VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ANALÝZA LATENCE RASPBERRY PI OS S PREEMPT_RT ZÁPLATOU

LATENCY ANALYSIS OF RASPBERRY PI OS PATCHED WITH PREEMPT_RT

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Mitko Mitěv

AUTHOR

VEDOUCÍ PRÁCE — Ing. Lukáš Pohl, Ph.D.

SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Mitko Mitěv ***ID:*** 191958

Ročník: 3 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Analýza latence Raspberry Pi OS s PREEMPT_RT záplatou

POKYNY PRO VYPRACOVÁNÍ:

1. Proveďte analýzu klíčových vlastností PREEMPT_RT záplaty jádra OS Linux.
2. Aplikujte PREEMPT_RT záplatu na Raspberry Pi OS 64bit. Použijte konfiguraci umožňující izolaci jednotlivých jader procesoru.
3. Vytvořte aplikaci reagující na reálný vstupní signál nastavením výstupního GPIO pinu Raspberry PI.
4. Vytvořte seznam zátěžových testů vhodně simulujících reálné vytížení jednotlivých částí Raspberry Pi
5. Proveďte měření odezvy testovací aplikace na vstupní signál v závislosti na použitém plánovači, konfiguraci použitých/izolovaných jader CPU a vytížení systému.

DOPORUČENÁ LITERATURA:

LIBERAL de los RÍOS, Alberto, 2021. Linux Driver Development with Raspberry Pi - Practical Labs. Independently published, ISBN 979-8516120688.

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 23.5.2022

Vedoucí práce: Ing. Lukáš Pohl, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

Abstrakt

Bakalářská práce se věnuje analýze latence Linux jádra na Raspberry Pi OS a zjišťuje jaký vliv na latenci má rozšíření (patch) operačního systému (OS) PREEMPT_RT. Dále popisuje postup při kompilaci jádra Linuxu a jak při kompilaci zavést PREEMPT_RT rozšíření do OS. Práce také poskytuje náhled do tvorby RT aplikace a jejího využití při měření latence. Dále práce popisuje metody pro měření latence.

Klíčová slova

Raspberry Pi, GPIO, PREEMPT_RT patch, cyclictest

Abstract

This bachelor's thesis deals with latency analysis on Raspberry Pi OS and it finds out how to PREEMPT_RT patch will affect latency. The thesis describes how to compile Linux kernel and how to apply the PREEMPT_RT patch. The thesis also provides an insight into creation of real-time application and how to use this application in measurement of latency. The thesis further describes methods for measurement of latency.

Keywords

Raspberry Pi, GPIO, PREEMPT_RT patch, cyclictest

Bibliografická citace

MITĚV, Mitko. *Analýza latence Raspberry Pi OS s PREEMPT_RT záplatou.* Brno, 2022. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/142699. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Ing. Lukáš Pohl, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta:	Mitko Mitěv
VUT ID studenta:	191958
Typ práce:	Bakalářská práce
Akademický rok:	2021/22
Téma závěrečné práce:	Analýza latence Linux jádra na Raspberry Pi OS

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.

V Brně dne: 23. května 2022 ----------

podpis autora

Poděkování

Tímto bych chtěl poděkovat vedoucímu bakalářské práce Ing. Lukáši Pohlovi, Ph.D. za odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

V Brně dne: 23. května 2022 ----------

podpis autora

Obsah

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

Úvod

V současné době existuje několik metod, jak z operačního systému Linux vytvořit real-time systém. Tyto snahy jsou v poslední době navíc motivovány rostoucím počtem mikropočítačů založených převážně na platformě ARM. Jedním z významných zástupců této sféry jsou mikropočítače Raspberry Pi od firmy Raspberry Pi Foundation.

Cílem semestrální práce je aplikace PREEMPT_RT patche na operační systém Raspberry Pi OS a změření latence OS po aplikaci tohoto rozšíření.

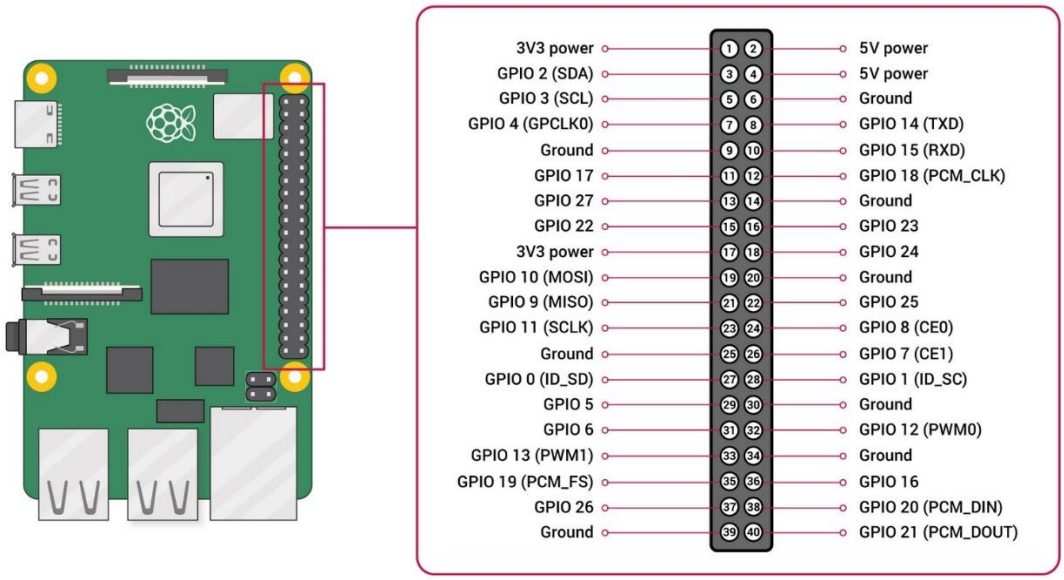
První kapitola se zabývá představením Raspberry Pi a popisem jeho vstupně/výstupních pinů. Druhá kapitola se zabývá představením real-time systému. Dále popisuje možnosti realizace real-time systému na Linuxu a podrobně se zabývá PREEMPT_RT rozšířením. Třetí kapitola popisuje kompilaci Linux kernelu a aplikaci PREEMPT_RT rozšíření. Ve čtvrté kapitole jsou probrány základní plánovací politiky OS, které se budou využívat v RT aplikaci, která je popsána v páté kapitole. V šesté kapitole jsou popsány metody pro uspání a probuzení procesu. Sedmá a osmá kapitola se zabývá měřením latence OS a vyhodnocením výsledků.

1. RASPBERRY PI

Raspberry Pi je jednodeskový počítač s architekturou ARM vyvinutý britskou společností Raspberry Pi Foundation. Výhodou Raspberry Pi je univerzálnost. Lze jej použít buď k ovládání různých zařízení přes rozhraní GPIO nebo jej lze používat jako náhradu za stolní počítač. Součástí Raspberry Pi jsou výstupy pro HDMI a USB rozhraní, které umožňuje připojit klávesnici a myš.[7]

1.1 GPIO

Velkou výhodou Raspberry Pi je skutečnost, že obsahuje GPIO piny, pomocí kterých lze řídit různá zařízení (ovládání LED Diod, řízení krokových motorů atd…). Raspberry Pi obsahuje GPIO se 40 piny. Každý pin lze softwarově nastavit buď jako vstup nebo jako výstup. [8]



Obrázek 1.1 Rozložení GPIO pinů na Raspberry Pi [6]

2. SYSTÉMY REÁLNÉHO ČASU

Základní požadavek na systémy reálného času je determinismus. Operace jsou prováděny ve fixovaných, předem určených časech nebo časových intervalech. Reakce na přerušení musí proběhnout tak, aby byl systém schopen obsluhy všech požadavků v požadovaném čase.[2] Jinými slovy, bez ohledu na to, jakou úlohu real-time aplikace vykonává, musí ji vykonat nejen správně, ale i včas. Real-time neznamená, že má aplikace běžet co nejrychleji, ale musí kritické úkony vykonávat v přesně stanovený čas, neboli real-time se vypořádává s garancí a ne s čistým časem.

V zásadě jsou real-time systémy charakterizovány tím, že aplikace nevytvářejí vlastní plánování v prostředí, na kterém pracuje. Ale je to prostředí (fyzický svět), které vytváří plánování pro real-time aplikace prostřednictvím přerušení, které aplikace musí zpracovat ve specifikovaném čase. To znamená, že tyto procesy vyžadují nějakou formu předvídatelnosti od systémového softwaru. Smysl real-time systémů je garance včasné reakce při interakci s reálným světem za předpokladu, že nedojde k poruše. V praxi dokonce může systém obětovat propustnost, protože nezáleží na tom, kolik úloh vykonáme v daném časovém období, ale jestli je včas zpracována. [1]

2.1 Realizace real-time systému na Linuxu

Linux jako takový není navržen jako operační systém reálného času, ale existují metody, jak ho adaptovat pro real-time použití. Jedním z řešení je použití dual-kernel architektury. Dual-kernel architektura obsahuje real-time jádro a Linux zde běží jako úloha s nízkou prioritou. Nevýhoda tohoto řešení spočívá v tom, že je nutné udržovat mikro kód pro nový hardware. Dále toto řešení vyžaduje abstraktní hardwarovou vrstvu (HAL), na které poběží Linux. Tyto systémy také často neumožňují při vývoji aplikací využívat standardní knihovny, ale je zapotřebí využívat speciální nástroje.

