D_i(t) \tag{5.5}$$

DASA

Všechny úlohy čekající na spuštění jsou v sestupném pořadí dle VU postupně prověřovány následovně. Úloha je vložena do prozatímního plánu PP, u kterého je prověřena jeho proveditelnost. V PP jsou úlohy vzestupně řazeny dle $d_i(t)$. Pokud vložení úlohy τ_i do plánu

[7] best effort
[8] Dependent Activity Scheduling Algorithm
[9] Locke's Best Effort Scheduling Algorithm
[10] value density

PP má za následek neproveditelnost plánu, je úloha z plánu odstraněna. Jsou-li prověřeny všechny úlohy, vybere se z plánu PP pomocí EDF úloha, která bude spuštěna. Tento celý mechanismus se provádí v každém bodu plánování.

LBESA

Zatímco DASA prověřuje úlohy v sestupném pořadí dle VU, LBESA prověřuje úlohy v pořadí vzestupném. Podobně jak u DASA jsou úlohy vkládány do prozatimního plánu, který je následně zkontrolován na proveditelnost. Pokud vložení úlohy do PP způsobí neproveditelnost plánu, není úloha odstraněna z plánu jako u DASA, ale z PP jsou postupně odstraňovány úlohy s nejmenším VU, dokud není PP proveditelný. Jsou-li prověřeny všechny úlohy, je opět spuštěna úloha dle EDF.

Nevýhodou obou výše uvedených algoritmů je tvoření prozatimního plánu v každém bodu plánování. Tento postup klade s rostoucím počtem úloh vysoké režijní náklady. V [18] se např. snaží pomocí heuristiky zlepšit výkony těchto algoritmů. Výsledky byli pozitivní, avšak nikoliv moc významné. Poznamenejme, že mimo přetížení se oba algoritmy chovají jak EDF.

5.6.7 D^{over}

Jedná se o jediný algoritmus, který dosahuje faktoru soutěžení 0.25 [3, str. 312]. Pokud nedojde k přetížení chová se jako EDF. Přetížení je detekováno, pokud je zbytková relativní doba volnosti úlohy $L(t) = 0$. V tomto případě musí být úloha buď spuštěna nebo zahozena.

D^{over} rozděluje množinu úloh připravených k běhu na dvě disjunktní množiny: privilegovaných úloh a čekajících úloh. Je-li u úlohy τ_i detekováno $L(t) = 0$ bude spuštěna, pokud

$$V_i > (1 + \sqrt{k})(V_{aktual} + V_{priv}), \tag{5.6}$$

kde V_{aktual} je hodnota úlohy τ_i , V_{aktual} je hodnota právě vykonávané úlohy, V_{priv} je součet hodnot všech privilegovaných úloh a k je poměr mezi nejvyšší a nejnižší hodnotou úloh v systému.

Běží-li již úloha τ_i, která má $L_i(t) = 0$ a vyskytne se další úloha τ_j, která má také $L_j(t) = 0$, bude úloha τ_j pouze pokud bude splněna podmínka 5.7, v opačném případě bude zahozena.

$$V_j > (1 + \sqrt{k})V_i \tag{5.7}$$

5.6.8 RED

Algoritmus RED[11] [3] využívá kombinaci funkcí pro omezení provozu při přetížení, tolerance časových mezí a znovu využití zdrojů. Díky tomu poskytuje znamenitý výkon jak při normálním zatížení, tak při přetížení, navíc umožňuje předpovídat nejen zmeškání časových mezí, ale i velikost přetížení, jeho trvání a dopad na systém.

K základním charakteristikám úlohy přidává nový parametr M, který určuje toleranci časové meze. To nám říká, jak dlouho po zmeškání časových mezí ještě úloha produkuje korektní data. Součet D a M používá akceptační test, pro plánování se pak již používá pouze časová mez D. Akceptační test se snaží vytvořit proveditelný plán pro EDF. Pokud neuspěje, odmítá úlohy s nejnižší hodnotou C, dokud se nepodaří vytvořit proveditelný

[11] Robust Earliest Deadline

plán. Odmítnuté úlohy nejsou zcela zahozeny, ale čekají ve frontě. Bude-li některá z úloh dokončena dříve, mohou být odmítnuté úlohy opět naplánovány.

Kapitola 6

Výběr platformy

Výběr implementačních nástrojů je důležitý krok v průběhu projektu. Nástroje a další případné prostředky je nutné vhodně zvolit. Nástroje jsem zvolil po domluvě s vedoucím diplomové práce. Vzhledem k dohodnutému cíli, že budou experimenty s algoritmy prováděny na reálném hardwaru a k dosavadním zkušenostem autora, zvolil jsem nástroje následovně. Cílový RT systém je μC/OS-II, a tedy implementační jazyk je jazyk C. Dále jako hardware byla vedoucím poskytnuta vývojová deska demoQE128 od firmy P&E [9]. Tyto nástroje dále krátce popíšu.

6.1 μC/OS-II

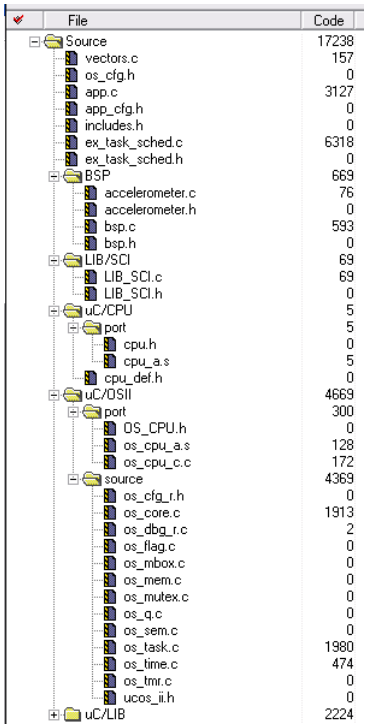
μC/OS-II je male real-time jádro napsané panem Jean J. Labrosse [22]. Jeho základní vlastnosti jsou preemptivnost, škálovatelnost, přenositelnost, víceúlohovost a prioritní plánování. Celé jádro je napsané v jazyku ANSI C. V dnešní době je toto jádro dostupné již ve své 3. verzi. K vývojové desce byla dostupná verze 2.85, již jsem použil. Níže jsem v krátkosti popsal strukturu souboru RT sytému, pro případné podrobnější informace o μC/OS-II doporučuji knihu [16].

Celý kód jádra μC/OS-II je rozdělen do 14 souborů, kde každý soubor obsahuje funkce pro jednotlivé součásti jádra. Hlavní služby jádra se nacházejí v souboru `os_core.c`. Rozšiřující služby jádra pro komunikaci mezi úlohami (semafory, poštovní schránky, fronty zpráv), práci s časem či s úlohami jsou v samostatných souborech. Tyto rozšiřující služby pak mohou být vypnuty či zapnuty. Vypnutím nepotřebných funkcí je možné zmenšit výslednou velikost koncové aplikace. Celá struktura souborů jádra je vidět na obrázku 6.1. Uvedené základní soubory jádra jsou nezávislé na platformě. Aby bylo možné jádro přenést na různé platformy, jsou součástí RT sytému ještě soubory závislé na platformě, tzn. musí být pro každý hardware vytvořeny na míru. Tyto soubory jsou vidět na obrázku ve složce `µC/OS-II` a dále ve složce `port`. Soubory obsahují specifické informace o procesoru.

Rád bych ještě zmínil, že ve složce `µC/CPU` se nacházejí soubory s kódem pro vstup, resp. výstup z kritických sekcí. Složka `BSP` (board support package) obsahuje funkce pro práci s periferiemi nacházejícími se na desce, např. led diody či akcelerometr. Složka `LIB/SCI` obsahuje kód pro nastavení komunikace skrz virtuální COM port. Tato komunikace pak probíhá skrze USB kabel.

Ve struktuře souboru se ještě nacházejí soubory, které nejsou v žádné složce. Postupně krátce popíši, co obsahují:

- `os_cfg.c` - hlavičkový soubor, kde je možné zapnout či vypnout různé služby jádra



Obrázek 6.1: Struktura souborů portu jádra pro demoQE128

- `vectros.c` - obsahuje mapování rutin na vzniklá přerušení
- `includes.h` - jak název napovídá, obsahuje pouze hlavičkové soubory
- `app.c` - obsahuje uživatelský kód. Zde je možné vytvářet úlohy a nastavovat jejich parametry.
- `app_cfg.h` - hlavičkový soubor, kde se nastavuje např. počet úloh systému či možnost logování
- `ex_task_sched` - obsahuje kód rozšiřující jádro RT sytému o nové plánovací algoritmy

6.2 Vývojová deska demoQE128

Jedná se o vývojovou desku, kterou je možné osadit 32-bitovým procesorem MCF51QE128 [10] či 8-bitovým procesorem MC9S08QE128 [11]. V implementaci bude využit 8-bitový procesor s jádrem HCS08, jelikož port jádra μC/OS-II je určen právě pro tento procesor. Deska je dále osazena různými periferiemy, např. led diody, tlačítka, potenciometr, akcelerometr či port RS232. V práci jsem tyto periférie nepoužil, proto je nebudu dále více popisovat. Podrobnější informace o desce a osazených periféríích lze nalézt v mauálu, který je dodáván spolu s vývojovou deskou.

6.2.1 Mikroprocesor MC9S08QE128

Tento procesor patří do řady nízkopříkonových procesorů Flexis QE128. Může pracovat až na maximální frekvenci 50 MHz při napájecím napětí nad 2,4 V, zatímco sběrnice pracuje

s polovičním kmitočtem hodin, tedy maximálně 25 MHz. Jádro je však schopno pracovat již při napájení od 1,8 V. Jádro má nízké energetické nároky a podporuje i úsporné mody, proto je vhodné k aplikacím s nízkými energetickými nároky či aplikacím napájeným z akumulátoru.

Původní myšlenkou bylo implementovat algoritmy, a zejména u algoritmů pro plánování při nízkém napětí změřit reálně spotřebovanou energii. U desky je sice možné měnit frekvenci mikroprocesoru a sběrnice, bohužel pouze dělením generátoru hodinového kmitočtu. Tímto dělením lze docílit celkem čtyř frekvencí, vyjádřených procentuálně 25%, 50%, 75% a 100%. Následkem tohoto chování je, že odebraná energie se nikterak nemění. Procesor je nízkopříkonový, bohužel neumožňuje DVS.

K programování je k dispozici 128 kB programové flash paměti a dále pro data ještě 8 kB SRAM paměti. Jako zdroj hodinového kmitočtu může být použit jak vnější zdroj (krystal), tak vnitřní zdroj.

Z periferie procesoru uvádím jen některé: 24-kanálový analogově-číslicový převodník s rozlišením 12-bitů, dvojice sériových sběrnic kompatibilních se standardem I2C či pro nás důležitá dvojice sériových rozhraní s plným duplexním režimem a programovatelnou přenosovou rychlostí.

6.3 Software

Posledním důležitým nástrojem k vývoji je zvolený software. Jelikož jsem jako cílovou platformu zvolil vývojovou desku s mikroprocesorem HCS08 od firmy Freescale, bylo vývojové prostředí v podstatě dopředu známé. Jedná se o Codewarrior verze 6.3, které jsou součástí balení. Jedná se o velmi dobré a intuitivní prostředí, které je možné docela rychle ovládnout. Funkce prostředí jsou editace kódu, nahrání aplikace do vývojové desky a velmi důležité krokování nahraného kódu v desce. Netřeba dodávat, že funkce posledně jmenované jsem hojně využíval při ladění kódu.

Krokování kódu je dobrá metoda k odhalování chyb, bohužel nenapoví při ověřování správnosti kódu. Jelikož součásti desky není obrazovka, kde by mohly být znázorněny průběhy algoritmů, je pro získávání dat z desky využito sériové rozhraní. Přes toto rozhraní jsou posílány údaje o změnách v plánování. Samotný text již pomůže k nalezení chyb, avšak není to také zrovna nejlepší forma pro porovnávání průběhu vykonávání úloh. K vizualizaci textových informací jsem v práci použil jednoduchou aplikaci *Grasp* [27]. Tuto aplikaci vytvořil Mike Holenderski v jazyku TCL. Výhodou aplikace je jednoduchá struktura logu, který může být pomocí ní vykreslen. Více o formátu log souborů je obsaženo v 7.4.

Kapitola 7

Implementace

V této kapitole podrobněji rozeberu implementaci zvolených algoritmů, popis případných změn provedených v kódu jádra a principy funkčnosti. Snahou při implementaci algoritmu do jádra μC/OS-II bylo provedení co nejmenšího počtu změn v jádře. Důvodem bylo případné přenesení vytvořených algoritmů na jinou platformu. Konečné řešení nakonec neprovádí v kódu jádra žádné změny a veškerý nový kód je vyčleněn do nových souborů. Důležitým faktem, který je nutné zmínit, je, že v celém kódu se pracuje pouze s celočíselnými proměnnými. Toto rozhodnutí bylo vytvořeno, jelikož proměnné s plovoucí desetinou čárkou zabírají více místa, ale hlavně operace s těmito čísly jsou pomalé. Práce s celými čísly jistě ovlivňuje účinnost některých algoritmů, ale na druhou stranu práce s desetinou čárkou by měla větší negativní vliv.

7.1 Upravené úlohy

Informace o úlohách jsou v μC/OS-II uchovávány v TCB blocích. TCB blok může dle nastavení systému obsahovat až 26 položek. Pro potřeby implementace plánovacích algoritmů bohužel tyto údaje nepostačují. Již při návrhu jádra μC/OS-II myslel autor na možnost rozšíření základních informací o úlohách. Mimo jiné se tedy mezi základními informacemi nachází ukazatel na možné další údaje. Tohoto řešení jsem využil k přidání potřebných informací k plánování.

7.1.1 Rozšířený TCB blok

Pro dodatečné informace jsem vytvořil novou datovou strukturu `EX_TCB`, která obsahuje 17 dalších položek. Všechny položky jsou i s popisem uvedeny v 7.1. Vzhledem k tomu, že rozšířený TCB blok je využíván jak pro plánování úloh při přetížení, tak i pro plánování za nízké spotřeby, jsou některé položky využity rozdílně dle zvolených algoritmů.