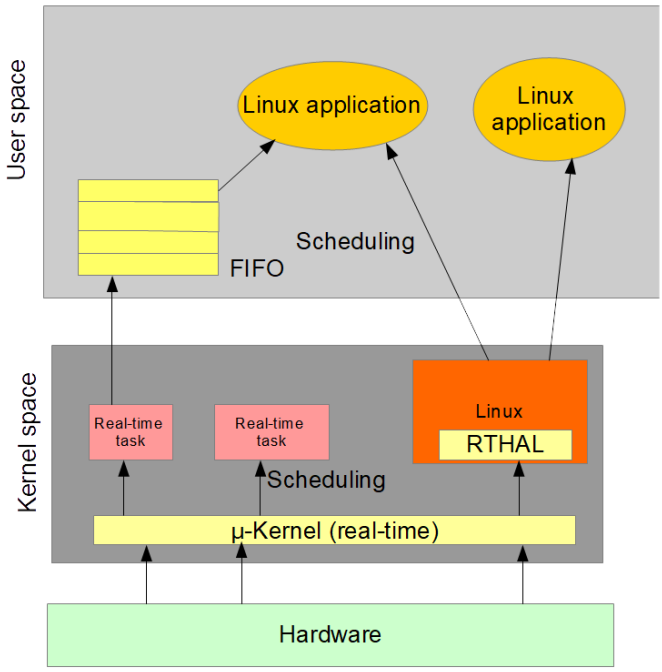
Mezi dual-kernel systémy patří:

- RTAI – Real time application Interface
- Xenomai
- RTLinux

Dalším řešením je vytvořit samotné Linux jádro real-time. Toto řešení se označuje jako single-kernel. V současné době se toto řešení využívá PREEMPT_RT patch. Výhodou PREEMT_RT rozšíření je, že většina vlastností je součástí “mainline“ a při psaní real-time aplikací lze používat standardní C knihovny a POSIX standardy.

2.1.1 RTAI

RTAI obsahuje řadič přerušení, který odchytává přerušení a v případě potřeby je přesměruje do Linuxu. Využívá konceptu HAL (Hardware Abstraction Layer) k získávání informací z Linuxu a k zachytávání některých nezbytných funkcí. RTAI považuje Linux za úlohu běžící na pozadí.[2]

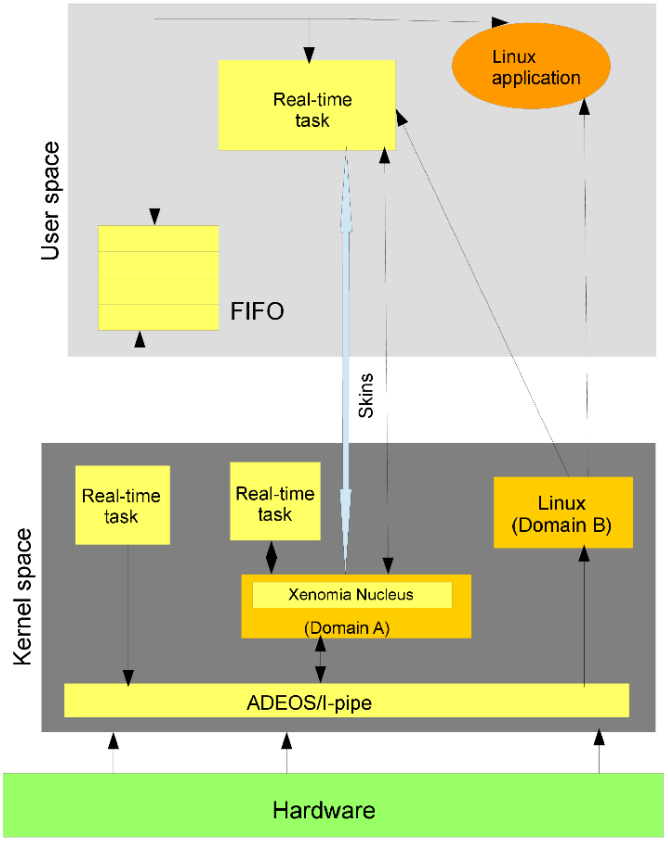


Obrázek 2.1 RTAI architektura [2]

2.1.2 Xenomai

Jedná se o real-time podsystém, který může být integrován v Linuxu, aby garantoval předvídatelné časové odezvy aplikací. Realizace je pomocí co-kernelu, který běží společně s Linuxem na stejném hardwaru. Xenomai umožnuje, aby real-time aplikace běžely v uživatelském prostředí s ochrannou jednotkou správy paměti. Real-time aplikace jsou během jejich časově kritických operací exkluzivně pod kontrolou co-kernelu, čímž je dosažena nízká latence pro jejich kód běžící ve standardním

Linuxovém jádře. Xenomai se skládá se tří hlavních částí: ADEOS/I-pipe, Xenomai jádra a Xenomai skinu.



Obrázek 2.1.2 Xenomai architektura [2]

ADEOS/I-pipe:

Jedná se o tzv. pipeline přerušení, které má zaručit, že Linuxové jádro nikdy neodloží externí přerušení. Snahou je, aby přerušení bylo blokováno nebo maskováno po co nejkratší dobu a co nejrychleji bylo doručeno do Xenomai (Domain A na obr.2) bez ohledu na to, zda by jej přijalo Linuxové jádro. Přerušení směrované do Linuxového jádra by nikdy nemělo spouštět obsluhu přerušení v sekci, kde jsou přerušení blokována.

Abychom tohoto chování dosáhli, potřebujeme dodatečný software mezi hardwarem, Linuxem a Xenomai, který bude fungovat jako virtuální programovatelný kontrolér přerušení. Tento virtuální kontrolér přerušení se nazývá I-pipe nebo ADEOS (viz. obr.2). I-PIPE organizuje systém jako sadu domén propojených pomocí pipeline softwaru. V implementaci I-PIPE tyto domény sdílejí společný adresový prostor. Tato implementace je výhodná ve chvíli, kdy vlákno v jednom bodě potřebuje probudit službu z Linuxu a v jiném bodě službu z Xenomai.

Další velkou výhodou I-PIPE implementace je snadný port na jiné CPU platformy. Při portu na jinou platformu CPU je vyžadována změna pouze malého množství kódu.[1]

Xenomai jádro:

Jádro má na starosti zásobování všech zdrojů operačního systému, které mohou požadovat skiny pro správné napodobení tradičních RTOS APIs. Tento požadavek je způsoben tím, že Xenomai běží na non-real-time Linux jádře a real-time procesy musí exklusivně volat služby Xenomai, aby byly zachovány předvídatelné latence.

Jádro Xenomai lze popsat jako abstraktní RTOS, protože definuje základní stavební bloky, které mohou být speciálně určeny k implementaci jakéhokoliv real-time API. Tyto bloky jsou shromážděny do jednoho modulu nazvaného Xenomai nucleus.

Xenomai skiny:

Skiny slouží k emulaci různých RTOS a vývojáři mají možnost navrhnout i nové skiny pro port aplikací z RTOS, které Xenomai zatím nepodporuje. Skiny jsou navrženy jako speciální Linux ovladače, které programátor může povolit staticky nebo jako modul při vytváření obrazu cílového kernelu.[1]

2.1.3 RTLinux

„Jednalo se o hard real-time variantu Linuxu, která umožňovala ovládání robotů, data pořizujících systémů nebo zařízení citlivých na čas, výrobních díle etc.

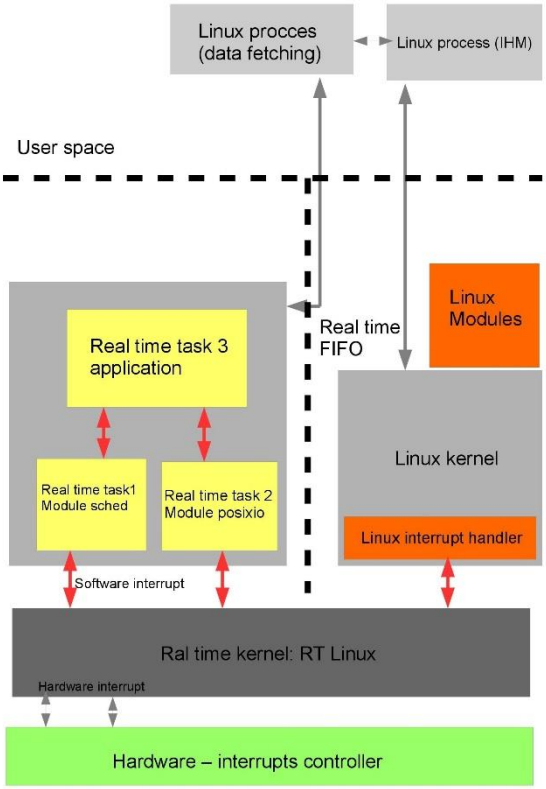
1. verze RTLinuxu byla navržena pro provoz na méně výkonných a levnějších počítačích založených na architektuře x86.

2. verze již umožňovala symetrický multiprocessing a pracovala na větším rozsahu systémů a měla rozšíření pro zjednodušení použití.

RTLinux nabízel možnost běhu speciálních real-time úloh a obslužných rutin přerušení na stejném počítači jako standartní Linux. Tyto úlohy a obslužné rutiny se vykonávají podle potřeby bez ohledu na to, co Linux právě dělá.

RTLinux druhé generace byl strukturován jako soubor volitelných komponentů doplňujících jádro. Například instalace obslužných rutin s velmi nízkou prodlevou a rozšíření této komponenty o SMP (Symetric Multi Processing).“ [3]

Vývoj RTLinuxu je od roku 2011 zastaven.



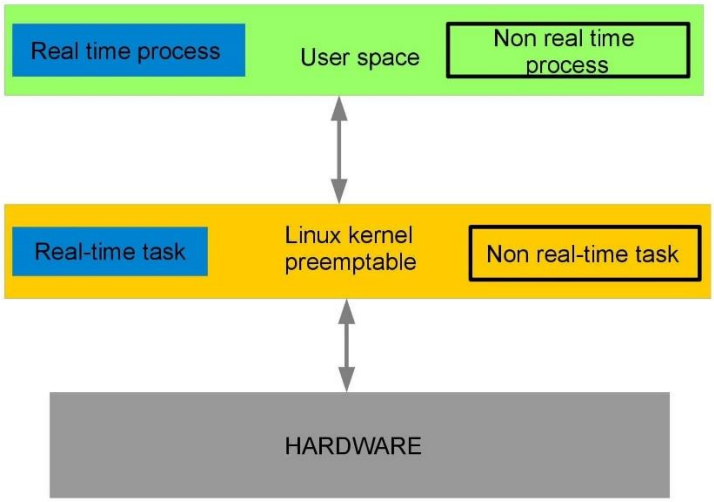
Obrázek 2.1.3 RTLinux architektura [4]

2.1.4 PREEMPT_RT patch

Cílem je, aby Linux jádro bylo předvídatelné a poskytovalo omezené latence.

PREEMT_RT patch umožnuje přerušení jádra během systémových volání za účelem obsloužení úlohy s vyšší prioritou. Hlavním úkolem tohoto rozšíření je zvýšení stupně preempce kódu směrem k plně preemptovatelnému jádru (PREEMPT_RT_FULL). Tento stupeň preempce dovoluje real-time úlohám preemptovat jádro i v kritických sekcích. Nicméně některé oblasti nejsou stále preemptivní: například horní polovina obsluhy přerušení a kritické sekce chráněné tzv. raw spinlock (raw_spinlock_t). Všechny “sleeping mutexy“ byly nahrazeny rt_mutex, které implementují dědičnost priorit.

Výhodou PREEMT_RT patche je, že většina vlastností je součástí “mainline“ a při psaní real-time aplikací lze používat standardní C knihovny a POSIX standardy.[2]



Obrázek 2.1.4 PREEMPT_RT patch architektura [2]

2.2 Vlastnosti PREEMPT_RT patche

V následujících kapitolách budou popsány některé vlastnosti, které PREEMPT_RT patch přináší do Linuxu tak, aby splňoval požadavky na RT systémy.

V této kapitole se budeme zabývat:

- Přerušením ve formě vlákna
- Hard IRQ ve formě vlákna
- Softirq ve formě vlákna
- Softirq vlákna časovačů
- Dědičností priorit

2.2.1 Přerušení ve formě vlákno

Pokud zařízení vyvolá asynchronní událost, která vyžaduje akci CPU a jsou povolená přerušení, tak zařízení skrze řadič přerušení vyšle signál. Procesor po obdržení tohoto signálu přeruší právě prováděný kód a vykoná obslužnou rutinu přerušení (ISR). ISR má vyšší prioritu než kterákoliv uživatelská úloha. Jediná událost, která může ISR přerušit je jiné přerušení.

Pokud jsou ovladače zařízení dobře napsané, tak se snaží co nejméně využívat ISR a snaží se využívat jiné metody, které Linux nabízí pro dokončení úlohy vyvolané asynchronní událostí.

První takovou metodou je softirq. Jedná se o servisní rutinu, která se provádí po návratu ISR a před tím, než se obnoví proces, který byl přerušen. Pokud je příliš mnoho softirq ve frontě, tak kernel probudí kernel vlákno s vysokou prioritou (ksoftirqd), aby je dokončilo. Ksoftirqd vlákno se spustí, pokud je spuštěna softirq rutina a současně

jsou zakázaná další přerušení. Anebo se spustí, pokud je softirq rutina zpracována vícekrát než jednou před návratem k procesu, který byl přerušen.

Další možností je metoda tasklet. Tasklet je podobná softirq, také se provádí po ISR a před návratem do procesu, který byl přerušen. Rozdíl mezi tasklet a sofiqr je ten, že stejné softirq může běžet na dvou samostatných CPU, zatímco tasklet ne. Proto softirq, na rozdíl od tasklet, musí používat zámky pro ochranu nelokálních dat a dalších zdrojů. Z těchto skutečností vyplývá, že tasklet nemusí být reentrantní. Dalším rozdílem je, že tasklet funkce může běžet na jiném CPU, než které spustilo tasklet. Tasklet je ve skutečnosti implementován pomocí softirq. Softirq funkce, která realizuje tasklet se musí ujistit, že dvě tasklet funkce neběží ve stejný čas. Proto tasklety mohou být prováděny pomocí ksoftirqd vláken.

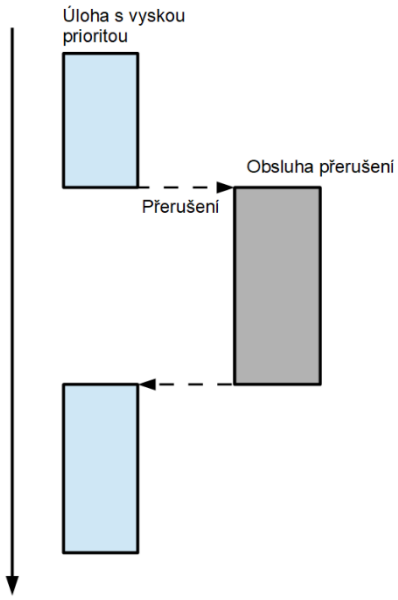
Kernel vlákno je vlákno, které může běžet pouze v kernelu a může ho probudit ISR pro zpracování jakékoliv úlohy, která zůstala k dokončení v obslužné rutině přerušení. Tak se lze rychle vrátit z ISR a procesor nebo kernel se může vrátit k úloze, která byla přerušena.

Kernel vlákno má stejné vlastnosti jako jakékoliv jiné vlákno v Linuxu. Například jej lze plánovat, může měnit svoji prioritu, běží na přiděleném procesoru a může s ním manipulovat systémový administrátor. Kernel vlákno má větší režii oproti sofirq nebo tasklet, ale nabízí větší flexibilitu.

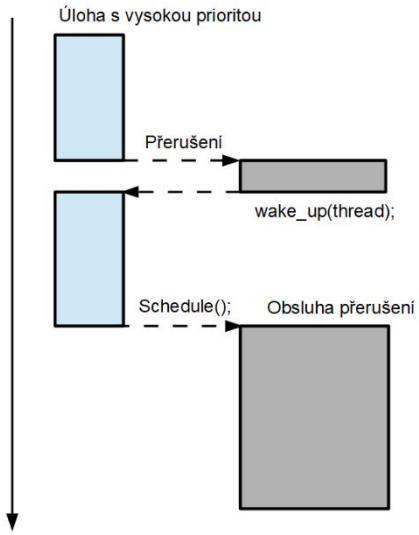
U Linux jádra, které není RT, nemůže žádný proces bez ohledu na jeho priority být upřednostněn pčed softirq nebo taskletem. Na druhé straně softirq nebo tasklet může být upřednostněno před jakýmkoliv procesem. Proto i když má kernel vlákno větší režii, tak dává plánovači kernelu větší flexibilitu a systémovému správci větší kontrolu. Tato vlastnost je důležitá pro real-time prosředí. Proto RT patch v rámci dosažení co nejmenší latence transformuje softirq a tasklet do kernel vlákna.[1]

2.2.2 Hard IRQ ve formě vlákna

Tvrdý požadavek na přerušení (IRQ) je druh obálky kolem ISR, trvající od chvíle, kdy přerušení pozastaví proces do chvíle, kdy ISR vrátí procesor danému procesu. To může vést k inverzi přerušení, kde obslužná rutina přerušení vezme čas procesu s vysokou prioritou, aby upřednostnila proces s nižší prioritou. RT rozšíření snižuje čas inverze přerušení na co nejmenší dobu. A to tak, že převede obslužnou rutinu na vlákno. Když je přerušení spuštěno tak ISR pouze probudí kernel vlákno, které spustí funkci registrovanou ovladačem zařízení. Pokud má vlákno obsluhy přerušení vyšší prioritu než právě běžící vlákno, tak toto vlákno pozastavit právě běžící proces. Pokud má původní vlákno větší prioritu, než vlákno obsluhy přerušení tak obsluha přerušení je přesunuta na jiné CPU, anebo musí čekat než se vlákno s vyšší prioritou vzdá procesoru.[1]



Obrázek 2.2.1 Inverze priorit [1]



Obrázek 2.2.2 Preemt_RT patch řešení inverze priorit [1]

2.2.3 Softirq ve formě vlákna

Pokud se k vyřízení požadavku přerušení používaji softirq tak může dojít k inverzi priorit. Softirq mohou často způsobit mnohem víc škody než kernel vlákno, protože většinou běží pro rutiny, které jsou časově náročnější. Proto softirq způsobí vetší latenci než přerušení. Původně RT patch jednoduše spustil všechny softirq pod ksoftirqd vláknem. To umožnilo systémovému správci umístit vlákna s vysokou prioritou před sofirq. Toto řešení mělo nevýhodu v tom, že všechna sofirq jsou sjednocena do jedné skupiny s jednou prioritou a jsou spárována jedním vláknem. V současné době RT rozšíření separuje jednotlivé softirq rutiny a spouští je jako oddělená vlákna. [1]

2.2.4 Softirq vlákna časovačů

Jedná se o nejdůležitější část RT systémů. Softirq-timer vlákno se stará o většinu časových událostí včetně vrstvy pro síťové události.