7.1.2 Aperiodické úlohy

Pro ověření plánovacích algoritmů při přetížení je potřeba aperiodických úloh. Aby byl zajištěn příchod úloh v různých časových intervalech, využil jsem pro generování dalšího příchodu úlohy generátor pseudo-náhodných čísel `rand()`, který je součástí knihovny jazyka ANSI C. Generování nového čísla je provedeno vždy při novém spuštění úlohy. Důvodem přenesení generování čísla do smyčky úlohy je výpočetní náročnost, která by zpomalovala

```
typedef struct  ex_tcb {
CPU_BOOLEAN  Aperiodic;
CPU_BOOLEAN  Stop;
CPU_BOOLEAN  Done;
CPU_BOOLEAN  D_pri;
CPU_INT08U   Value;
CPU_INT16U   Dead;
CPU_INT16U   Util;
CPU_INT16U   DeadAct;
CPU_INT16S   DeadTolerance;
CPU_INT16U   WCExecT;
CPU_INT16U   ExecRem;
CPU_INT16U   LastExecT;
CPU_INT16U   Period;
CPU_INT16U   Sleep;
CPU_INT16U   Job;
} EX_TCB;
```

přetížení	nízké napětí
typ úlohy aperiodická/periodická (TRUE/FALSE)	
indikuje, že došlo k preempci úlohy	
indikuje, že bylo vykonávání úlohy dokončeno	
skupina úloh do které patří	-
hodnota úlohy	čas dalšího spuštění úlohy
časová mez úlohy	
vytížení úlohy	
aktuální časová mez úlohy	
tolerance časové meze (RED)	nejmenší čas potřebný k vykonání úlohy (BCET)
čas potřebný k vykonání ulohy C	nejdelší čas potřebný k vykonání úlohy (WCET)
zbývající čas potřebný k vykonání úlohy	
-	doba vykonávání úlohy v posledním spuštění
perioda úlohy P, pro aperiodické se používá k výpočtu dalšího příchodu	
doba jak dlouho bude úloha v systému uspána	
instance úlohy	

Obrázek 7.1: Rozšířený TCB plok

plánovač. Vygenerované číslo je uloženo v položce `Sleep` rozšířeného TCB. Jakmile je úloha dokončena, je na dobu `Sleep` uspána pomocí funkce `OS_EX_TimeDly()` 7.2.3.

7.1.3 Proměnná délka vykonávání úlohy

Algoritmy pro plánování při nízké spotřebě pracují s předpokladem, že doba vykonávání úlohy může a často také je menší než nejhorší doba běhu úlohy $WCET$. Nevyužitý čas je pak dále použit k možnému snížení frekvence, resp. napětí. Z tohoto důvodu je třeba pro úlohy opět náhodně generovat doby běhu. Zde jsem opět využil generátor pseudo-náhodných čísel `rand()`. Jak tady, tak u aperiodických úloh využitím `rand()` dostáváme čísla v uniformním rozložení. V různých článcích (např.[8]) či knihách (např. [3]) používají autoři pro generování doby běhu úlohy často Poissonovo rozložení. Z důvodu vyšší výpočetní náročnosti jsem toto rozložení nepoužil. Při použití výkonnějšího procesoru by to již pravděpodobně bylo možné.

7.1.4 Vytvoření úloh

K vytvoření úloh μC/OS-II používá funkce `OSTaskCreate()` a `OSTaskCreateExt()`. Jelikož je nutné úlohám přiřadit rozšířený TCB blok, využíváme k vytvoření úloh funkci `OSTaskCreateExt()`. Aby případný uživatel nemusel stále ručně psát rozsáhlý kód při každém tvoření úlohy, vytvořil jsem funkci `GenerateTasks()`, která dle nastavené hodnoty `COMM_NUM_TASKS` vytvoří požadovaný počet úloh. `COMM_NUM_TASKS` se nalézá v souboru `app_cfg.h`. Takto vytvořené úlohy ještě nemají nastavené žádné údaje pro plánování. K nastavení informací o úloze slouží funkce `InitTaskAttributes()`. Touto funkcí se tedy nastaví např. typ úlohy, časová mez úlohy, perioda a čas běhu úlohy. Celý proces vytváření úloh se provádí ještě před spuštěním systému ve funkci `AppCreateTasks()`. Jedná se vlastně o jediné místo v aplikaci, kde je potřeba, aby uživatel zadal informace systému. Nachází se zde i volání funkcí k nastavení komunikace přes sériový port, nastavení frekvence procesoru a zvolení plánovacího algoritmu.

7.1.5 Tělo úlohy

Při každém spuštění úlohy v μC/OS-II je vykonávána funkce, která je úlohám přiřazena již při vytvoření. Funkce se skládá z nekonečné smyčky, ve které mohou být prováděny další úkony. V našem případě se uvnitř smyčky nachází tři důležité součásti. Jedná-li se o aperiodickou úlohu, je při spuštění úlohy vygenerován čas jejího příštího spuštění. Dále se zde nachází další nekonečná smyčka. Pokud je zapnuté logování, dochází v druhé smyčce k zápisu logovacích údajů do sériové linky. Dojde-li k preempci či dokončení úlohy, je smyčka zastavena. Ve skutečnosti je však smyčka zastavena až při dalším spuštění úlohy. Zda došlo k preempci či dokončení úlohy, pozná úloha dle hodnot nastavených v rozšířeném TCB bloku. Pokud je pravdivá hodnota v proměnné `Stop`, došlo k preempci. Při pravdivé hodnotě v proměnné `Done`, byla-li úloha dokončena, nebo překročila-li časovou mez. V tomto případě je při plánování při přetížení úlohám vygenerována příští doba běhu úlohy.

7.2 Princip plánování

7.2.1 Body plánování a čas

K přeplánování v systému dochází při příchodu nové úlohy, dokončení úlohy a nebo když je volnost úlohy $L = 0$ (D^{over}). Abychom zjistili, kdy tyto body nastanou, musíme vědět o všech úlohách, jak dlouho poběží, kdy budou znovu spuštěny a další informace. Všechny tyto informace mají společné to, že se mění v čase. Důležitým prvkem v sytému je tedy měření času. Čas běhu aplikace lze získat voláním funkce `OSTimeGet()`. Abychom mohli aktualizovat údaje o úlohách, je pro aplikaci důležitá funkce `OSTimeTickHook()`, která je spuštěna při každé změně času v sytému. Poznamenejme, že v čase volání funkce `OSTimeTickHook()` ještě nedošlo k posunu času v systému.

Do funkce `OSTimeTickHook()` může být vložen uživatelský kód, který bude v každém tiku proveden. V tomto místě je u všech úloh, které jsou připraveny k běhu, snížena hodnota aktuální časové meze. U právě běžící úlohy je navíc snížena doba potřebná k vykonání úlohy. Dále se testuje, je-li úloha dokončena, resp. je-li konec časové meze. Pokud je testování pozitivní, je úloha uspána do doby dalšího spuštění a navíc je nastavena proměnná `resched` na hodnotu `OS_TRUE`, čímž dojde k volání algoritmu plánování. K lepšímu pochopení je v příloze C.2 znázorněn diagram funkcí `OSTimeTickHook()` a `ExSched()`.

7.2.2 Priority a plánování

Původní plánovač jádra μC/OS-II je čistě prioritní, tzn. že je vždy spuštěna úloha s nejvyšší prioritou. V jádru μC/OS-II je nejvyšší priorita 0 a nejmenší 63, označována také jako `OS_LOWEST_PRIO`. Každá priorita může být přiřazena právě jedné úloze. Tento přístup není bohužel vhodný pro práci s vybranými algoritmy, jelikož se u nich priority dynamicky mění. Naštěstí dovoluje jádro měnit úlohám priority voláním funkce `OSChangePrio()`. Aby nedocházelo v každém bodu plánování k neustálému přiřazování priorit, je úloze, která má běžet, vždy přiřazena priorita větší než priorita všech úloh v systému. Tato priorita je v sytému označena `OS_EX_TASK_RUN_PRIO`. Dojde-li k přeplánování úloh, vrátí běžící úloha prioritu a nastaví si svou původní prioritu. Nově vybraná úloha si pak opět změní prioritu na `OS_EX_TASK_RUN_PRIO`, čímž si zajistí spuštění pomocí původního plánovače μC/OS-II. V podstatě se tedy jedná o jakési předávání jakéhosi tiketu, který umožňuje úloze běžet.

Jak je uvedeno výše, ke změně priorit poskytuje jádro funkci`OSChangePrio()`. Jelikož ke změnám priorit dochází vždy v těle funkce `OSTimeTickHook()`, není možné původní funkci

pro změnu priorit použít. Důvodem je kontrola, zda není funkce volána při obsluze přerušení. Funkce `OSTimeTickHook()` je součástí obsluhy přerušení. Aby nemusela být měněna původní funkce jádra, vytvořil jsem obdobnou funkci `OSTaskChangePrio()`, která neobsahuje kontrolu volání z přerušení. Toto řešení bylo otestováno a nebyly zaznamenány žádné změny v chování systému.

7.2.3 Uspání a vzbuzení úlohy

Uspávání či buzení úloh probíhá v těle funkce `OSTimeTickHook()`. Je-li úloha dokončena, a nebo promeškala svoji časovou mez, musí být uspána. Podobně jako u změn priorit původní funkce, jádra pro uspávání a buzení úloh nemohou být spuštěny v obsluze přerušení. Opět tedy, aby nemuselo být změněno původní jádro, jsem vytvořil nové funkce `OS_EX_TimeDly` pro uspání a `OS_EX_TimeDly_Resume()` pro vzbuzení, které neprovádí tuto kontrolu.

Jádro μC/OS-II dokáže samo pomocí svých funkcí vzbudit úlohu, může tedy vzniknout otázka, proč potřebujeme budit úlohy dříve. Odpověď je vcelku jednoduchá. Ačkoliv je funkce `OSTimeTickHook()` volána při dalším tiku, stále nedošlo ke změně času. Úlohy tedy budou vzbuzeny až po provedení těla této funkce. Naše rozšířené plánování ale probíhá ještě před dokončením `OSTimeTickHook()`. Z tohoto důvodu jsou vzbuzeny rozšířené úlohy, které mají `OSTimeDly = 1`. Díky vzbuzení těchto úloh mohou plánovací algoritmy pracovat s aktuální sadou připravených úloh.

7.2.4 Plánování

Jak je uvedeno v 7.2.1, má-li dojít k přeplánování, je nastavena proměnná `resched` na `OS_TRUE`. Tímto je zajištěno, že bude spuštěna funkce `ExSched()`. Na základě zvoleného algoritmu je zavolána funkce konkrétního algoritmu. Funkce vrátí odkaz na úlohu, která bude spuštěna. Této úloze je nastavena priorita `OS_EX_TASK_RUN_PRIO`. Je-li zapnuto logování, dojde ještě k zaznamenání případné akce.

7.2.5 Změny frekvence

Při plánování pro nízký příkon dochází ke změnám frekvence procesoru a s tím i ke změnám doby vykonávání úloh. Úloha běžící 1 vteřinu při 100% výkonu poběží 4 vteřiny při 25% výkonu. Tato skutečnost se musí jistým způsobem promítnout do systému uchovávání informací o úlohách. Celý princip je následující. Při nastavování informací o úlohách před spuštěním systému jsou veškeré informace pracující s časem vynásobenym hodnotou 4. Aktuální hodnoty zbývající doby běhů úloh jsou pak snižovány dle frekvence procesoru. Časová mez úlohy je vždy snižována o hodnotu 4, jelikož se snížením frekvence její velikost nemění. Pro změnu frekvence procesoru jsem implementoval funkci `FLL_Change()`, která kromě změny frekvence upraví i nastavení komunikačního sériového portu. Důvod úprav v nastavení je provázání frekvence sběrnice s přenosovou rychlostí sériového portu.

7.3 Implementace plánovacích algoritmů

Cílem každého plánovacího algoritmu je sestavit plán a vybrat z plánu úlohu, která bude spuštěna. V implementovaných algoritmech je výběr proveden dle jejich pravidel. Algoritmy jsem implementoval tak, jak byly navrženy jejich autory a popsány v teoretické části práce. U některých algoritmů došlo k nepatrným změnám, jelikož musely být upraveny pro práci

s celými čísly. Princip předávání priorit, uspávání a buzení úloh jsem popsal již výše, proto se při popisu implementace plánovacích algoritmů zaměřím pouze na způsob výběru úlohy, která bude spuštěna.

7.3.1 Podpůrné prostředky

Ačkoliv již byly zmíněny některé podpůrné funkce využívané v implementaci, pro potřeby plánovacích algoritmů jsem vytvořil další, o kterých ještě nebylo nic uvedeno. Budu-li dále v textu o plánovacích algoritmech hovořit o změně frekvence, myslím tím zavolání funkce `FLL_Change()`. Pro některé složitější algoritmy bylo nutné vytvořit pole `valDenArr[]` prvků typu `DASA_LOAD`. Název `DASA_LOAD` je možná trochu zavádějící, jelikož toto pole není využito pouze v mechanismu DASA. Důležitější je, že se skládá z ukazatele na TCB blok, celočíselné bezznaménkové proměnné a celočíselné znaménkové proměnné. Dále jsem pro potřeby algoritmů vytvořil funkci `QuickSort()`. Jak již název napovídá, jedná se o řadící algoritmus quicksort. Řazení prvků je prováděno nad již výše uvedeným polem `valDenArr []`. Poslední důležitou podpůrnou funkcí je `sqrt()`. Jedná se o počítán9 odmocniny pro celá čísla. Důvodem implementace byla absence v dodávané knihovně pro procesor HCS08.

7.3.2 EDF

Všechny implementované algoritmy mají jednu společnou věc, a sice základ v algoritmu EDF. Kritériem výběru běžící úlohy je časová mez úlohy D_i. Algoritmus je celkem jednoduchý, proto ani implementace nebyla nikterak složitá.

Nejprve je získán první prvek seznamu TCB bloků úloh, který je uchováván v `OSTCBList`. Následně procházíme všechny úlohy a kontrolujeme, zda se jedná o úlohy s rozšířeným TCB a zda úloha nespí. Mezi úlohami, splňujícími tato kritéria, pak hledáme tu, která má nejmenší relativní časovou mez D_i. Výstupem algoritmu je ukazatel na TCB blok úlohy, která bude spuštěna. Dodávám, že nenachází-li se v systému úloha, která by mohla být spuštěna, vrací algoritmus hodnotu `NULL`. Potom je tedy spuštěna IDLE úloha. Celý algoritmus je implementován funkcí `EDF_sched()`.

7.3.3 Statické EDF

Plánovací mechanismus statického EDF je spuštěn vždy ještě před spuštěním systému. Jedná se o jediný off-line algoritmus, který byl implementován. Implementace jsem rozdělil do dvou funkcí. První funkce `staticEDF()` zkontroluje, zda systém již běží. Pokud systém ještě nebyl spuštěn, je zavolána funkce `SetFreq()`, která nastaví frekvenci procesoru. Na zvolené frekvenci pak běží systém po celý čas. V těle funkce `SetFreq()` je sečteno celkové vytížení procesoru U. Dle velikosti U je pak nastavena frekvence procesoru, při které budou splněny časové meze všech úloh. Nakonec je vždy zavolána funkce `EDF_sched()`, která vrátí ukazatel na úlohu.

7.3.4 ccEDF

Implementace tohoto on-line algoritmu je do značné míry podobná s předešlým algoritmem. Samozřejmě se nemění frekvence jen před spuštěním, ale vždy, kdy je třeba. Algoritmus je implementován funkcí `ccEDF()`, která se skládá pouze ze dvou řádků. První volá funkci `SetFreq()` pro výběr frekvence a druhý volá funkci `EDF_sched()` pro plánování.