Softirq-hrtimer je vlákno, které se stará o POSIX implementaci časovačů. Pokud RT aplikace používá volání POSIX časovačů (např. timer_create), musí se dát pozor na

priority softirq-hrtimer vlákna. Proces s vysokou prioritou, který očekává přerušení od POSIX časovače, musí mít nižší prioritu než softirq-hrtimer vlákno. V opačném případě, když se časovač vypne, tak funkce pro signalizaci procesu s vysokou prioritou může být proces s vysokou prioritou vyhladověn sám sebou. Pokud tento signál slouží k zastavení smyčky, může se systém zablokovat, protože se systém nevzdá procesoru. RT patch nutí POSIX časovače, aby každý časový interval byl alespoň jeden jiffie, což je jednotka času, kterou používá Linux kernel pro měření času a je definována v Hz pevně nastavenou v kernel konfiguraci. To má zabránit tomu, aby softirg-hrtimer vlákno nehladovělo, pokud proces ignoruje signál, který má přijít od časovače. Také to zabraňuje procesu s nízkou prioritou, aby prováděl více jak jednu časovou událost za jeden jiffie. Tato událost by mohla zabránit spuštění procesu s vyšší prioritou a tím způsobit latenci, která by způsobila přetížení systému.[1]

2.2.5 Dědičnost priorit

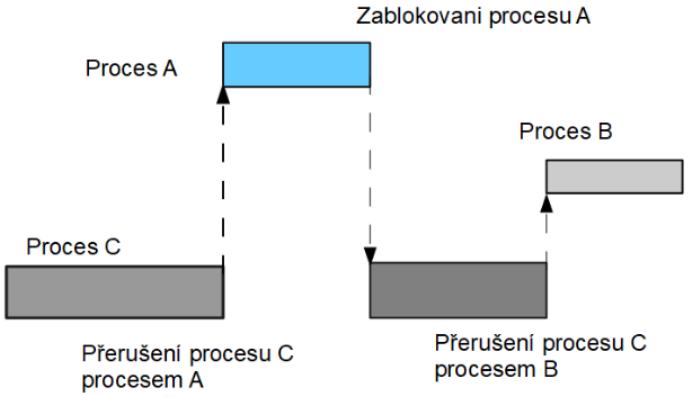
Dědění priorit se používá pro řešení neomezené inverze priorit. Pro demonstraci neomezené inverze priorit předpokládáme tři separátní procesy A,B a C. Kde proces A má nejvyšší prioritu a proces C nejnižší prioritu. Nejdříve se spustí proces C a vezme si zámek. Proces C je během svého provádění přerušen procesem A, který se pokusí vzít si stejný zámek jako proces C, ale jelikož je tento zámek zabraný, je proces A zablokován a systém vrátí procesor procesu C a čeká, než proces C vrátí zámek. Ale poté je spuštěn proces B, který přeruší proces C a běží blíže nespecifikovanou dobu. Problém spočívá v tom, že proces B nepřerušil pouze proces C, který měl nižší prioritu, ale i proces A, který čekal na dokončení procesu C a měl vyšší prioritu.

RT patch k řešení tohoto problému přináší dědění priorit pro zámky v kernelu pro zabránění neomezené inverze priorit. Využívá k tomu stejný kód, který se používá pro futexy.

Futexy byly implementovány do Linux jádra Rustym Russelem jako možnost, jak provádět zamykání v uživatelském prostoru bez nutnosti volání jádra Linuxu. Pro jeho realizaci se využívá sdílený adresový prostor mezi dvěma vlákny s atomickými operacemi poskytovanými hardwarem k vytvoření, uzamknutí a odemčení mutexu. Uzamčení a uvolnění mutexu proběhne ve většině případů bez problému. To znamená že, jakýkoliv vstup do kernelu by způsobil zbytečnou režii. Při použití sdíleného paměťového prostoru a hardwaru s atomickými operacemi, může futex vzít a uvolnit zámek mutexu bez nutnosti použít systémové volání. Jediný okamžik, kdy je potřeba použít systémové volání, nastává ve chvíli, když vlákno, které je blokováno, spí. Když operační systém zjistí, že na uvolňovaný mutex čeká další vlákno, tak toto čekající vlákno probudí pomocí systémového volání jádra. Problém toho řešení je, že pokud vlákno, které drží mutex je nečekaně ukončeno. Potom vlákno nikdy neuvolní daný mutex a vlákno které, na tento mutex čeká nebude nikdy probuzeno. To může u kritických aplikací vést až k zablokování systému.

Tento problém řeší tvz. robust futex. Robust futex přidává vrstvu v uživatelském prostoru, kterou může kernel použít, aby zjistil, která futex vlákna mohla být ukončena. Kernel zkontroluje všechny futex vlákna a pokud některý futex je držen již ukončeným vláknem, tak je tento futex uvolněn a další čekající vlákna jsou probuzena.

Od kernel verze 2.6.18 je dědičnost priorit součástí „mainline“ Linux kernelu a je aplikována pouze na rychlé mutexy (futex). Ale původní algoritmus byl vyvinut jako součást RT patche a stejný algoritmus používá RT patch i pro interní zámky.



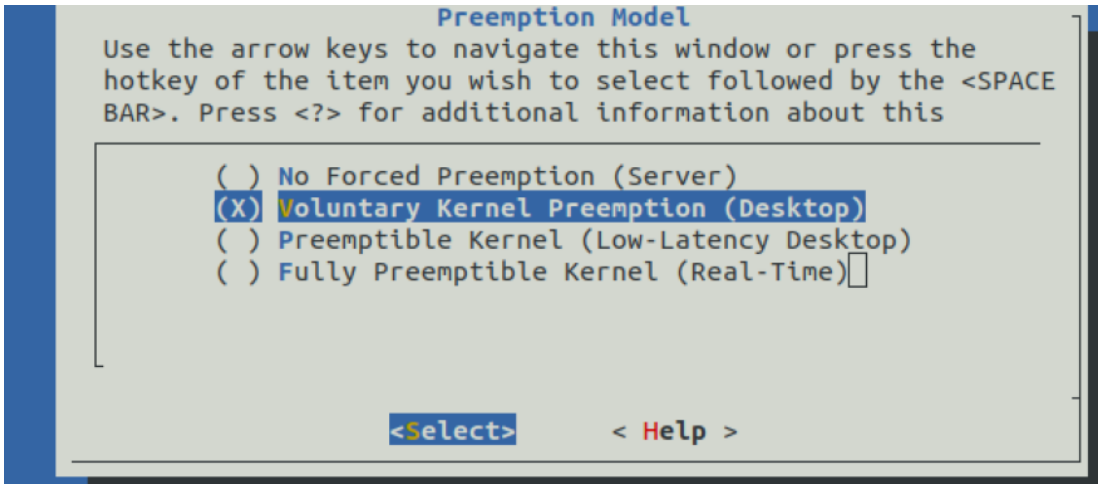
Obrázek 2.2.3 neomezená inverze priorit [1]

3. KOMPILACE LINUX JÁDRA A ZAVEDENÍ PREEMPT_RT PATCHE

Kompilace Linux jádra se využívá pro zavedení ovladačů hardwaru které nejsou součástí jádra. Dále slouží k odstranění nepotřebných modulů z jádra, a tím zlepšuje výkon počítače. V této práci je kompilace jádra využita pro zavedení PREEMPT_RT patche do Linux jádra. Postup kompilace je popsán v příloze A.

Pro kompilaci byla zvolena verze jádra 5.4.82. Toto jádro bylo zkompilováno ve dvou verzích. A to s PREEMPT_RT patchem - a jádro bylo pojmenováno 5.4.82-rt46-v8-Azuki+. Další jádro bylo zkompilováno bez patche, aby bylo možné porovnat vliv patche na latenci a to jádro bylo pojmenováno 5.4.83-v8-Maple+.

Během kompilace lze zvolit několik typů preempce, které jsou ukázány na obrázku č.3.1.1. Tyto typy preempce a jejich vliv na chování Linux jádra jako real-time systému budou popsány v následujících kapitolách.



Obrázek 2.2.1 Volba typu preempce při kompilaci Linux jádra

3.1 No Forced Preemption

Pokud je zvolena tato možnost, tak proces s vysokou prioritou, který je probouzen, musí počkat, než se právě běžící proces přepne zpět do uživatelského módu nebo musí počkat, než se dokončí I/O operace.

Výhodou tohoto řešení je malý počet přepínání kontextu a tím i nízká režie. Proto je toto řešení vhodné pro servery, kde aplikaci běží sériově a není zapotřebí rychle přepínat mezi procesy. Naopak toto řešení není vhodné pro klasické kancelářské počítače, jelikož může způsobovat trhání kurzoru myši během pohybu po obrazovce nebo špatné přehrávání a nahrávání zvuku.

3.2 Voluntary Kernel Preemption

Tento mód nabízí větší flexibilitu tím, že některé sekce Linux jádra mohou provádět změnu kontexu. Tyto sekce se volí na základě toho, zda mohou dané části způsobit přeplánování. Kód běžící v Linux jádře nesmí způsobit proces přeplánování, pokud je v jádře zakázaná obsluha přerušení. A to z toho důvodu, že kód zakáže přerušení, pokud na daném CPU běží atomická operace, která používá proměnou specifickou pro dané CPU nebo pokud tento kód zabere nějaký zámek, který může být používán i v obsluze přerušení. K zjištění, zda může daná část kódu způsobí přeplánování, se využívá funkce might_sleep.

Pokud je tedy zvolena možnost Voluntary Kernel Preemption tak vždy, kdy by měl proces s vysokou prioritou čekat, než právě běžící proces přejde zpět do uživatelského módu nebo bude uspán, tak se zavolá funkce might_sleep. Ta zjistí, zda danému procesu lze přiřadit jiné CPU a pokud ano, tak funkce zjistí, zda nenastane přeplánování. Proto každá sekce v Linux jádře, která volá funkci might_sleep, se stává preemptivním bodem. [1]

3.3 Preemptible Kernel

Tato možnost byla vytvořena pro systémy se symetrickým multiprocesorem (SMP). V SMP jsou kritické sekce chráněny před přístupem od jiného procesu běžícího na jiném CPU. K tomu jsou většinou využity spin-locky. Tento mód přináší do Linux jádra proměnou „preempt count“. Normálně je tato proměnná nastavena na nulu. Ve chvíli, kdy vlákno obdrží spin-lock, se tato proměnná inkrementuje, a když vlákno spin-lock uvolní, dojde k dekrementaci této proměnné. Pokud je hodnota této proměnné větší jak nula, tak právě běžící proces nemůže být upřednostněn. Ale ve chvíli, kdy se probudí proces s vyšší prioritou, než kterou má právě běžící proces a hodnota „preemt count“ je nulová, tak to značí, že nehrozí přeplánování a procesu může být přidělen procesor. V opačném případě proces musí počkat, než daná proměnná dosáhne hodnoty nula.[1]

3.4 Fully Preemptible Kernel

Tento mód převádí všechny spin-locky na mutexy. Převod je zde z toho důvodu, že pokud nějaký proces držel spin-lock, tak nemohl být přerušen žádným jiným procesem. A to i za podmínky, že nový proces nepřistupoval ke kritické sekci, kterou spin-lock chránil. Navíc proces, který čekal na uvolnění spin-locku, tak činil v aktivní čekací smyčce. Při převodu spin-locku na mutex proces, který se pokusí přistoupit ke kritické sekci, je uspán a jádro může pro procesor naplánovat jiný proces. Toto řešení přibližuje Linux nejblíže k požadavkům na real-time systém.