CcEDF funguje na principu šetření cyklů. V 4.3 je dobře vidět, jakým způsobem dochází k výběru frekvence. Aby mohla být použita funkce `SetFreq()`, musel jsem aktualizovat údaje o vytížení U_i veškerých úloh τ_i dle pravidel ccEDF. K tomuto dochází v těle funkce `OSTimeTickHook()`. Při doknčení úlohy je vytížení vypočteno jako podíl doby potřebné k posledním běhu úlohy cc_i a periody T_i. Při dalším příchodu úlohy je pak vytížení počítáno jako podíl nejhoršího času potřebného k běhu C_i a periody T_i.

7.3.5 laEDF

LaEDF je již složitější on-line algoritmus. Implementaci jsem rozčlenil do tří funkcí. Funkce `laEDF()` obsahuje jen volání funkce `lookAheadFreq()` a `EDF_sched()`. Nejdůležitější činnost je prováděna v prvně jmenované funkci. Na začátku funkce je nejprve provedena funkce `lookAheadLoad()`, která uloží do pole `valDenArr`[] hodnoty aktuálního vytížení úlohy U_i, aktuální časové meze D_i a ukazatel na TCB úlohy. Navíc ještě provede součet vytížení všech úloh. Údaje, uložené v poli, jsou následně seřazeny dle časových mezí. Do pomocné proměnné je ještě před samotným výpočtem potřeba uložit nejbližší časovou mez systému. Pak již procházíme pole od úlohy s největší D_i až po nejmenší a provádíme výpočty nutné ke stanovení frekvence dle návrhu. Jakmile jsou vypočteny všechny nutné údaje, stanoví se požadovaná frekvence, která je následně nastavena.

7.3.6 lppsEDF

Jak je uvedeno v návrhu 4.5.3, pracuje lppsEDF v součinnosti s off-line algoritmem. Před spuštěním je určena nejnižší možná frekvence, při které jsou splněny časové meze všech úloh. To je provedeno voláním funkce `SetFreq()`. Výsledná hodnota je uložena pro budoucí využití do globální proměnné `lppsMaxSpeed`. Při spuštění systému algoritmus vždy zjistí, kolik úloh je připraveno k běhu. Při dvou a více úlohách je frekvence nastavena dle `lppsMaxSpeed`. Pokud se však v systému nachází pouze jedna úloha, stanoví se frekvence tak, aby byla úloha rozprostřena mezi aktuální čas a čas nejbližšího příchodu další úlohy. Pokud je časová mez úlohy menší než čas příchodu další úlohy, použije se pro výpočet časová mez. Následně je změněna frekvence a vybrána úloha, která bude spuštěna pomocí `EDF_sched()`.

7.3.7 DASA

Implementace DASA jsem rozdělil do pěti funkcí. Hlavní funkcí je `DASA()` a pomocné funkce pro výpočty jsou `LoadValueDensity()`, `DASAComputeLoad()`, `UnsetPrivTask()` a `DASA_EDF()`. Postup výpočtu je následující. Nejprve je zavolána funkce `LoadValueDensity()`, která do pole `valDenArr`[] uloží váhu užitku úlohy VU_i 5.5. Toto je provedeno jen pro úlohy, které jsou připraveny k běhu, navíc je ještě zjištěn počet připravených úloh. Pokud je připravena jen jedna úloha, vrací ihned algoritmus ukazatel na její TCB, pokud žádná, vrací ukazatel na `NULL`.

Při více připravených úlohách se pokračuje takto. Pomocí funkce `DASAComputeLoad()` je vypočteno maximální vytížení systému . Nedošlo-li k přetížení systému, je pomocí `EDF_sched()` vybrána úloha, která bude spuštěna. Dojde-li k přetížení systému, je u všech úloh pomocí funkce `UnsetPrivTask()` nastavena proměnná `D_pri` na hodnoty `OS_FALSE`, což znamená, že žádná z úloh není momentálně přidána do prozatimního plánu PP. Zavoláním funkce `QuickSort()` je pole `valDenArr`[] seřazeno dle časové meze úloh a úlohy jsou pak v sestupném pořadí přidávány do plánu PP. V každém kroku přidání úloh je funkcí

`DASAComputeLoad()` zkontrolováno, zda přidání úlohy do plánu nezpůsobilo přetížení. Když je způsobeno přetížení, je úloha z plánu PP zase odstraněna. Nakonec je pomocí funkce `DASA_EDF()` vybrána úloha, která poběží. `DASA_EDF()` oproti `EDF_sched()` bere navíc ohled na proměnnou `D_pri` při plánováni s plánem PP. Pro lepší pochopení průběhu algoritmu lze využít diagram C.5 nacházející se v přílohách.

7.3.8 D^{over}

D^{over} je algoritmus pracující s relativní volností úlohy $L_i(t)$. Poznamenávám, že přetížení je indikováno, pokud kterákoliv úloha má $L_i(t) = 0$. Implementoval jsem ho pomocí 4 funkcí. Funkce `DOVERLoad()` zkontroluje všechny úlohy a případně zjistí, zda došlo k vytížení. Dále pro případné další výpočty ukládá $L_i(t)$ úloh do pole `valDenArr[]`, zjišťuje nejvyšší (`VMax`) a nejnižší (`VMin`) hodnotu úloh v systému a sečte hodnotu všech privilegovaných úloh v systému (`VPriv`). Nedojde-li k přetížení, plánuje se opět pomocí `EDF_sched()`.

Při diagnostikování přetížení je úloha, která ho způsobila ($L_i(t) = 0$) při splnění podmínky 5.6, vybrána ke spuštění. Kontrolu podmínky provádí funkce `DOVERComputeCond()`. Pokud úloha podmínku nesplní, je zahozena pomocí funkce `DOVERAbandonTask()`. Nastane-li situace, kdy více úloh má $L_i(t) = 0$, je prověřena podmínka 5.6 v `DOVERComputeCond()` a jedna z úloh je následně opět odstraněna ze sytému.

Úlohy jsou označeny jako privilegované, pokud bylo přerušeno jejich vykonávání preempcí. To se děje v těle funkce `ExSched()` nastavením `D_Pri` na hodnotu `OS_TRUE`. Při detekci přetížení jsou všechny privilegovaná úlohy přesunuty zpět mezi čekající voláním funkce `UnsetPrivTask()`. Hlavní funkcí zastřešující D^{over} algoritmus je `DOVER()`. Na konci je ještě zavolána funkce `EDF_sched()` pro výběr úlohy.

Jak jsem zmínil na začátku kapitoly, některé algoritmy jsem do jisté míry poupravil pro práci v celočíselném prostředí. Algoritmus D^{over} v podmínkách 5.6 a 5.7 používá druhou odmocninu. Její implementace již byla také zmíněna. K lepšímu pochopení je v přílohách znázorněn digram průběhu algoritmu C.4.

7.3.9 RED

RED jsem opět rozdělil do několika funkcí. Hlavní funkce `RED()` nejprve zavolá funkci `REDLoad()`, která zjistí počet připravených úloh a naplní pole `valDenArr[]` údaji o relativní časové mezi úlohy D_i a zbývající době výpočtu C_i. Je-li jedna připravená úloha, je návratovou hodnotou ukazatel na její TCB, pokud není žádná úloha připravena, je vrácen ukazatel NULL. Při více připravených úlohách je pole `valDenArr[]` seřazeno dle Di a zavolána funkce `REDComputeLaxity()`.

`REDComputeLaxity()` implementuje podmínky pro výpočet [3, str. 308-311]. V poli `valDenArr[]` na místo C_i ukládá $L_i(t)$, místo D_i uloží hodnotu úlohy V_i, vypočítá maximální možné překročení času `E_max` a vrátí ukazatel na TCB úlohy, která bude nejdříve ukončena.

Pokud `E_max > 1`, došlo k přetížení. Pole je znovu seřazeno, tentokrát podle hodnot úloh V_i, které byly uloženy ve funkci `REDComputeLaxity()`. Následně jsou úlohy postupně kontrolovány funkcí `REDRejection()` a pokud neprojde podmínkou, je úloha odstraněna funkcí `DOVERAbandonTask()`. Nakonec je ze zbývajících úloh vybrána úloha, která bude spuštěna. V příloze na obrázku C.3 je znázorněn jednoduchý diagram průběhu algoritmu.

Návrh algoritmu RED počítá s proměnnou délkou vykonávání úloh, proto nejsou úlohy ihned odstraněny ze sytému, ale pouze odloženy. Vzhledem k tomu, že v rámci implementace algoritmu pro přetížení není proměnná délka vykonnávání úloh implementována a pracuje

se vždy s nejhorším časem běhu úlohy, byla provedena změna, tzn. úlohy jsou při nesplnění podmínky ihned odstraněny ze sytému pomocí `DOVERAbandonTask()`.

7.4 Logování

V úvodu kapitoly jsem několikrát zmínil možnost logování běhu systému. Jedná se o velmi důležitou činnost při testování sytému, ověřování správností algoritmu a jejich porovnávání. Jelikož pro vizualizaci byla zvolena aplikace GRASP, popíšu nejprve notaci logovacích informací. Pro jednoduchost uvedu příklad a popíšu význam jednotlivých příkazů.

```
newTask task1 -priority 7 -name "Uloha 1"
newTask task2 -priority 8 -name "Uloha 2"
plot 5 jobArrived job2.1 task2
plot 5 jobResumed job2.1
plot 5 taskAnnotation task1 50 -message "text zpravy"-arrow both -color red
plot 20 jobArrived job1.1 task1
plot 20 jobPreempted job2.1 -target job1.1
plot 20 jobResumed job1.1
plot 35 jobCompleted job1.1 -target job2.1
plot 35 jobResumed job2.1
plot 40 taskDeadline task1
plot 45 jobDeadline job2.1
plot 50 jobCompleted job2.1
```

7.4.1 Syntaxe logů

První příkaz `newTask` sdělí aplikaci, že byla vytvořena úloha. Hodnota `task1` je jednoznačný identifikár úlohy pro pozdější příkazy. Parametr priority určuje pořadí zobrazených úloh pod sebou a `name` její název ve vizualizaci. Další příkazy již vždy obsahuji prefix `plot čas`. Hodnota času udává, kde bude daná událost v časové linii zaznamenána.

Další příkaz `plot 5 jobArrived job2.1 task2`, sděluje aplikaci, že úloha `task2` je připravena k běhu. Parametr `job2.1` sděluje, že se jedná o první spuštění úlohy 2 v systému.

Pokud dojde ke spuštění úlohy, je to zaznamenáno pomocí `jobResumed` a identifikátoru instance úlohy `job2.1`. Pokud je úloha dokončena, zaznamená se `jobCompleted job2.1 -target job1.1`. Tato formulace říká, že úloha instance úlohy `job2.1` byla dokončena a parametr `-target job1.1` sděluje, která úloha bude následně spuštěna. Při přepnutí z jedné úlohy na druhou je syntaxe podobná. Rozdíl spočívá pouze v záměně slova `jobCompleted` za `jobPreempted`.

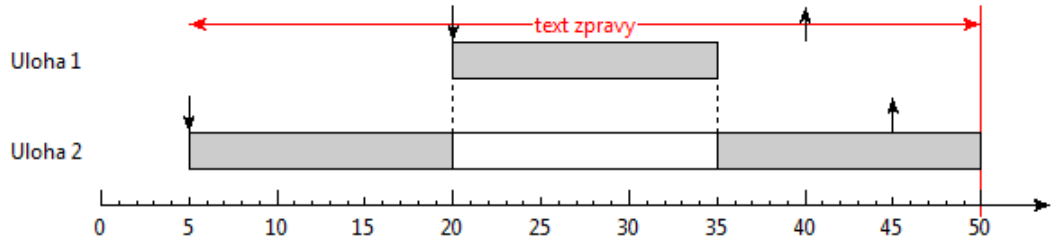
Dalším příkazem uvedeným v ukázce je `plot 45 jobDeadline job2.1`. Tento příkaz pouze sdělí aplikaci, aby zaznamenala čas, kdy končí úloze časová mez. Poslední nepopsaný příkaz je `taskAnnotation`. V příkladu příkaz vytvoří úsečku s šipkami na koncích (`-arrow both`) a umístí ji nad úlohou `task1`. Úsečka bude vykreslena od času 5 do času 50 a bude obsahovat zprávu uvedenou za parametrem `-message`. Výsledek vizualizace je vidět na obrázku 7.2.

7.4.2 Implementace logování

Samotnou implementaci logování jsem realizoval pomocí kruhového seznamu prvků typu `EventInfo`. EventInfo obsahuje šest položek potřebných pro zaznamenání všech událostí. Obsahuje čas, kdy se událost vyskytla, typ události, identifikační číslo úlohy, číslo instance úlohy a případně ještě identifikační číslo úlohy, která bude následně spuštěna, a také její číslo instance. Kruhový seznam `eventLog`[] jsem zvolil, jelikož byl nejvhodnější pro zaznamenávání předem neznámého počtu událostí. Počet maximálního počtu prvků je zvolen v závislosti na počtu úloh v systému již před spuštěním. V zásadě se jedná o pole prvku, ke kterému si ještě uchováváme index prvního platného prvku pole `indexEvent` a počet uložených událostí `numEvents`.

Ukládání událostí do seznamu je implementováno do funkce `LogTaskEvent()`. Výpis informací na terminál provádí funkce `LogTaskPrint()` v součinnosti s funkcí `LogTaskOne()`. Funkce `LogTaskPrint()` je umístěna v těle každé úlohy, aby nebylo zatěžováno plánování. Funkce je umístěna i v těle IDLE úlohy.

Logování lze případně vypnout nastavením `OS_EX_GRASP_LOG_EN` na hodnotu 0 v souboru `app_cfg.h`. Volání funkce `LogTaskEvent()` pro zaznamenání události je umístěno v těle funkce `ExSched()`. Výběr události je rozlišen dle změn v plánování úloh. Kromě této funkce se zaznamenávají události v `ExSchedTickHook()`, zejména příchod nové úlohy. Dále k zaznamenávání události dochází ve funkci `DOVERAbandonTask()`, kde je třeba zaznamenat nedokončení úlohy. Posledním místem je funkce `FLL_Change()`. Zde se zaznamenává pomocí události taskAnnotation, na jaké frekvenci pracoval procesor v daný interval.



Obrázek 7.2: Vizualizace ukázkového příkladu v aplikaci Grasp

Kapitola 8

Experimenty

V této kapitole je uvedeno, jakým postupem byly provedeny experimentální měření. Získané výsledky prezentuji pomocí grafů a tabulek, které jsou též patřičně okomentovány. Jsou zde uvedeny testovací sady úloh, jejich nastavení a případná nastavení sytému pro simulaci. Také jsou zde uvedeny omezení kladená na systém a postup interpretace výsledků.

8.1 Algoritmy pro nízký příkon

Experimenty jsem prováděl se čtyřmi algoritmy určenými pro plánování při nízké spotřebě. Jedná se o algoritmy statické EDF (dále již jen statEDF), ccEDF, dopředné EDF (laEDF) a nízko energetické prioritní EDF (lppsEDF). Tyto algoritmy jsou implementovány pomocí ANSI C do jádra RT sytému muC/OS-II, viz kapitola 7.

8.1.1 Sady úloh

Pro ověření vlastností algoritmů jsem zvolil dvě sady reálných úloh a čtyři sady náhodných úloh. Sady reálných úloh jsou odvozeny od řídící jednotky CNC stroje [13] a inerciálního navigačního systému (INS) [2]. Sady náhodných úloh se skládají z pěti úloh a mají vytížení 20%, 40%, 60% a 80%. Sady náhodných úlohτ_i mají $T_i = D_i$. Perioda T_i úloh byla v rozmezí od 20 do 500 tiků a doba potřebná k vykonání úlohy C_i v rozmezí od 1 do 100 tiků. Pro generování doby potřebné k běhu bylo používáno uniformní rozložení, důvody viz. 7.1.3. Další informace o sadách úloh jsou uvedeny v tabulce 8.1. Kritéria pro nastavení náhodných úloh byla zvolena s ohledem na literaturu s obdobným tématem [3, 24, 14, 8]. Kromě již zmíněného nastavení bylo u úloh změnou nastavením nejlepšího potřebného času pro výpočet úlohy(dále jen BCET) pozorováno, jaký vliv má proměnná délka času potřebného pro vykonání úlohy C_i. Dodejme, že celkové vytížení procesoru U pro sadu úloh INS je přibližně 59% a pro CNC 49.5%. U sady úloh CNC ještě upozorněme na fakt, že dvě úlohy mají $D_i < T_i$, pak případné zatížení procesoru CH je 64,5%.

8.1.2 Nastavení měření

Měření probíhalo s popsanými sadami úloh. Každý experiment byl spuštěn po dobu 10 000 tiků v systému. Po uplynutí stanoveného počtu byla získaná data vypsána na obrazovku terminálu. Sledovanou vlastností bylo, kolik tiků procesor strávil v jednotlivých frekvencích. Měření bylo vcelku zdlouhavé, jelikož při jakékoliv změně musel být celý kód opětovně nahrán do vývojové desky. Z tohoto důvodu nemohlo testování probíhat automatizovaně.

úloha	WCET	D	T
1	1	3	3
2	4	40	40
3	10	625	625
4	20	1000	1000
5	100	1000	1000
6	25	1250	1250

a)

úloha	WCET	D	T
1	4	240	240
2	5	240	240
3	18	480	480
4	72	480	480
5	16	240	240
6	17	240	240
7	57	400	960
8	57	400	780

b)

Obrázek 8.1: Informace o sadách úloh: a) INS, b) CNC

Systém byl nastaven, aby bylo vykonáno 100 tiků za vteřinu. Provedení testu pak v závislosti na době kompilace trvalo přibližně 3 - 4 minuty.

8.1.3 Předpoklady pro interpretaci výsledků

Jelikož nebylo možné změřit pro jednotlivé frekvence změny v odběru, jsou použity následující předpoklady a postup vyhodnocení výsledků uvedený v [24] a [8]:

- Jedná se o hard RT systém.
- Během jednoho tiku je dle zvolené frekvence odebíráno konstantní množství energie.
- Množství odebrané energie E je vypočteno jako druhá mocnina napájecího napětí V: $E \propto V^2$
- Ve výpočtech se počítá jen s energií odebranou procesorem.

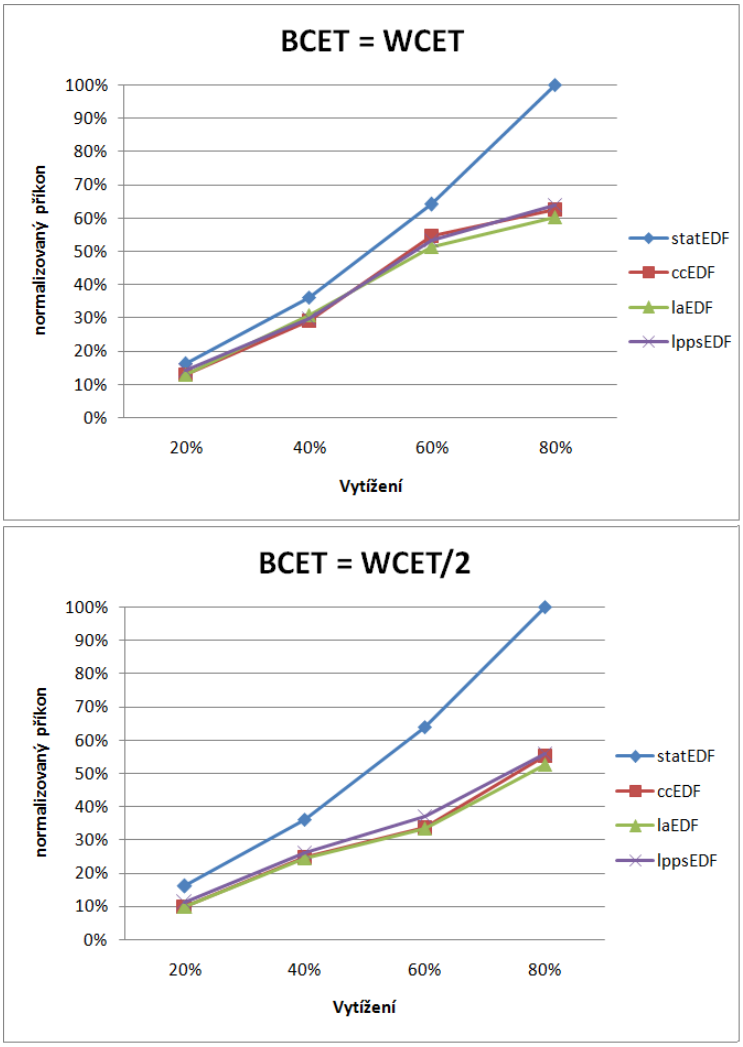
Díky tomuto zjednodušení potřebujeme měřit pouze počet tiků. Dalším zjednodušením je, že se v systému vyskytují pouze běžící úlohy či IDLE úloha. Nepředpokládáme tedy žádný čas nutný k přepnutí úloh či změně frekvence. Tento čas je součástí vykonávání úloh.

V implementační části 7.2.5 uvádím, že systém může být přepnutý do 4 různých frekvencí, ve kterých je prováděn výpočet. Dále ještě uvažujeme IDLE stav, při kterém je odebíráno ještě méně energie. Ve výpočtech pro možné stavy (100%, 75%, 50%, 25% a IDLE) počítáme s napájecím napětím (5, 4, 3, 2 a 0,5).

8.1.4 Výsledky

Získaná data z měření jsou prezentována pomocí grafů. V grafech 8.2 a 8.3 je znázorněno, jaké množství energie bylo spotřebováno pro dané sady úloh. V grafech je k znázornění použit normalizovaný příkon, který je vypočten jako poměr mezi příkonem procesoru při využití DVS algoritmů a příkonem bez DVS algoritmů.

Jak jsem předpokládal, nejhorších výsledků dosahuje algoritmus statEDF, který naplno nevyužívá potenciál DVS. Na druhou stranu toto plánování má zanedbatelné paměťové a výpočetní nároky. Z grafů lze vyčíst, že tento algoritmus již z podstaty jeho návrhu nikterak nereaguje na změny v dobách potřebných k vykonání úloh.



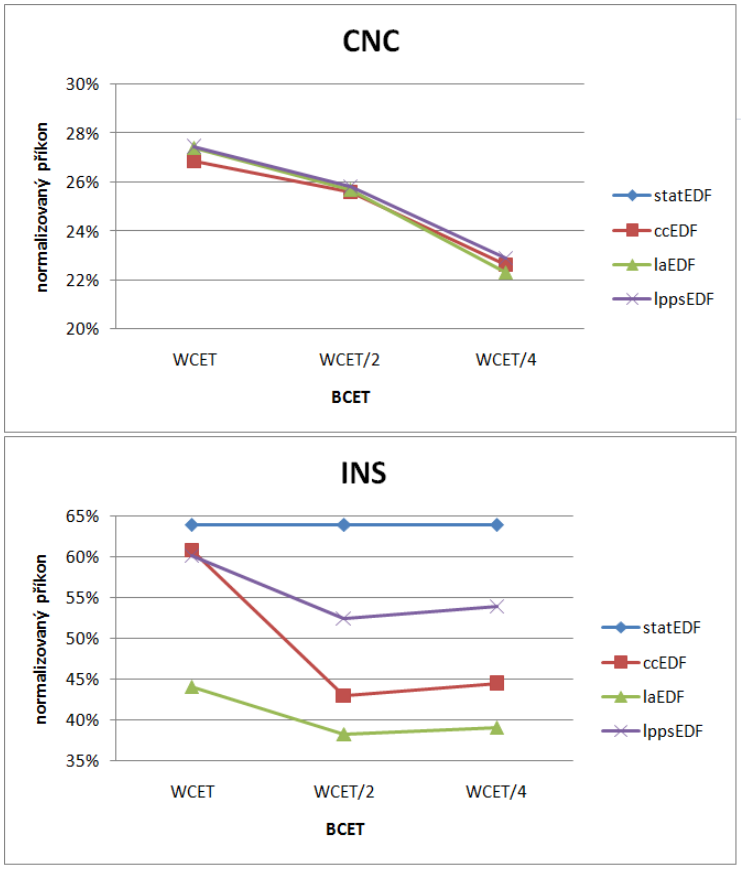
Obrázek 8.2: Porovnání vytížení a normalizovaného příkonu pro sady náhodných úloh

Výsledky dosažené zbylými algoritmy jsou do značné míry podobné, nicméně nejlepších výsledků dosáhl algoritmus laEDF. Opět se jedná o výsledek, který jsem očekával. Očekávání nejlepších výsledků je zapříčiněno podstatou algoritmu, jelikož používá nejprogresivnější mechanismus pro snížení příkonu. Lepší výsledky však mají daň ve formě vyšších paměťových a výpočetní nároků.

Algoritmy ccEDF a lppsEDF dosahují velmi obdobných výsledků, s ohledem na paměťové a výpočetní nároky bych však jako lepší algoritmus považoval ccEDF. Pokud by kritériem pro výběr vhodného algoritmu byl poměr mezi výkonem a výpočetními nároky, je nejlepším algoritmem ccEDF.

Jak jsem uvedl v 8.1.1, sada úloh CNC jako jediná z použitých sad obsahovala i úlohy s $D_i < T_i$. Nicméně tento fakt nikterak významně neovlivnil dosažené výsledky všech algoritmů, ačkoliv všechny algoritmy pracují s vytížením procesoru. Jako případnou možnost ke zlepšení vlastností algoritmů by mohlo být nahrazení ve výpočtu vytížení U za zatížení CH.

Na dalších grafech 8.4 a 8.5 je znázorněno, jak dlouhou dobu systém strávil v jednotlivých stavech pro zvolené sady úloh. Nejprve se zaměřme na algoritmus statEDF. Tento



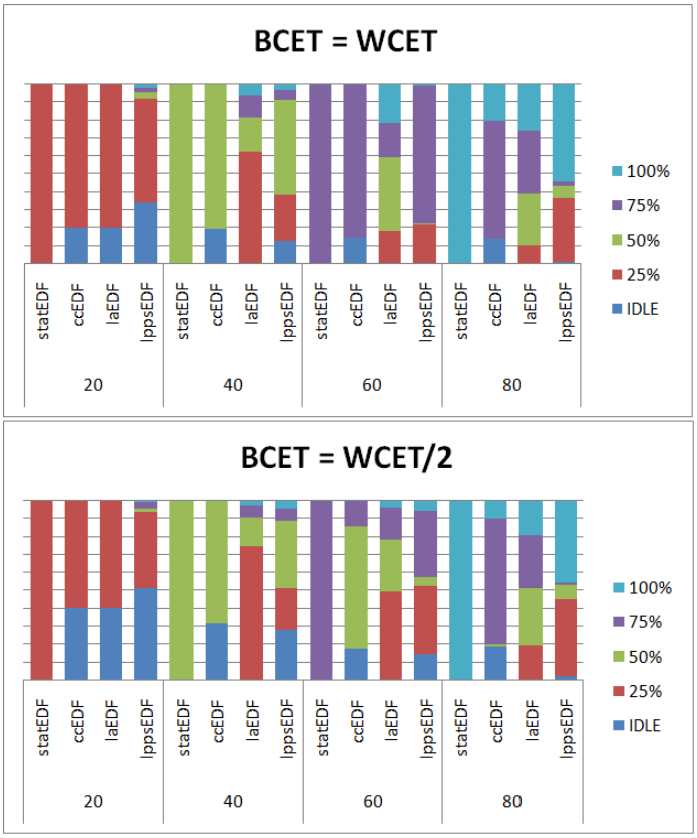
Obrázek 8.3: Porovnání vytížení a normalizovaného příkonu pro sady raálných úloh

algoritmus běží po celou dobu na stejné frekvenci. Jako možné zlepšení výsledků by mohlo být využito přepnutí do IDLE stavu. Tímto krokem by se již sice nejednalo o off-line algoritmus, nicméně by mohlo být dosaženo lepších výsledků. Důležitým faktorem by případně zůstalo rozhodnutí, kdy uspat a kdy nikoliv. Tyto vyšší výpočetní nároky by případně mohly mít nežádoucí vliv na výkon systému. Toto chování nebylo testováno a je uvažováno jako případná možnost pokračování v práci.

Nyní se zaměřme na zbylé tři algoritmy. Na grafech jsou dobře viditelné způsoby, kterými se snaží algoritmy dosáhnout co nejnižší spotřeby energie. Pozorujme, že lppsEDF se snaží strávit mnoho času v nízkých frekvencích, avšak případné výskyty naléhavých situací je třeba kompenzovat přechodem na maximální frekvenci. Na obr. 8.4 je vidět, že i při vytížení pouze 20% stráví lppsEDF jistý čas i v maximální frekvenci. U ccEDF je trend jiný a ze všech algoritmů tráví v maximální frekvenci nejméně času. U algoritmu laEDF lze v chování pozorovat snahu trávit co nejméně času ve stavu IDLE. Toto chování považuji za velikou výhodu. Využití této snahy může být vhodné pro systémy, kde se nenachází žádný IDLE stav, případně pokud je přechod do nějakého obdobného stavu velmi drahou záležitostí.