4. PLÁNOVAČE

Základní úlohou plánovačů v operačním systému je rozhodnout, kterému z čekajících procesů bude přidělen procesor. Zároveň plánovací algoritmy musí splňovat několik cílů, které jsou většinou protichůdné a jsou jimi: rychlá odezva procesu, zabránění hladovění procesů, sladění procesů s nízkou a vysokou prioritou a dobrou propustnost pro úlohy běžící na pozadí. [5]

Pravidlo pro to, který proces bude vybrán, se nazývá plánovací politika. V praxi se používají následující typy plánovačů:

- FIFO
- Round-robin (RR)
- Deadline

4.1 FIFO plánovač

Využívá principu first in first out. Proces běží, dokud se nevzdá procesoru nebo pokud ho nepředběhne úloha s vyšší prioritou.[16]

Tabulka 4.1 Seznam procesů pro demonstraci FIFO a RR plánovače [9]

Process	Čas příchodu	Potřebný čas	Časové kvantum pro RR
A	0	3	1
B	2	6	1
C	4	4	1
D	6	5	1
E	8	2	1

Tabulka 4.2 FIFO plánovač [9]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A																				
B																				
C																				
D																				
E																				

4.2 Round-robin plánovač

Jedná se o modifikovanou verzi plánovače FIFO, kdy každému procesu je přiřazeno časové kvantum, po které může běžet. Pokud proces běží po dobu, která je delší, než je toto kvantum, je proces přesunut na konec fronty. [16]

Tabulka 4.3 Round-robin plánovač [9]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A																				
B																				
C																				
D																				
E																				

4.3 Deadline

Plánovač vybere úlohu na základě velikosti jejího deadlinu. Čím nižší je deadline, tím větší má úloha prioritu.[16]

Tabulka 4.4 Seznam procesů pro demonstraci Deadline [10]

Process	Potřebný čas	Perioda
A	1	4
B	2	6
C	3	8

Tabulka 4.5 Deadline [10]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A																							
B																							
C																							

5. VYTVOŘENÍ CYKLICKÉ RT APLIKACE

Cyklická RT aplikace je taková aplikace, která se opakuje po pevně stanovené době, například četní dat ze senzoru. Čas pro vykonání této aplikace by měl být menší než je perioda této úlohy. [12]

V této práci jsou použity dva zdrojové kódy. První kód používá plánovací politiku FIFO a RR a druhý kód je vytvořen pro plánovací politiku typu deadline.
Obě aplikace řídí výstupní pin 18 na GPIO, které se nachází na Raspberry Pi s periodu 1 ms. Pro řízení pinů GPIO je použita knihovna funkcí *gpiod.h.*

5.1 RT cyklická aplikace s FIFO a RR plánovací politikou

Při použití FIFO a plánovací politiky se využívají následující knihovní funkce (viz. příloha B) :

- *clock_nanosleep*
- *pthread_attr_setschedpolicy*

Clock_nanosleep

Tato funkce uspí proces na určitý časový úsek. Funkce využívá časovač CLOCK_MONOTIC k určení, zda již uplynula požadovaná doba, po kterou měl proces spát. Výhodou časovače CLOCK_MONOTIC je, že představuje absolutní čas, který se od vyvíjí od pevně stanoveného bodu v minulosti a není ovlivněn změnami systémových hodin. [12]

Pthread_attr_setschedpolicy

Tato funkce je určena pro změnu plánovací politiky. Pokud se má používat politika typu FIFO tak vstupní parametry budou následující: pthread_attr_setschedpolicy(&attr, SCHED_FIFO). V případě použití RR plánovací politiky pak budou: pthread_attr_setschedpolicy(&attr,SCHED_RR).

Základní kostra pro vytvoření aplikace je převzata z [11] a [12].

5.2 RT cyklická aplikace s Deadline plánovací politikou

Na rozdíl od vytváření aplikace s FIFO nebo RR plánovací politikou je zapotřebí definovat systémové volání a strukturu sched_attr, která obsahuje parametry se kterými pracuje Deadline. Tyto parametry totiž nejsou součástí všech Linuxových jader. Také se zde nenastavuje priorita procesu, jelikož deadline má plovoucí prioritu na základě velikosti deadline daného procesu. Celý kód je obsažen v příloze C. Základní kostra kódu je převzata z [13].

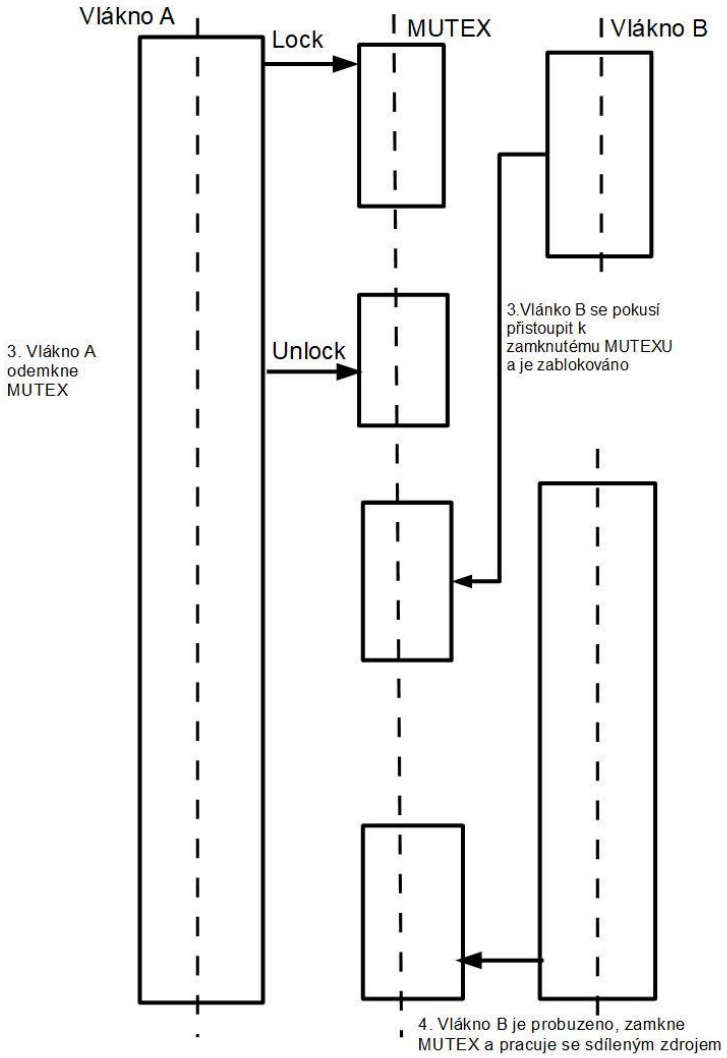
6. PROBOUZENÍ A USPÁNÍ PROCESU

V následujících kapitolách budou probrány metody sloužící pro uspání a následující probuzení procesu. Tyto principy budou využity v kapitole 8 pro probouzení procesu na základě příchozího signálu na GPIO pinu. Následující kapitoly popisují:

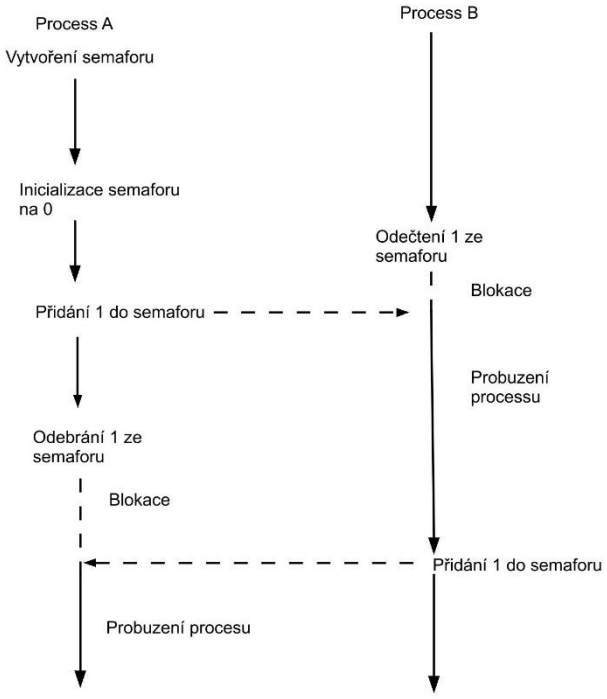
- Mutexy a semafory
- Spin-locky
- Character device

6.1 Mutexy a Semafory

Jedná se o metody, které se používají v kritických sekcích kódu. Kritická sekce je taková sekce v kódu, kde dochází ke sdílení zdrojů více procesy. V těchto sekcích je třeba spravovat přístup ke zdrojům jednotlivých procesů. Semafory jsou číselná hodnota s párem funkcí, které se obvykle jmenují *P* a *V*. Pokud proces chce vstoupit do kritické sekce, zavolá *P* funkci na daném semaforu. Pokud je hodnota semaforu větší než 0, je tato hodnota snížena o 1 a proces pokračuje do kritické sekce. Ale pokud je hodnota semaforu rovna nebo menší než 0, tak proces musí čekat, než jiný proces uvolní semafor. Uvolnění semaforu je pomocí funkce *V*. Tato funkce zvýší hodnotu semaforu o 1 a pokud je to možné, probudí čekající proces. Pokud ke zdrojům potřebuje přistupovat více vláken, je zapotřebí metoda, která zajistí exkluzivní přístup ke sdílenému zdroji pouze jednomu vláknu. Tato metoda se nazývá mutex. Pokud se vlákno pokusí získat mutex, který vlastní už jiné vlákno, je toto vlákno pozastaveno (uspáno), dokud první vlákno mutex neuvolní. Mutexy také lze předávat mezi hranicemi aplikační domény. Mutexy jsou implementovány jako binární semafory s hodnotou 1 nebo 0.[21]