8.1.5 Zhodnocení výsledků

Jak se prokázalo, lze použitím DVS algoritmů dosáhnout nezanedbatelného snížení spotřeby energie. V porovnání výsledků nejjednoduššího algoritmu statEDF s výsledky nejle-



Obrázek 8.4: Porovnání dob strávených v jednotlivých stavech systému pro sady náhodných úloh

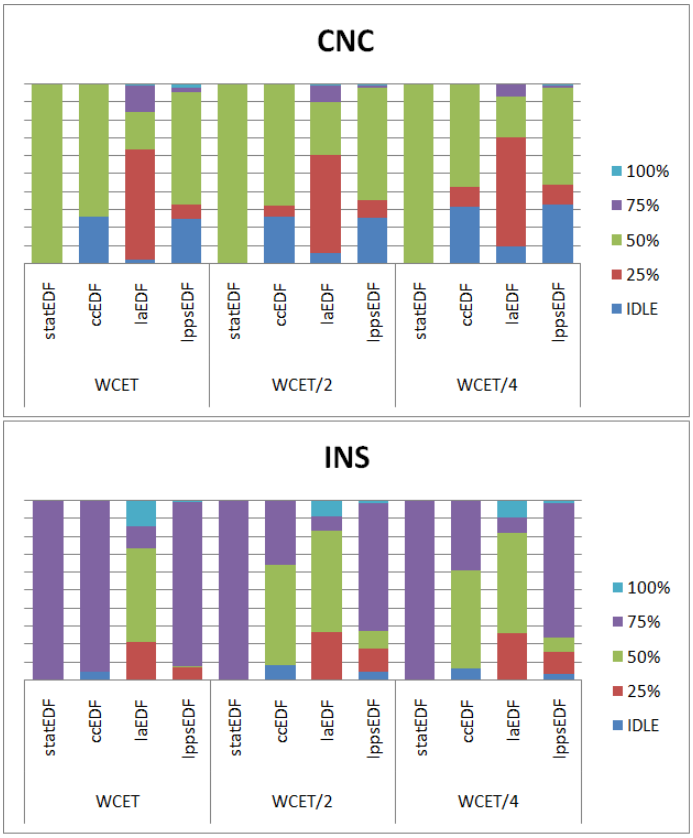
pšího implementovaného algoritmu, lze docílit snížení spotřeby přesahující 40%. Jak jsem již naznačil dříve, výkonnost algoritmů je do jisté míry svázána s jejich výpočetními nároky. Pokud by kritériem vyhodnocení bylo snížení příkonu systému, jeví se jako nejlepší algoritmus laEDF. Vezmeme-li v úvahu ještě výpočetní nároky a jako hodnotící kritérium poměr mezi dosaženými výsledky a těmito nároky, nejlepším algoritmem by byl ccEDF.

8.2 Algoritmy pro zvládání přetížení

Pro účely experimentů jsem implementoval čtyři algoritmy, více viz. 7.3. Jako porovnávací kritérium algoritmů byl zvolen poměr mezi počtem dokončených a spuštěných úloh a poměr mezi hodnotou dokončených a spuštěných úloh.

8.2.1 Sady úloh

Pro účely experimentů byly použity smíšené sady úloh. Sady byly složeny z 5 náhodných periodických úloh a 2, resp. 4 náhodných aperiodických úloh. Celkové vytížení periodických úloh bylo 80%, 90%, 100%, 110% a 120%. K těmto sadám úloh pak byly dále spuštěny 2, resp. 4 aperiodické úlohy. Zatížení aperiodických úloh CH_i pak bylo v rozmezí od 30 do 50%. Příchod aperiodických úloh byl generován pomocí uniformního rozložení v rozsahu od 20 do 200 tiků, D_i od 10 do 25 tiku a C_i od 5 do 15 tiků. Sady byly tvořeny s ohledem na



Obrázek 8.5: Porovnání dob strávených v jednotlivých stavech systému pro sady reálných úloh

literaturu [3, 7, 18], rozdíl byl udělán v rozložení generovaného příchodu úloh. Autoři používají Poissonovo rozložení, já jsem z důvodu výpočetních nároku použil pouze uniformní. Příchodem aperiodických úloh je docíleno dočasného zvýšení vytížení procesoru U, čímž je způsobeno přetížení sytému.

8.2.2 Nastavení měření

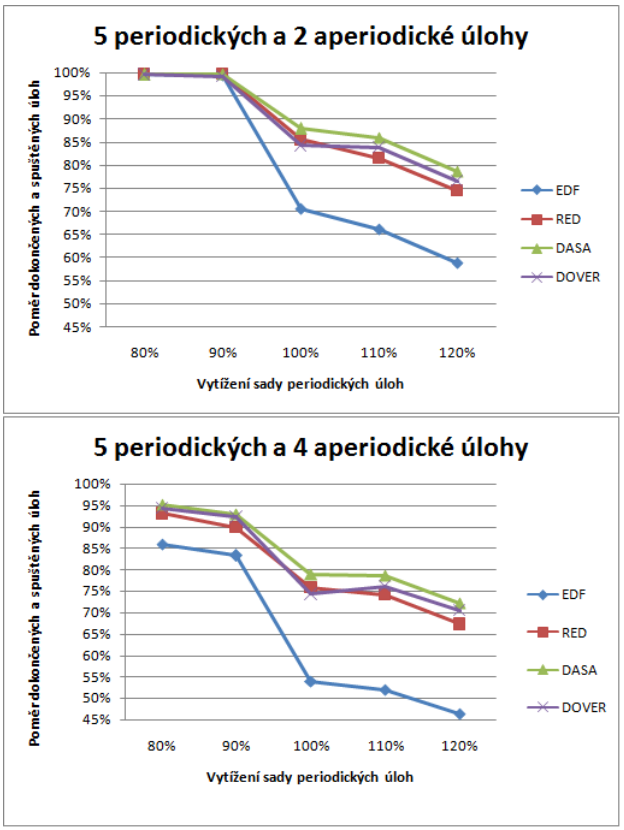
Měření a nastavení probíhalo obdobným způsobem jako u algoritmů pro nízkou spotřebu 8.1.2, tzn. doba měření byla 10 000 tiků a sytém prováděl 100 tiků za vteřinu. Různé byly jen sledované vlastnosti, a to počty spuštěných a dokončených úloh a suma hodnot dokončených úloh a spuštěných úloh.

8.2.3 Předpoklady pro interpretaci výsledků

Předpoklad kladený na systém je, že čas strávený v plánování je součásti času vykonávání úlohy. V systému tedy běží pouze spuštěné úlohy či IDLE úloha. U všech periodických úloh τ_i bylo $T_i = D_i$. U aperiodických úloh byl příchod generován dle uniformního rozložení. Hodnota zisku při dokončení úlohy byla v rozsahu od 0 do 30. Dále předpokládáme, že se jedná o hard RT systém.

8.2.4 Výsledky

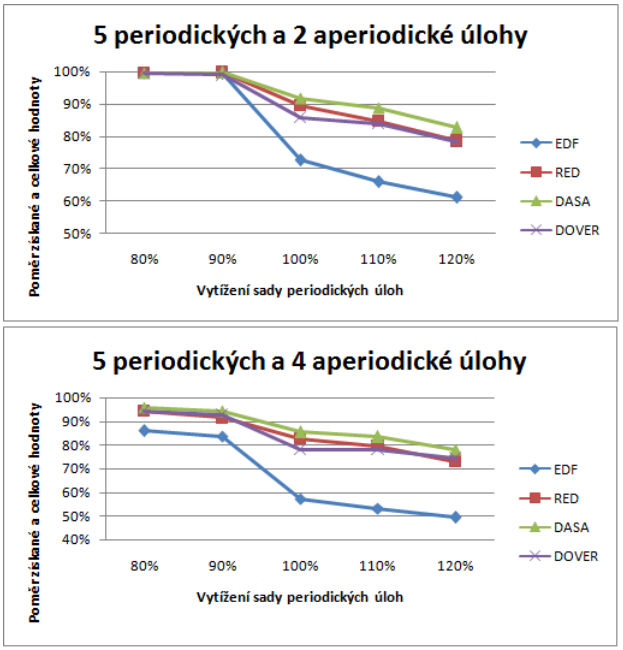
Experimenty na algoritmech byly prováděny s popsanými sadami úloh. Na grafech 8.6 je znázorněn poměr mezi počtem dokončených a spuštěných úloh pro dané sady úloh. Jednoznačně nejhorších výsledků dosáhl algoritmus EDF. Není to neočekávaný jev, jelikož EDF nepatří do algoritmů při přetížení. Co se týče počtu dokončených úloh, tak si nejlépe vedl algoritmus DASA. Algoritmus DASA dosáhl až o 4,5% lepších výsledků vůči zbylým algoritmům RED a D^{over}.



Obrázek 8.6: Poměr dokončených úloh pro dané vytížení periodických úloh

Algoritmus RED při nižším celkovém vytížení systému dosahoval lepších výsledků jak D^{over}, avšak s rostoucím vytížením tento trend opadal a pak lepší výsledky podával algoritmus DOVER. Podobné výsledky v porovnání těchto algoritmů byly zjištěný i v [3, str. 313-315].

Druhou sledovanou vlastností algoritmů byl celkový zisk hodnot dokončených úloh. Poměr hodnoty dokončených a puštěných úloh je znázorněn v 8.7. V tomto kritériu dosáhl opět nejlepších výsledků algoritmus DASA. Trend je podobný jako u předešlého kritéria. Rozdíl procentuálního zisku u DASA a RED, resp. D^{over} dosahuje opět až 5%, resp. 8%. I zde lze vypozorovat zlepšování výsledků algoritmu D^{over} vůči RED algoritmu, ačkoliv se nejedná o tak viditelný rys.



Obrázek 8.7: Poměr získaných hodnot úloh pro dané vytížení periodických úloh

8.2.5 Zhodnocení výsledků

Celkově nejlepších výsledků dosáhl algoritmus DASA, který dosáhl nejlepších výsledků v obou sledovaných kritériích. Zlepšení vůči zbylým testovaným algoritmům dosahuje až 8%. Který algoritmus se umístil na druhém místě, nelze jednoznačně určit. Pro systémy s nižším celkovým vytížením dosahoval lepších výsledků algoritmus RED, avšak s rostoucím vytížením prokazoval lepších výsledků D^{over} algoritmus. Nicméně v porovnání s obyčejným algoritmem EDF dosahovaly algoritmy pro zvládání přetížení až o 25% lepší výsledky v počtu dokončených úloh, resp. o 28 % lepší výsledky v celkově získané hodnotě systému.

Kapitola 9

Závěr

Oblasti plánování úloh při přetížení a při nedostatku energetických zdrojů byly podrobně probrány. Pro obě témata jsou uvedeny jejich příčiny, možná východiska a postupy pro vypořádání se s těmito situacemi. Pro každou oblast bylo také vybráno několik algoritmů, které jsou podrobněji rozebrány.

Pro řešení diplomové práce byla zvolena vývojová deska demoQE128 a real-time operační systém μC/OS-II. Byl vytvořen modul, který umožňuje do jádra přidat 8 plánovacích algoritmů. Jedná se o jeden obecný algoritmus (EDF), tři algoritmy pro přetížení(DASA, D^{over}, RED) a čtyři algoritmy pro plánování při nízkém příkonu (statEDF, ccEDF, laEDF, lppsEDF). Kromě těchto algoritmů byla implementována ještě pomocná funkce pro logování průběhu algoritmů.

Experimenty byly provedeny na uvedené platformě s reálnými i náhodnými sadami úloh. Jelikož neposkytuje deska DVS, bylo vyhodnocení algoritmů pro nízký příkon provedeno dle stanovených předpokladů. Jednotlivé algoritmy byly mezi sebou dle naměřených výsledků porovnány a dále byly zjištěny jejich výkonnostní rysy.

Cílem diplomové práce bylo nastudovat literaturu ohledně jednotlivých oblastí plánování, popsat tyto mechanismy, vybrat několik algoritmů, implementovat zvolené algoritmy a porovnat jejich vlastnosti při spuštění na reálném hardware. Jediným neúplným bodem se stalo testování na reálném hardware, jelikož deska demoQE128 neposkytuje DVS. Ostatní cíle diplomové práce se však podařilo naplnit.

Jako možnosti pokračování práce by bylo vhodné použít vytvořený modul a odzkoušet algoritmy na platformě, která umožňuje DVS. Pro toto řešení změřit příkon a porovnat výsledky měření s teoretickými výsledky. Dále by bylo možné do modulu implementovat další plánovací mechanismy a také porovnat jejich výsledky. Pokud by případná platforma s DVS měla výkonnější procesor, bylo by vhodné využít pro generování čísel i další rozložení, např. Poissonovo či exponenciální. Další možným směrem v pokračování by mohlo být začlenění časů nutných pro přepnutí úlohy, změny frekvence a přeplánování do měření.

Literatura

[1] Aydi, H.; Mejía-Alvarez, P.; Mossé, D.; aj.: Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, 2001, s. 95–105, iSBN 0-7695-1420-0.

[2] Burns, A.; Tindell, K.; Wellings, A.: Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Trans. Softw. Eng.*, 1995: s. 475–480, iSSN 0098-5589.

[3] Buttazzo, G. C.: *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications, Third Edition*. Springer Publishing Company, Incorporated, 2011, 536 s., iSBN 978-1-4614-0675-4.

[4] Chen, W.-K.: *The VLSI Handbook, Second Edition*. CRC Press, 2006, 2320 s., iSBN 0-8493-4199-X.

[5] Cheng, A. M. K.: *Real-Time systems: Scheduling, Analysis and Verification*. John Wiley & Sons Inc., 2002, 552 s., iSBN 0-471-18406-3.

[6] Cottet, F.; Delacroix, J.; Kaiser, C.; aj.: *Scheduling in Real-Time Systems*. John Wiley & Sons Inc., 2002, 284 s., iSBN 0-470-84766-2.

[7] Davis, R.: Integrating Best-Effort Policies into Hard Real-Time Systems based on Fixed Priority Pre-emptive Scheduling. *DEPT. COMPUTER SCIENCE, UNIVERSITY OF YORK*, ročník 240, 1994.

[8] Dudani, A.; Mueller, F.; Zhu, Y.: Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints. *SIGPLAN Not.*, 2002: s. 213–222, iSSN 0362-1340.

[9] Freescale: DemoQE128 Demonstration Board [on-line]. dostupné z http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=DEMOQE128, [cit. 2012-17-05].

[10] Freescale: MCF51QE: Flexis 32-bit ColdFire V1 Microcontroller [on-line]. dostupné z http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MCF51QE, [cit. 2012-17-05].

[11] Freescale: S08QE: 8-bit Flexis Low-Power QE Microcontroller [on-line]. dostupné z http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=S08QE, [cit. 2012-17-05].

[12] Gruian, F.: *Energy-Centric Scheduling for Real-Time Systems*. Lund University, 2002, 176 s., iSBN 91-628-5494-1.

[13] Kim, N.; Ryu, M.; Hongt, S.; aj.: Visual assessment of a real-time system design: a case study on a cnc controller. In *in Proc. IEEE Real-Time Systems Symposium*, IEEE Computer Society Press, 1996, s. 300–310.

[14] Kim, W.; Shin, D.; Yun, H.-S.; aj.: Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, IEEE Computer Society Press, 2002, s. 219–228, iSBN 0-7695-1739-0.

[15] Kučera, P.: Výukové materiály předmětu Operační systémy reálného času. 2007.

[16] Labrosse, J. J.: *Microc/OS-II, The Real Time Kernel, Second Edition*. R & D Books, 1998, 648 s., iSBN 0-87930-543-6.

[17] Lee, I.; Leung, J. Y.-T.; Son, S. H.: *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 2007, 800 s., iSBN 1-58488-678-1.

[18] Li, P.; Ravindran, B.: Fast, Best-Effort Real-Time Scheduling Algorithms. *IEEE Trans. Comput.*, 2004: s. 1159–1175, iSSN 0018-9340.

[19] Li, Q.; Yao, C.: *Real-time concepts for embedded systems*. CMP Books, 2003, 294 s., iSBN 1-57820-124-1.