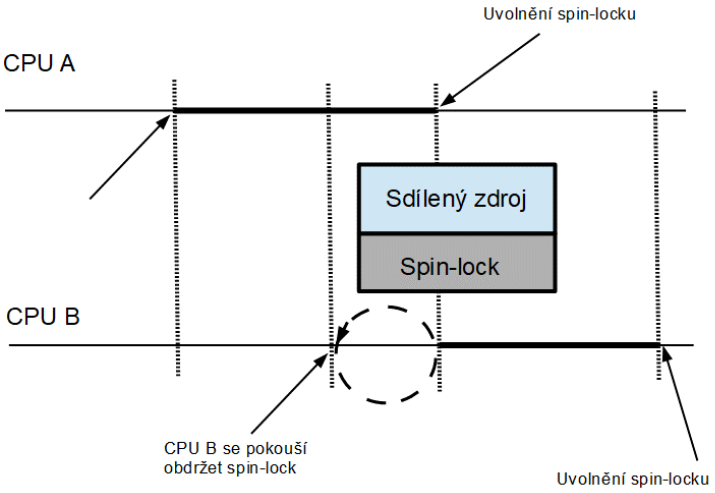
Obrázek 6.1.1 Mutex [24]



Obrázek 6.1.2 Semafor [22]

6.2 Spin-locky

Spin-locky jsou další metoda pro ošetření přístupu více procesů k jednomu zdroji. Spin-locky na rozdíl od semaforů nebo mutexů neuspí proces, který čeká na daný spin-lock. Proto se spin-locky používají v obsluze přerušení. Spin-locky se skládají opět ze dvou hodnot „lock“ a „unlock“. Tyto hodnoty jsou implementovány jako jeden bit v integer hodnotě. Proces, který se pokusí přistoupit ke kritické sekci, si nejprve zjistí stav spin-locku. Pokud je spin-lock dostupný, nastaví se „lock“ bit a proces pokračuje do kritické sekce. Pokud je spin-lock držen jiným procesem, proces, který se pokusil ke spin-locku přistoupit, přejde do aktivního čekání, dokud se spin-lock neuvolní.[21]



Obrázek 6.2.1 Spin-lock [23]

6.3 Character Device

Character device je speciální soubor, který umožňuje komunikovat mezi user-space aplikací a hardwarem. V této práci je soubor navržen, tak že pokud se z něj určitý proces pokusí číst, tak je uspán. Naopak pokud se proces do tohoto souboru pokusí zapsat, tak je probuzen proces, který byl předtím uspán. K tomu je využita metoda workqueues, která umožňuje uspat běžící proces.[21][20]

7. MĚŘENÍ LATENCE NA RASPBERRY PI

7.1 Měření pomocí cyclictest a skriptu mklatencyplot

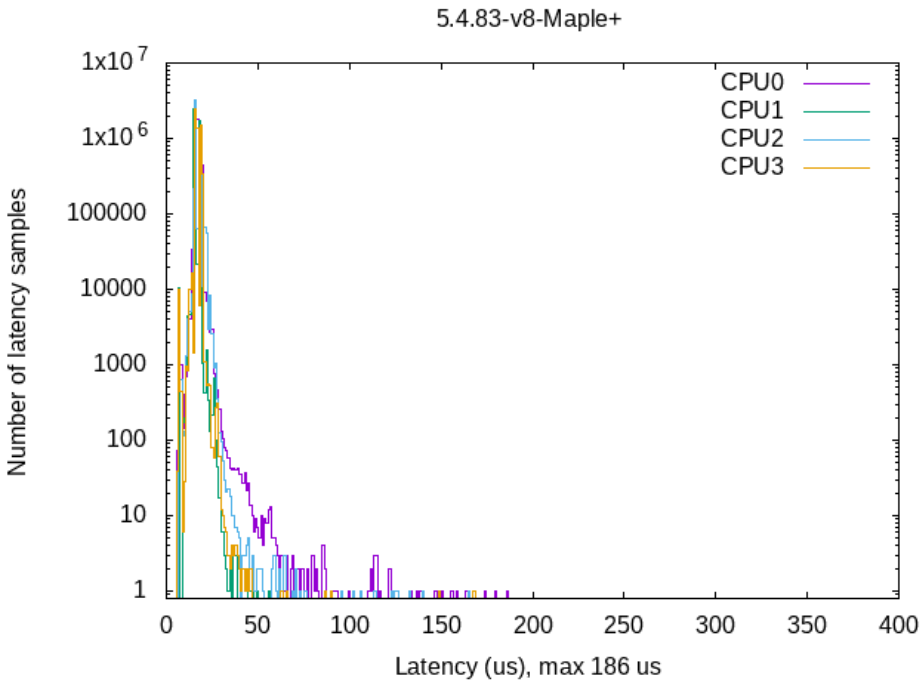
Cyclictest je nástroj pro opakované měření časového rozdílu mezi očekávaným časem probuzení vlákna a skutečným časem kdy se vlákno probudí. Tento nástroj dokáže měřit latency způsobenou hardwarem, firmwarem a OS. Tento nástroj je součástí rt-test balíčku.

Samotné měření se provádí tak, že program nejdříve spustí non real-time vlákno, které spustí definovaný počet měřících vláken s definovanou prioritou a plánovací politikou typickou pro real-time (SHCED_FIFO). Vlákna určená k měření se probouzejí v definovaném intervalu, který je určen vypršením časového limitu časovače (cyclic alarm). Poté se vypočítá rozdíl mezi naprogramovanou dobou probuzení a skutečnou dobou probuzení vlákna. Výsledek je předán hlavnímu vláknu skrze sdílenou paměť. Hlavní vlákno sleduje a tiskne minimální, maximální a průměrnou hodnotu latence. [14]

Mklatencyplot je bash skript, který v sobě spouští cyclictest a z naměřených hodnot vytvoří histogram. Skript je dostupný z [15].

7.1.1 Naměřené hodnoty pomocí mklatencyplot

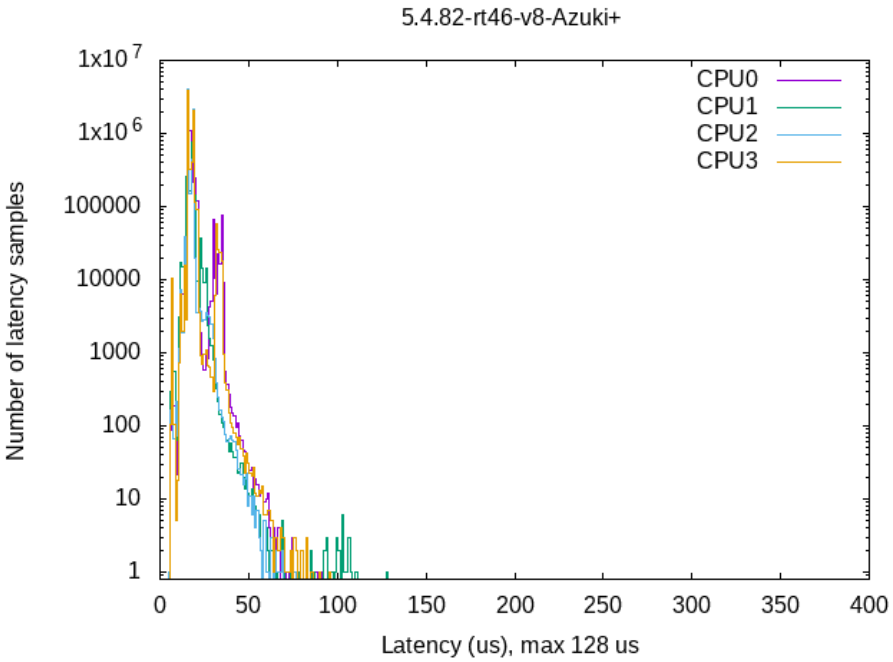
První měření proběhlo na Linux jádře bez PREEMPT_RT patche a výsledky jsou shrnuty na obrázku 7.1.1. Z grafu vyplývá, že největší naměřená latence je 186 µs, a že většina vzorků zaznamenané latence se pohybuje pod 50 µs.



Obrázek 7.1.1. Výsledek měření pomocí mklatency plot na Linux jádře bez PREEMPT_RT patche

Druhé měření proběhlo na jádře s PREEMPT_RT patchem a výsledky jsou shrnuty na obrázku 7.1.2. Z grafu vyplývá, že většina vzorků zaznamenané latence s pohybuje pod 50 µs a největší zaznamenaná latence je 128 µs.