[20] Liu, J.: *Real-Time systems*. Prentice Hall, 2000, 592 s., iSBN 0-13-099651-3.

[21] Locke, C. D.: Best-effort decision-making for real-time scheduling [on-line]. 1986 [cit. 2012-17-05], dostupné z http://douglocke.com/Downloads/BE.pdf.

[22] Micrum: MicroC/OS-II Microprocessor Ports [on-line]. dostupné z http://micrium.com/page/downloads/ports/freescale/8-16_bits, [cit. 2012-17-05].

[23] Mohammadi, A.; Selim, G. A.: Scheduling Algorithms for Real-Time Systems [on-line]. 2005-07-15 [cit. 2012-17-05], dostupné z http://research.cs.queensu.ca/home/akl/techreports/scheduling.pdf.

[24] Pillai, P.; Shin, K. G.: Real-time dynamic voltage scaling for low-power embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 2001: s. 89–102, iSSN 0163-5980.

[25] Shin, Y.; Choi, K.: Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, ACM, 1999, s. 134–139, iSBN 1-58113-092-9.

[26] Strnadel, J.: Real-time operační systémy (ROS) - Studijní opora [on-line]. 2006-10-10 [cit. 2012-17-05].

[27] WWW stránka pana Mike Holenderskiho: Vizualizační aplikace Grasp [on-line]. dostupné z http://www.win.tue.nl/~mholende/grasp/, [cit. 2012-17-05].

[28] Škarvada, J.: *Optimalizace aplikace testu číslicových systémů pro nízký příkon, Dizertační práce*. FIT VUT v Brně, 2009, 125 s.

Příloha A

Obsah CD

- doc
  - tex
    - Text diplomové práce ve formátu LaTeX
    - Obrázky použité v diplomové práci
  - Text diplomové práce ve formátu PDF
- src
  - example - ukázkový projekt aplikace Codewarrior
  - modul - soubory s kódem nového modulu
- etc
  - log - soubor s logy algoritmů
  - grasp - aplikace Grasp
  - articles - pdf soubory článků použitých v diplomové práci
  - excel - soubory formátu MS Excel s daty o měření a metrikách
  - virtualCOM - aplikace a návod pro přemostění komunikace z virtuálního COM

Příloha B

Metriky kódu

Nově implementované funkce modulu se nacházejí ve 3 souborech. Hlavním souborem je `ex_task_sched.c` a jeho hlavičkový soubor `ex_task_sched.h`. V těchto souborech jsou kromě funkcí také definovány globální proměnné a nové struktury. Celkem se v souboru `ex_task_sched.c` nachází 39 nových funkcí. Další 3 nové funkce se pak nalézají v souboru `app.c`. Tyto 3 funkce však nejsou nezbytně nutné pro správnou funkci nového modulu. Velikost přeloženého jádra s 5 úlohama se zapnutým modulem, případně bez něj, je znázorněno v B.1. V B.2 jsou uvedeny paměťové nároky globálních proměnných a rozšiřujících TCB bloků. Paměťové nároky jednotlivých funkcí jsou uvedeny v B.3. Tabulka B.4 ilustruje paměťové nároky jednotlivých algoritmů pro sadu pěti úloh s vypnutým logováním. Poznamenejme, že se jedná pouze o dodatečné paměťové nároky a nejsou zde započteny paměťové nároky samotného jádra μC/OS-II.

Velikost přeloženého systému v Bytech	app	modul	jádro	knihovny	celkem
původní jádro bez zapnutých modulů	1540	0	2402	2226	6168
zapnutý modul ex_task_sched bez logování	1665	74	2402	2226	6367
zapnutý modul ex_task_sched s logováním	1665	973	2402	2226	7266

Obrázek B.1: Velikost přeloženého jádra s 5 úlohami

globální proměnné	počet Bytů
EventInfo	13
DASA_LOAD	6
EX_TCB	15
ostatní	13
vypis statistik měření	36
eventLog[MAX_EVENT]	2+MAX_EVENT*EventInfo
valDenArr[COMM_NUM_TASKS]	2+COMM_NUM_TASKS*DASA_LOAD
CommTaskExTCB[COMM_NUM_TASKS];	2+COMM_NUM_TASKS*EX_TCB

Obrázek B.2: Paměťové nároky globálních proměných modulu ex_task_sched

Paměťové nároky funkcí	Parametry funkce + lokální proměnné
InitLoging()	0 + 0
LogTaskEvent()	7 + 3
LogTaskOne()	1 + 0
LogTaskPrint()	0 + 0
GenerateUniform()	6 + 0
GenNextArrive()	0 + 1
GenNextComputatio()	0 + 1
SetSchedMethod()	1 + 0
InitTaskAttributes()	13 + 0
OS_EX_TimeDlyResume()	2 + 1
OS_EX_TimeDly()	2 + 1
ExSchedTickHook ()	0 + 7
ExSched()	4 + 2
EDF_sched()	1 + 8
QuickSort()	2 + 10
ComputeLoad()	4 + 16
DASAComputeLoad()	5 + 18
LoadValueDensity()	1 + 3
DASA_EDF()	1 + 4
UnsetPrivTask()	1 + 2
DASA()	1 + 8

Paměťové nároky funkcí	Parametry funkce + lokální proměnné
sqrt()	4 + 4
DOVERComputeCond()	10 + 0
DOVERAbandonTask()	5 + 1
DOVERLoad()	5 + 3
DOVER()	5 + 17
REDLoad()	2 + 6
REDComputeLaxity()	4 + 6
REDRejection()	8 + 4
RED()	5 + 8
FLL_Change()	1 + 0
SetFreq()	0 + 4
staticEDF()	1 + 0
ccEDF()	1 + 0
lookAheadLoad()	3 + 4
lookAheadFreq()	0 + 19
laEDF()	1 + 0
lppsEDF()	1 + 6
Init_r()	0 + 2
GenerateTasks()	0 + 1
AppCreateTasks()	0 + 0
CommTask()	1 + 11

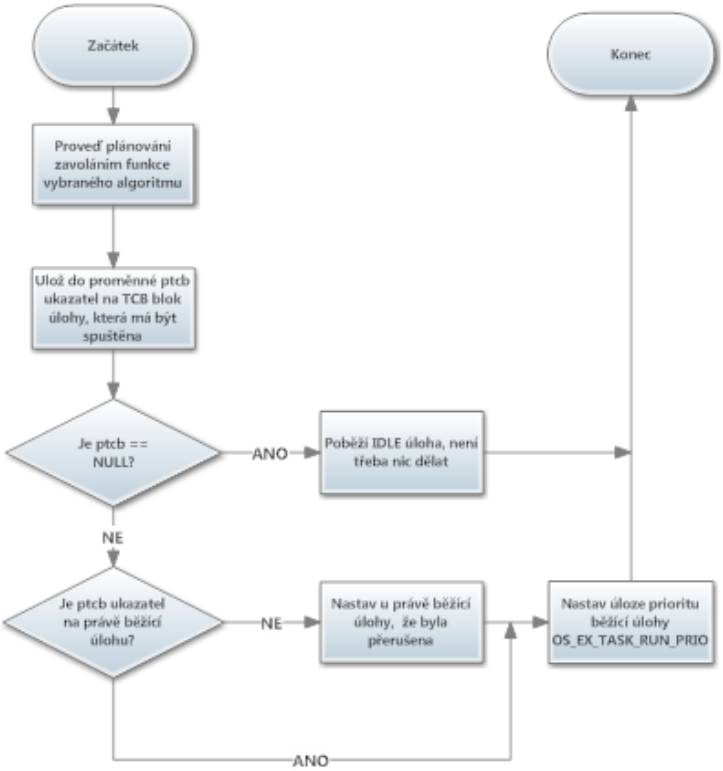
Obrázek B.3: Paměťové nároky funkcí implementovaných v modulu ex_task_sched

Paměťové nároky algoritmu pro pět úloh v systému, logování vypnuto	
algoritmus	potřebná paměť v Bytech
EDF	188
DASA	276
DOVER	285
RED	285
statEDF	196
ccEDF	196
laEDF	262
lppsEDF	230

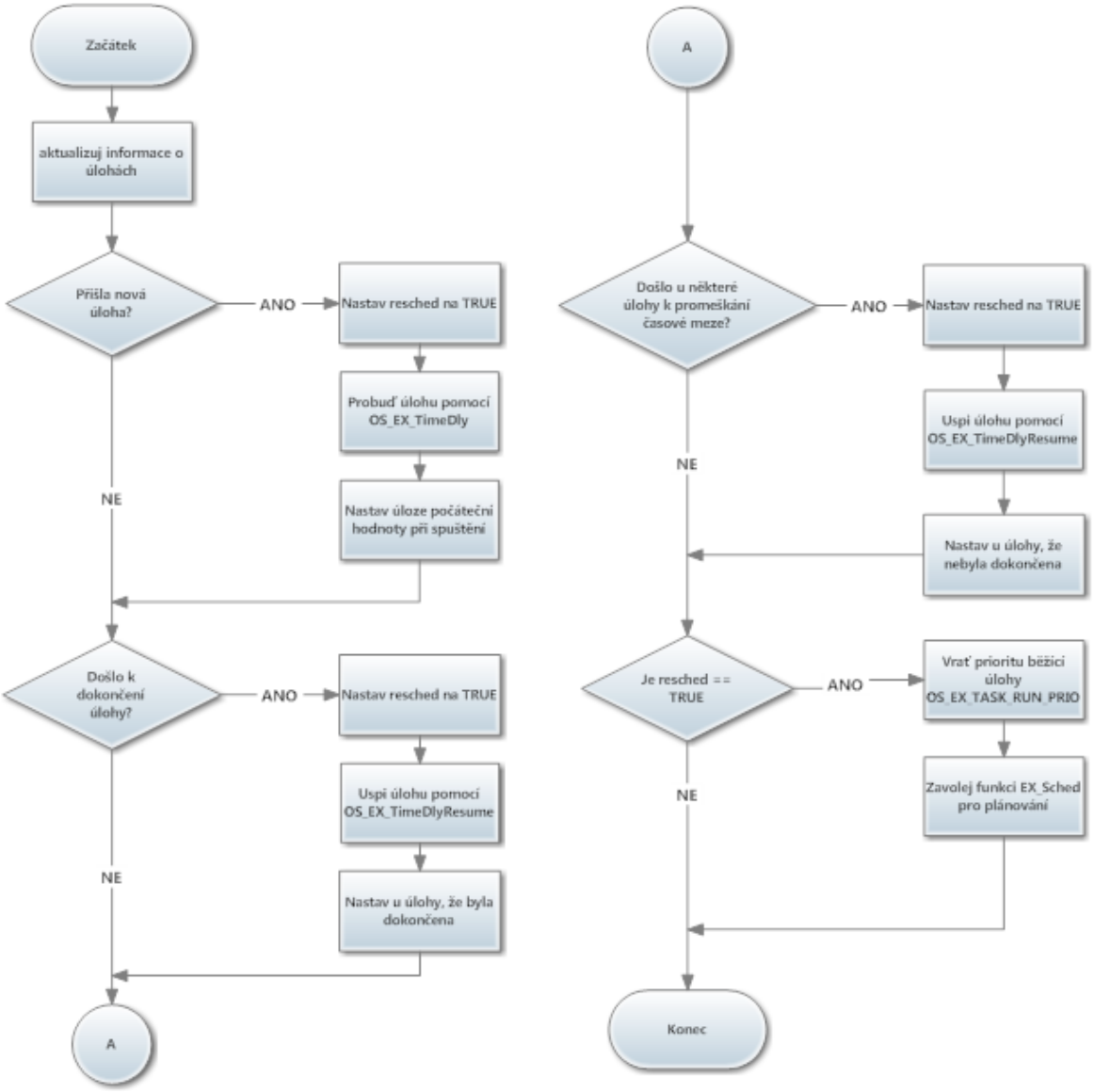
Obrázek B.4: Srovnání paměťových nároků algoritmů v modulu ex_task_sched

Příloha C

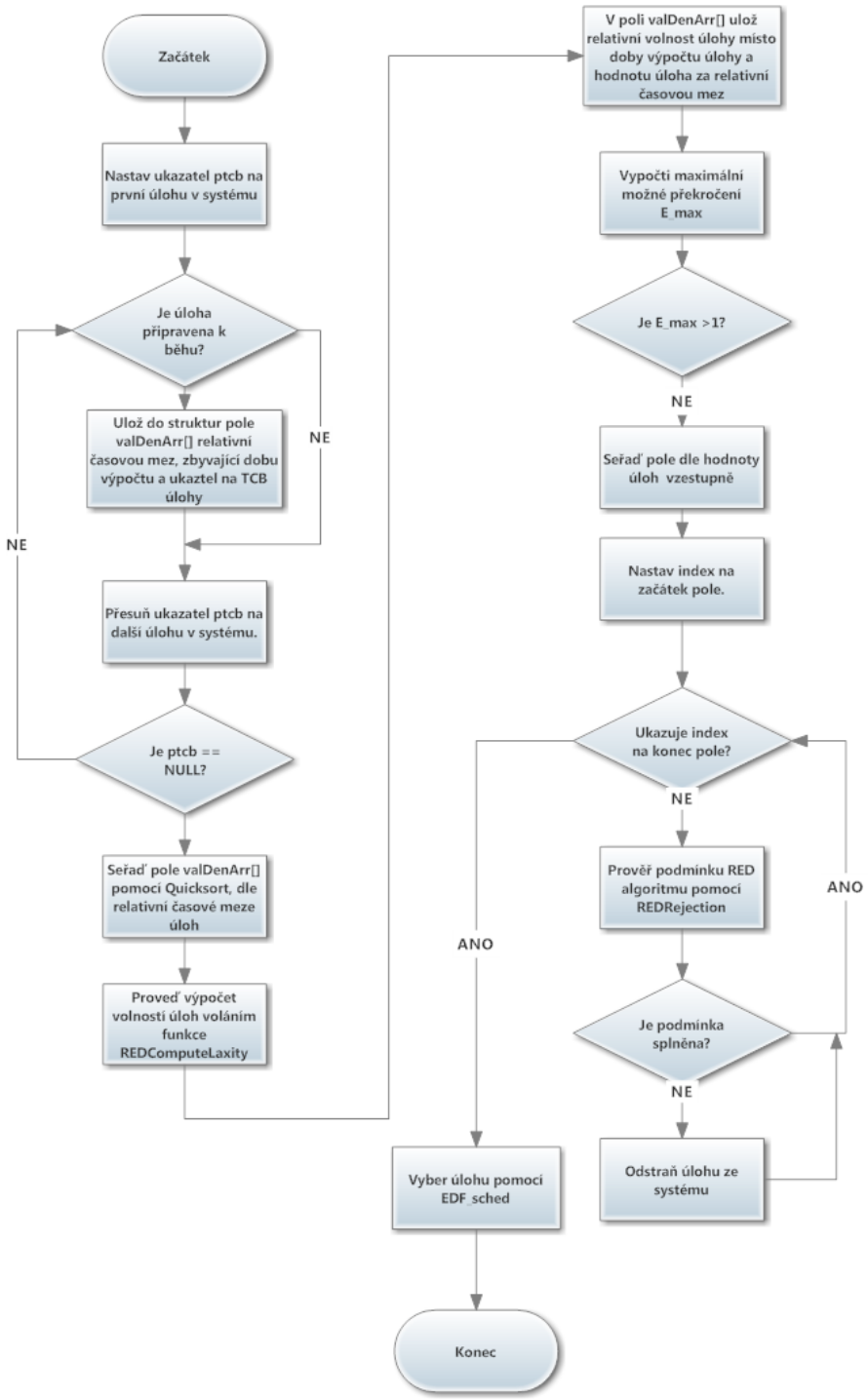
Diagramy implementace složitějších funkcí



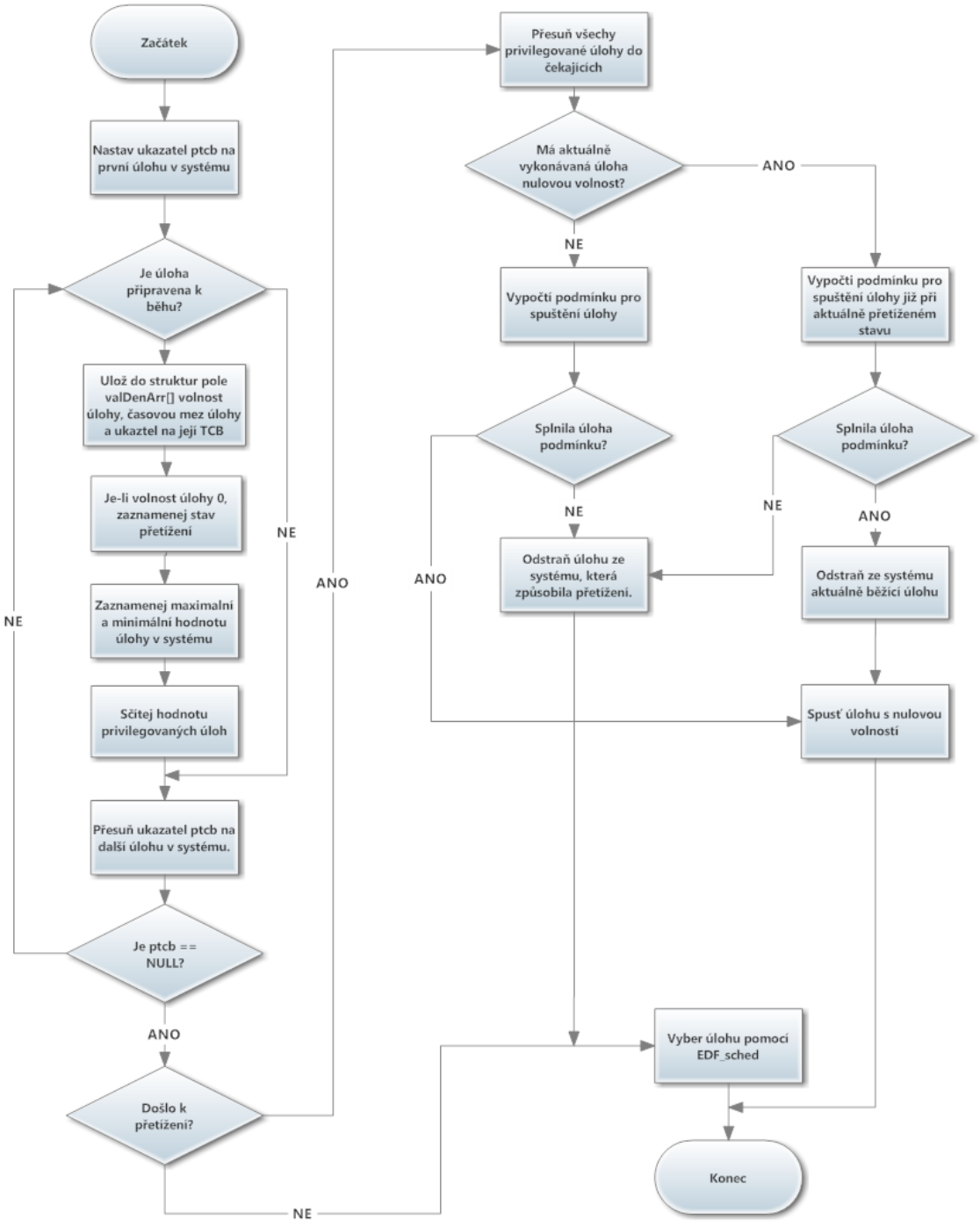
Obrázek C.1: Diagram průběhu ExSched



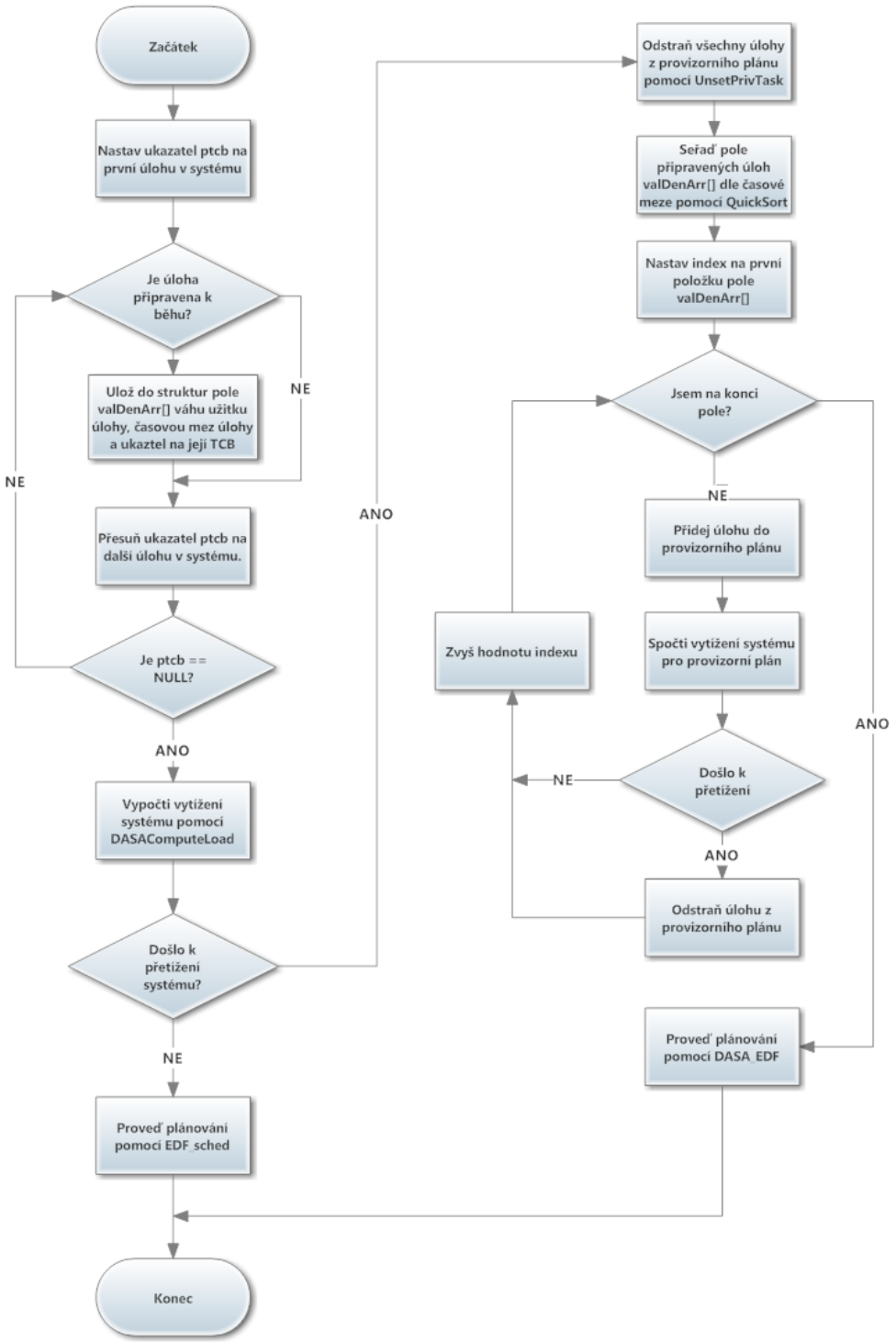
Obrázek C.2: Diagram průběhu TickHook



Obrázek C.3: Diagram průběhu RED algoritmu



Obrázek C.4: Diagram průběhu D^{over} algoritmu



Obrázek C.5: Diagram průběhu algoritmu DASA

Příloha D

Vizualizace logů

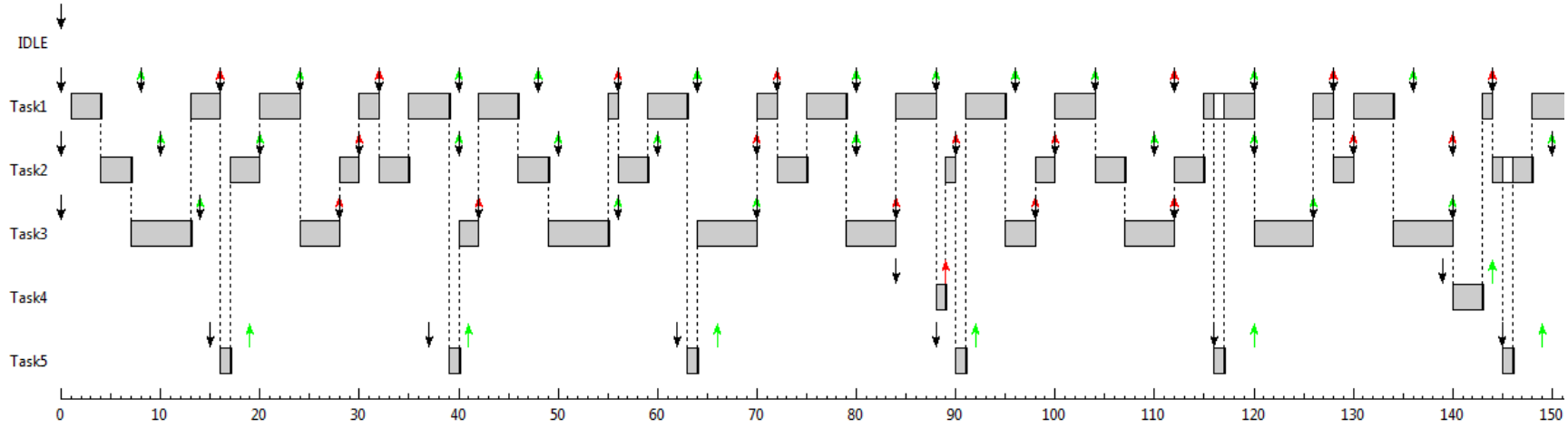
Algoritmy pro zvládání přetížení

Níže jsou uvedeny vizualizace logů pro sadu 5 úloh. Sada je složena ze 3 periodických a 2 aperiodických úloh. Červené šipky časových mezí znamenají, že úloha nebyla dokončena. Zelené zase označují, že úloha byla dokončena v rámci časových mezí. Na vizualizacích je vidět, že každý algoritmus pracuje odlišným způsobem. Poznamenejme, že u algoritmu EDF D.1 došlo v čase 28 k domino efektu. Na ostatních vizualizacích pak lze sledovat, jak se s tímto jevem vypořádaly ostatní algoritmy.

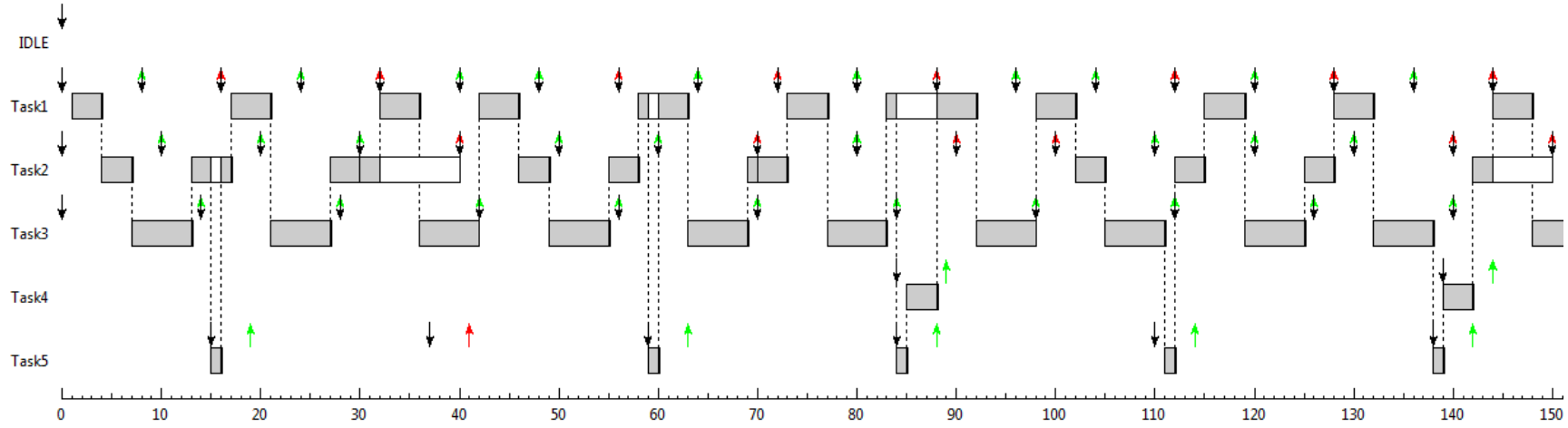
Algoritmy pro nízký příkon

Pro ukázku vizualizace algoritmů pro nízký příkon je použita sada 3 periodických úloh. Všimněme si, že zde se v obrázcích nachází pouze zelené šipky časových mezí. Novým prvkem jsou intervalové šipky s čísly nacházející se nad úlohou IDLE. Jedná se o informaci, na jaké frekvenci procesoru byla daná úloha provedena. Informace o frekvenci je uvedena v procentech maximální frekvence.