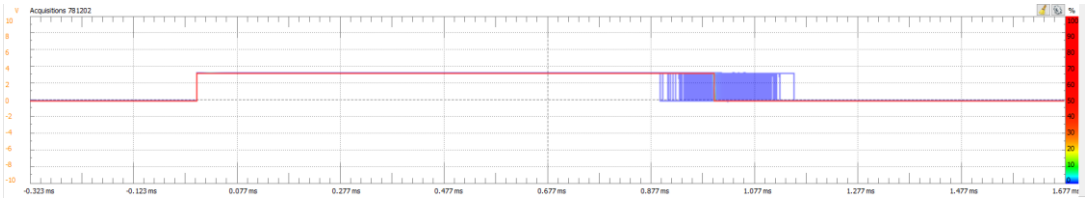
-



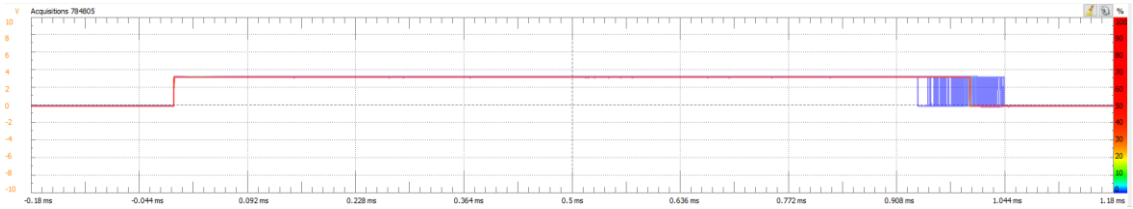
Obrázek 7.1.2 Výsledek měření pomocí mklatency plot na Linux jádře s PREEMPT_RT patche

7.2 Měření latence pomocí zařízení Analog Discovery 2

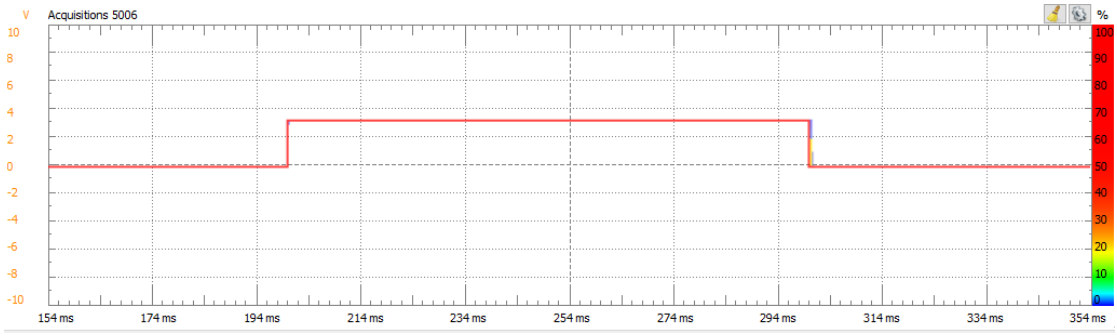
Dalšímu měření byla podrobena RT aplikace popsána v kapitole 6 a zařízení Analog Discovery 2. Měření se provádělo tak, že se na vstup digitálního osciloskopu sledoval čas, který RT aplikace potřebuje ke svému dokončení. Spolu s RT aplikací na pozadí běžel i cyclictest, aby během měření byl systém co nejvíc vytížen. Výsledky jsou shrnuty na obrázcích 7.2.3, 7.2.4 a 7.2.5. Z grafů vyplývá, že největší časová odchylka je u plánovací politiky typu FIFO a RR. Zatímco plánovací politika typu deadline dává nejlepší výsledek.



Obrázek 7.2.3 Výsledek měření pomocí Analog Dicovery 2 pro FIFO plánovací politiku



Obrázek 7.2.4 Výsledek měření pomocí Analog Discovery 2 pro RR plánovací politiku



Obrázek 7.2.5 Výsledek měření pomocí Analog Discovery 2 pro deadline plánovací politiku

7.2.1 Seznam použitých přístrojů a nastavení digitálního osciloskopu

Použité přístroje:

- *Raspberry Pi 4B 2GB RAM, FCC ID: 2ABCB-RPI4B, IC ID: 20953-RPI4B*
- *Analog Discovery 2, SN: 210321ABEA72*

Nastavení osciloskopu v programu WaveForms:

- *Source:* *Channel 1*
- *Type:* *Edge*
- *Condition:* *Rissing*
- *Level:* *2 V*

8. MĚŘENÍ LATENCE NA ZÁKLADĚ VYVOLÁNÍ PŘERUŠENÍ

Dalším typem měření bylo zjištění latence mezi příchozím signálem na vstupní pin GPIO a změnou hodnoty na výstupním pinu GPIO. Pro toto měření bylo opět použito zařízení Analog Discovery 2. To sloužilo jednak jako generátor obdélníkového signálu s periodou 100 μs ale také jako osciloskop pro měření latence.

Princip měření spočíval v tom, že při příchodu vzestupné hrany signálu z generátoru se vyvolala obsluha přerušení, která probudila spící proces. Tento proces poté nastavil požadovanou hodnotu na výstupním pinu GPIO a byl opět uspán. Uspání procesu a obsluha přerušení je v této práci řešena vytvořením samostatného Linuxového modulu. Tento modul bude podrobněji popsán v následující kapitole a součástí tohoto modulu je i character device, který slouží k uspání procesu. Proces uspání a probuzení procesu je popsán v kapitole 6.3.

Dále byla k proměření využita modifikovaná aplikace z kapitoly 5.2. Celý kód pro tuto aplikaci je dostupný v příloze D.

8.1 Linux modul pro obsluhu přerušení a uspání procesu

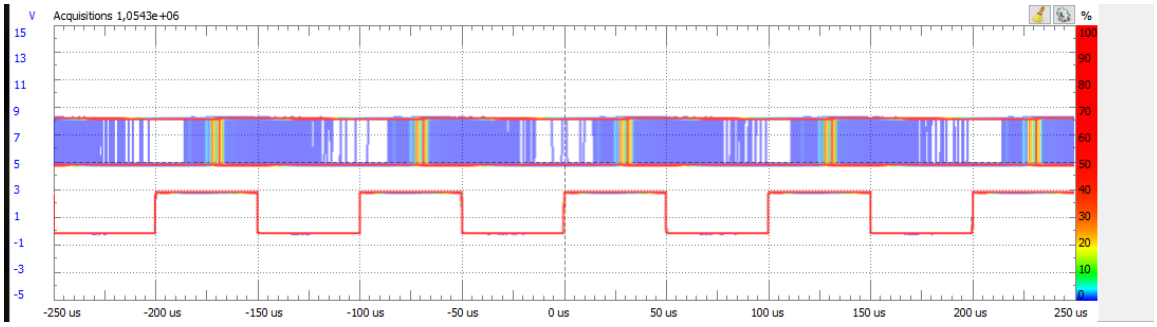
Jedná se o kód, který běží přímo v Linux jádře a lze jej do jádra přidat nebo odebrat bez nutnosti restartovat systém. [17]

Pro obsluhu přerušení obsahuje modul funkcí Input_ISR, která se stará o probuzení procesu na základě příchodu signálu na vstupní pin GPIO. Probuzení procesu probíhá pomocí workqueues a funcke wake_up_interruptible. Naopak uspání je realizováno pomocí funkce sleeepy_read. Tato funkce uspí každý proces, který se pokusí číst z character device souboru. Modul také umožňuje probouzet proces pomocí funkce sleepy_open, která proces uspí, pokud se pokusí otevřít character device soubor. Skutečnost, jestli se proces uspí pomocí čtení ze souboru nebo tím, že se soubor pokusí otevřít, se nastaví pomocí makra SLEEP_BY_READ. Pokud je toto makro definováno, je proces uspán pomocí funkce sleep_read. Pokud definováno není, je proces uspán pomocí funkce sleep_open. Celý kód je k dispozici v příloze E. Části kódu jsou převzaty z [18], [19] a [20].

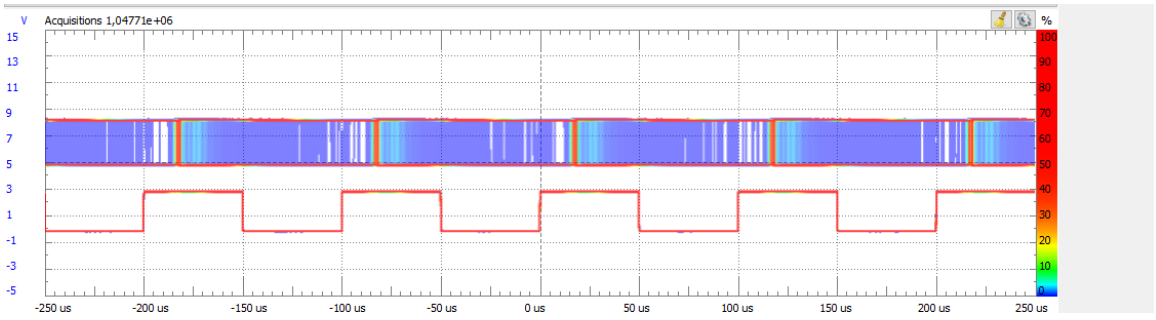
8.2 Výsledky měření

Celkově byly provedeny čtyři měření. První sada měření byla pro jednotlivé plánovací politiky. Konkrétně pro politiku typu FIFO a RR. Druhá sada měření se zabývala vlivem vytížení systému na latenci. Výsledky jsou shrnuty na obrazcích: 8.2.1, 8.2.2, 8.2.3 a 8.2.4. Zatížení systému bylo simulováno spuštěním programu cyclictest.

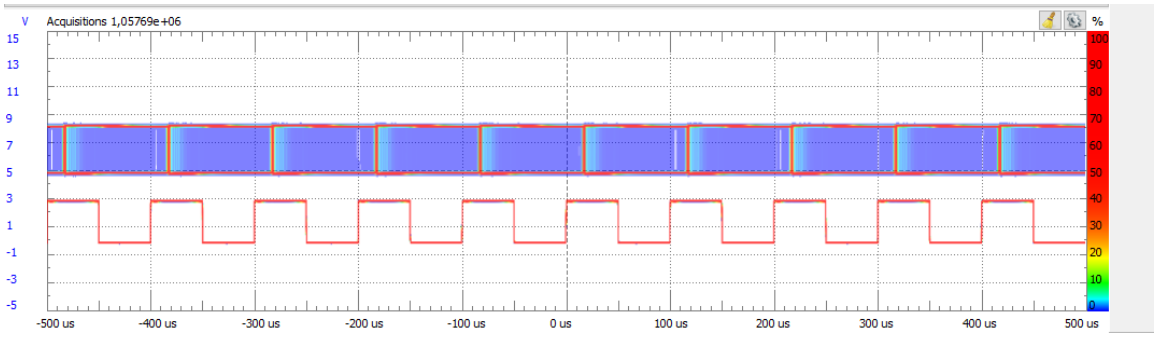
Oranžový průběh v grafech zobrazuje stav výstupního pinu GPIO a modrý průběh zobrazuje pulzy generované na vstupním pinu GPIO.