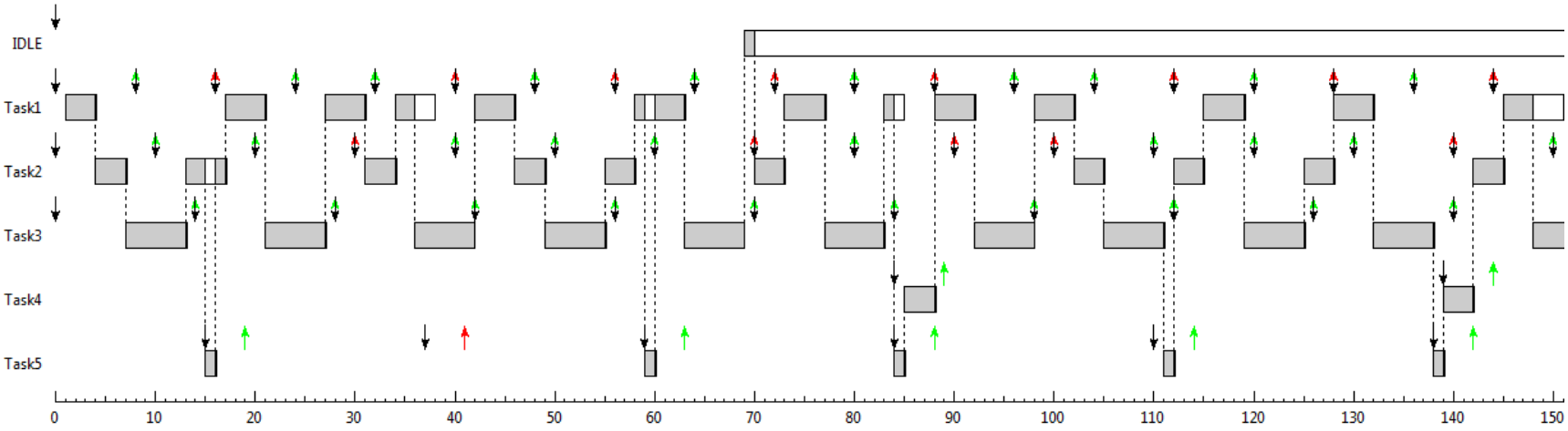
Pozorný čtenář si jistě všimne, že při vykonávání úlohy IDLE se nepřechází do IDLE stavu. Vzhledem k tomu, že je znázorněna úloha IDLE, ačkoliv by byl způsoben přechod do IDLE stavu, není tato informace o přechodu přidávána do logu. Toto chování je také způsobeno tím, že vývojová deska poskytuje pouze 4 možné frekvence, které byly využity k simulaci DVS. Předpokládejme tedy, že přechod do IDLE stavu nastal, i když není znázorněn.



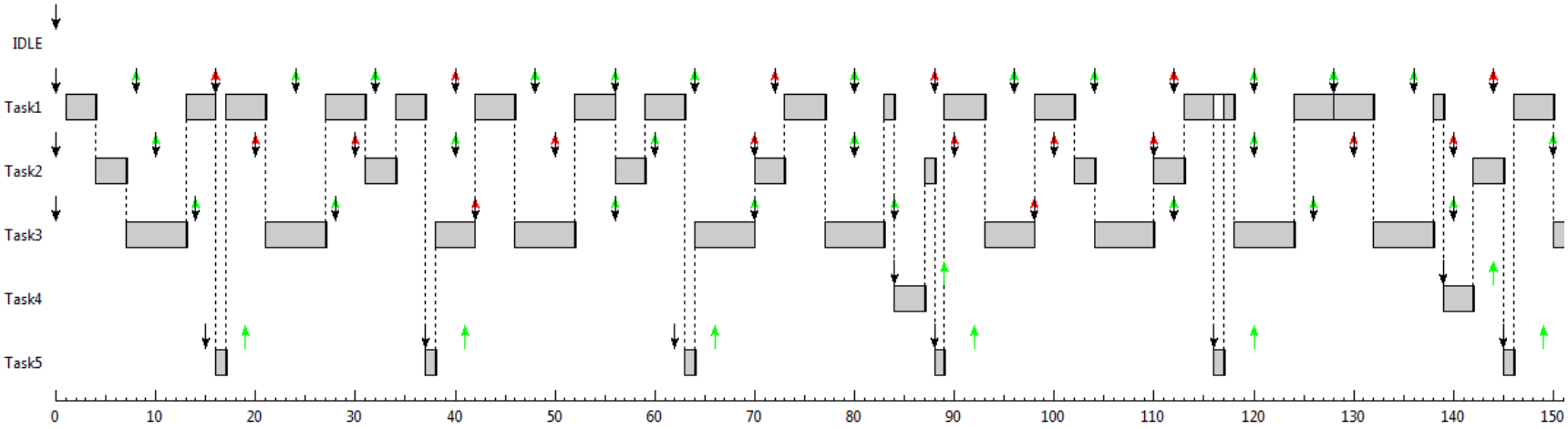
Obrázek D.1: Vizualizace algoritmu EDF pro ukázkovou sadu úloh



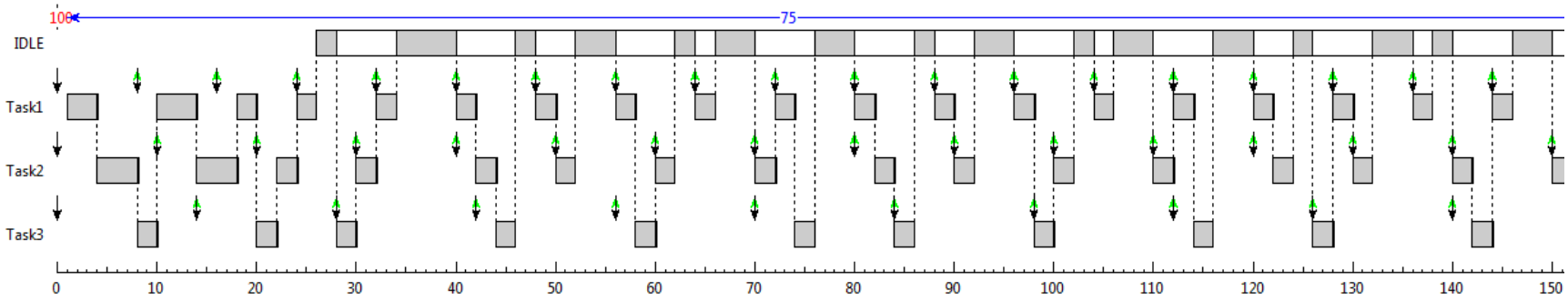
Obrázek D.2: Vizualizace algoritmu DASA pro ukázkovou sadu úloh



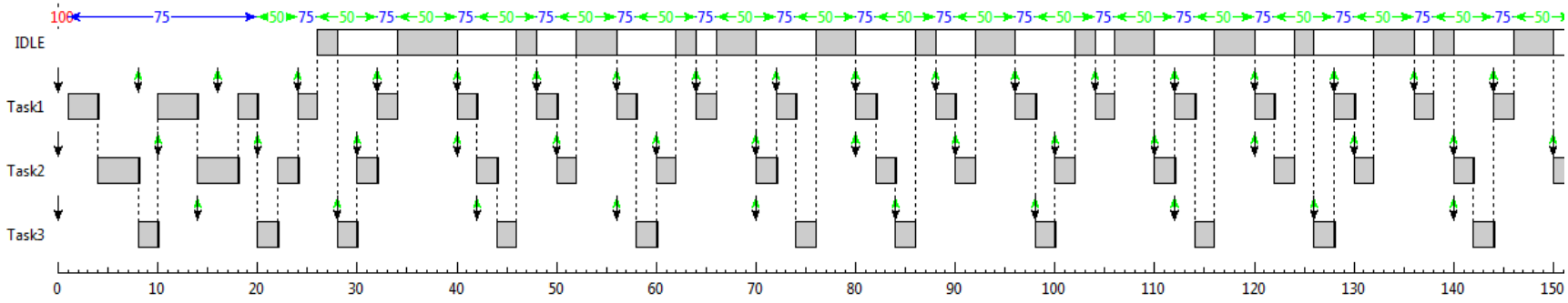
Obrázek D.3: Vizualizace algoritmu DOVER pro ukázkovou sadu úloh



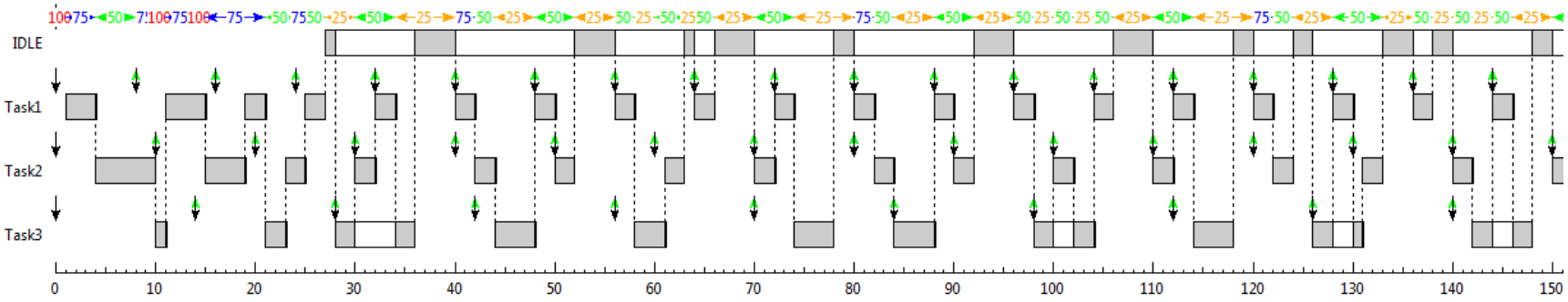
Obrázek D.4: Vizualizace algoritmu RED pro ukázkovou sadu úloh



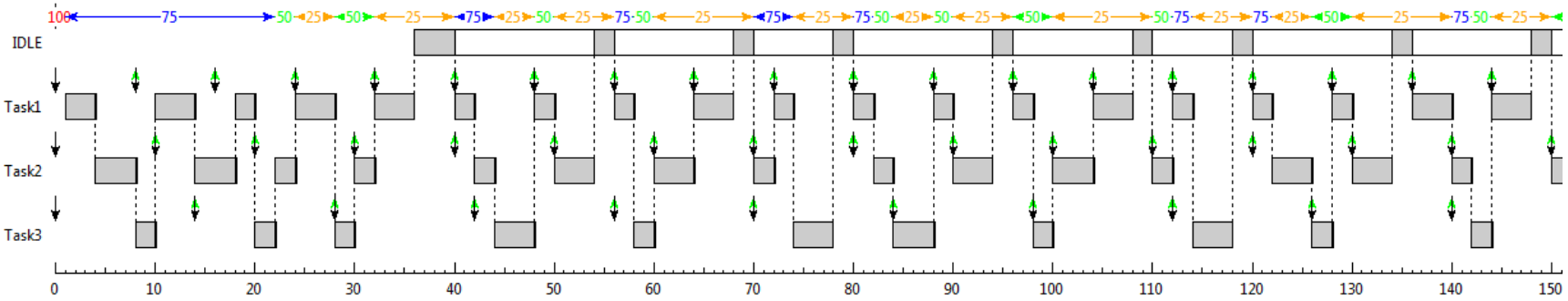
Obrázek D.5: Vizualizace algoritmu statEDF pro ukázkovou sadu úloh



Obrázek D.6: Vizualizace algoritmu ccEDF pro ukázkovou sadu úloh



Obrázek D.7: Vizualizace algoritmu laEDF pro ukázkovou sadu úloh



Obrázek D.8: Vizualizace algoritmu lppsEDF pro ukázkovou sadu úloh

Příloha E

Manuál

V krátkosti zde popíši nutné kroky pro přidání implementovaného modulu do portu jádra μC/OS-II pro vývojovou desku demoQE128. Dále je uveden stručný návod, jak modul používat.

Integrace modulu

Aby mohl být modul integrován je nutné udělat několik změn v původním kódu jádra. Nejprve je nutné do souboru `includes.h` přidat vložení hlavičkového souboru pomocí `#include <ex_task_sched.h>`. Dále je nutné do souboru `app_cfg.h` přidat následující kód.

```
#define OS_EX_TASK_SCHED_EN 1
#define OS_EX_GRASP_LOG_EN 1

#define COMM_NUM_TASKS 5
#define COMM_TASK_STK_SIZE 256

#define OS_EX_TASK_RUN_PRIO 11
#define OS_EX_TASK_FIRST 12
```

Tabulka E.1: Úpravy v souboru `app_cfg.h`

Nejvíce úprav je nutné udělat v souboru `app.c`. Na začátku je třeba přidat prototypy funkcí a vytvořit globální proměnou, jak je uvedeno v E.2.

Tělo funkce `main()` bude obsahovat pouze volání funkcí `OSInit()`, `AppCreateTasks()` a `OSStart()` v tomto pořadí. Dále je nutné vložit kód funkcí `AppCreateTasks()`, `CommTask()` a `GenerateTasks()`. Kód jmenovaných funkcí se nachází na přiloženém digitálním médiu. Do těla funkce `App_TaskIdleHook()` je nutné přidat volání funkce `LogTaskPrint()`. Poslední nutnou změnou je přidání volání `ExSchedSWHook()` do těla funkce `App_TaskSwHook()`. Celý upravený soubor `app.c` je uložen na přiloženém digitálním médiu. Na médiu je uložen i již předpřipravený projekt, který stačí otevřít v aplikaci Codewarrior verze 6.

```
#if OS_EX_TASK_SCHED_EN > 0
static void AppCreateTasks (void);
void CommTask (void *pdata);
void GenerateTasks();
EX_TCB CommTaskExTCB[COMM_NUM_TASKS];
#endif
```

Tabulka E.2: Úpravy v souboru `app.c`

Stručný návod

Pro vytvoření sady úloh a jejich spuštění je třeba udělat pouze několik jednoduchých kroků. Tyto kroky si zde teď popíšeme.

1. Nejprve je nutné v souboru `app_cfg.h` nastavit počet úloh COMM_NUM_TASKS.
2. Dále je nutné v těle `AppCreateTasks()` pomocí funkce `SetSchedMethod(OS_EX_TASK_SCHED_EDF)` zvolit plánovací algoritmus.
3. Posledním krokem je nastavení počátečních informací o úlohach pomocí funkce `InitTaskAttributes()`.

Při tomto postupu jsou úlohy vytvářeny pomocí funkce `GenerateTasks()`. Každé úloze je pak přidělena funkce `CommTask()`. Pokud by někomu nevyhovoval tento postup vytváření úloh, a nebo stejné tělo všech funkcí, může si mezi body 2 a 3 vytvořit úlohy sám. Nicméně do těla funkce `CommTask()` může uživatel případně vložit na určeném místě svůj vlastní kód.

Nastavení informací o úlohách se provádí pomocí funkce `InitTaskAttributes()` jak pro úlohy pro přetížení, tak pro úlohy pro nízký příkon. Rozdíl vložených údajů je znázorněn v E.3.

Pro lepší pochopení doporučuji vyzkoušet připravený příklad nacházející se na digitálním médiu. Aby fungoval výpis informací do terminálu korektně, je třeba nastavit baud rate přenosu na hodnotu 38 400, parita vypnuta a velikost dat 8 bitů.

```
void InitTaskAttributes (*pext, type, r, C, D, P, M, val)

  *pext  ukazatel na rozšířený TCB blok úlohy
  type   typ úlohy - aperiodická / periodická
  r      čas prvního spuštění úlohy
  C      nejdelší nutná doba potřebná k dokončení úlohy (WCET)
  D      časová mez úlohy
  P      perioda úlohy
         pro aperiodické úlohy používá jako hranice nejdelšího příchodu
  M      tolerance časové meze pro RED
         alg. pro nízký příkon - nejmenší nutná doba potřebná k dokončení
         úlohy (BCET)
  val    alg. pro přetížení - hodnota zisku při dokončení úlohy
         alg. pro nízký příkon - přiští doba nutná k dokončení úlohy
```

Tabulka E.3: Úpravy v souboru `app.c`