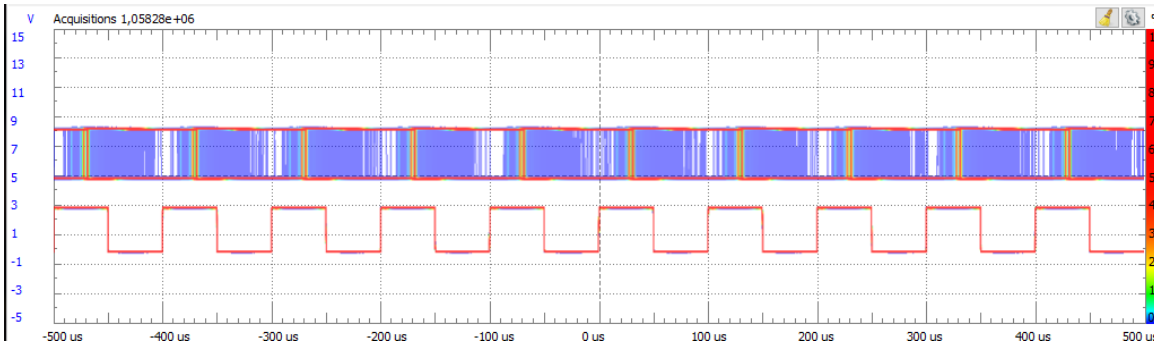
Obrázek 8.2.1 Výsledek měření latence na základě vyvolání přerušení pro plánovací politiku typu FIFO bez zatíženého systému



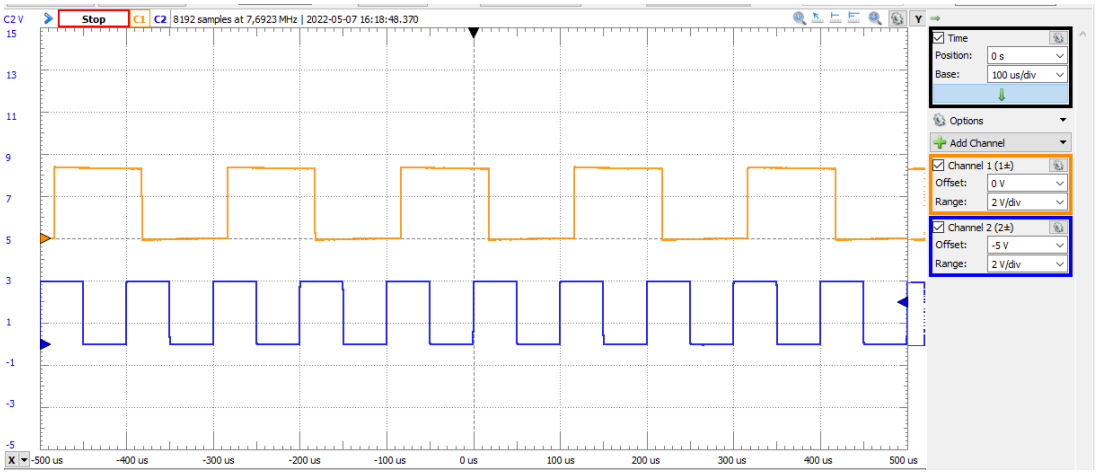
Obrázek 8.2.2 Výsledek měření latence na základě vyvolání přerušení pro plánovací politiku typu FIFO se zatěží systému



Obrázek 8.2.3 Výsledek měření latence na základě vyvolání přerušení pro plánovací politiku typu RR se zatěží systému



Obrázek 8.2.4 Výsledek měření latence na základě vyvolání přerušení pro plánovací politiku typu RR bez zatíženého systému



Obrázek 8.2.5 Ukázka průběhu jednotlivých signálů na osciloskopu

Z obrázků je patrné, že nejmenší rozptyl vzorků je při použití plánovací politiky typu FIFO a bez zatíženého systému.

8.3 Seznam použitých přístrojů a nastavení Analog Discovery 2 pro Měření latence na základě vyvolání přerušení

Použité přístroje:

- *Raspberry Pi 4B 2GB RAM, FCC ID: 2ABCB-RPI4B, IC ID: 20953-RPI4B*
- *Analog Discovery 2, SN: 210321ABEA72*

Nastavení osciloskopu v programu WaveForms:

- *Source:* *Channel 2*
- *Type:* *Edge*
- *Condition:* *Rissing*
- *Level:* *2 V*

Nastavení generátoru v programu WaveForms:

- *Source:* *Channel 1*
- *Type:* *Pulse*
- *Frequency:* *10 kHz*
- *Period:* *100 us*
- *Amplitude:* *3 V*
- *Offset:* *0 V*
- *Symetry:* *50 %*
- *Phase:* *0 °*

9. ZÁVĚR

V práci bylo popsáno, jak lze z operačního systému Linux udělat real-time systém. Hlavní důraz byl kladen na PREEMPT_RT patch.

Cílem práce bylo popsat, jak aplikovat PREEMPT_RT patch a měření latence systému. V práci jsou také popsané metody, podle kterých operační systém vybírá procesy, kterým bude přidělen procesor. Cílem bylo zjistit, zda volba těchto metod má vliv na latenci systému.

První metodou pro měření latence byl program cyclictest a script mklatencyplot pro vytvoření histogramu z naměřených hodnot. Tímto měřením se porovnávala latence na Linux jádře bez a s PREEMPT_RT patchem. Výsledky jsou shrnuty na obrázcích 7.1.1 a 7.1.2. Z grafů je patrné, že aplikací PREEMPT_RT patche se maximální latence snížila o 58 μs.

Další měření se věnovalo tomu, jaký vliv má volba plánovací politika na latenci. K ověření byla použita RT aplikace, která řídila výstupní pin GPIO umístěném na Raspberry Pi. Měření bylo provedeno pomocí zařízení Analog Discovery 2 a programu WaveForms. Z naměřených výsledků vyplývá, že největší vliv na latenci měla plánovací politika FIFO a RR. Nejlepší výsledek dávala plánovací politika typu deadline.

Posledním typem měření, který je v práci zpracován, je měření časového rozdílu mezi příchozím signálem na vstupní pin a nastavením výstupního pinu. Tyto výsledky jsou shrnuty na obrazcích 8.2.1, 8.2.2, 8.2.3 a 8.2.4. Z těchto výsledků vyplývá, že nejlepšího výsledku bylo dosaženo při plánovací politice FIFO bez zatíženého systému.

Možným zlepšením latence do budoucna by mohlo být využití nástrojů, které umožňují zjistit, jaké ovladače mají největší vliv na latenci systému. A pokud to bude možné, tak je ze systému odstranit. Další možností, jak zlepšit latenci je izolování jednoho jádra CPU tak, aby na něm běžela pouze RT aplikace a nezbytné procesy nutné pro komunikaci s operačním systémem.

Literatura

[1] Karim Yaghmour, Jon Mastrers, Gilad Ben-Yossef, Philipe Gerum. Building Embedded Linux Systems. 2008, Second edition. Sebastopol: O'Reilly ISBN 978-0-596-52968-0

[2] Jan Altenberg, linutronix GmbH. Introduction to Realtime Linux [online]. 1.1.2022, Dostupné z: https://youtu.be/BKkX9WASfpI

[3] Vít Pelčák, Real-time modifikace Linux – 1 (RTLinux a RTAI) [online]. 1.1.2022, Dotupné z: https://www.abclinuxu.cz/clanky/system/real-time-modifikace-linuxu-1-rtlinux-a-rtai

[4] Pierre Ficheux, Using real-time with Linux [online], 1.1.2022. Dostupné z: https://etr2021.ensma.fr/files/p-ficheux-realtime-linux.pdf

[5] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, 2005, 3rd Edition. Sebastopol: O'Reilly ISBN 978-0596005658

[6] Introduction to the Raspberry Pi GPIO and Physical Computing [online], 1.1.2022. Dostupné z: https://learn.sparkfun.com/tutorials/introduction-to-the-raspberry-pi-gpio-and-physical-computing/gpio-pins-overview

[7] Rasbperry Pi [online]. 1.1.2022, Dostupné z: https://cs.wikipedia.org/wiki/Raspberry_Pi

[8] Raspberry Pi Documantion [online]. 1.1.2022, Dostupné z: https://www.raspberrypi.com/documentation/computers/os.html#gpio-and-the-40-pin-header

[9] J.Lažánský, Plánování procesů [online], 1.1.2022, Dostupné z: http://labe.felk.cvut.cz/vyuka/A3B33OSD/Tema-04-Planovani-OSD.pdf

[10] Daniel Bristot de Oliveira, Deadline scheduling part 1 – overview and theory [online]. 1.1.2022, Dostupné z: https://lwn.net/Articles/743740/

[11] HOWTO build a simple RT application [online]. 1.1.2022, Dostupné z: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base

[12] HOWTO build a basic cyclic application [online]. 1.1.2022, Dostupné z: https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/cyclic

[13] Deadline Task Scheduling [online]. 1.1.2022, Dostupné z: https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html

[14] Cyclictest [online]. 1.1.2022, Dotupné z: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[15] mklatencyplot [online]. 1.1.2022, Dostupné z: https://www.osadl.org/uploads/media/mklatencyplot.bash

[16] sched(7) – Linux manual page [online]. 1.1.2022, Dostupné z: https://man7.org/linux/man-pages/man7/sched.7.html

[17] Kernel module [online]. 10.5.2022, Dostupné z: https://wiki.archlinux.org/title/Kernel_module

[18] a simple linux driver code to register GPIO as button input to trigger LED [online]. 10.3.2022, Dostupné z: https://gist.github.com/itrobotics/224a0c549ae073ce991c#file-btn_led-c
[19] sleepy.c [online]. 10.3.2022, Dostupné z: https://github.com/fffaraz/kernel/blob/master/ldd3-examples/examples/misc-modules/sleepy.c
[20] Device File Creation – Linux Device Driver Tutorial Part 5 [online]. 5.4.2022, Dostupné z: https://embetronicx.com/tutorials/linux/device-drivers/device-file-creation-for-character-drivers/
[21] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers. 2005, Third Edition. United States of America: O'Reilly ISBN 0-596-00590-3
[22] How are semaphores implemented in the Linux kernel? [online]. 20.4.2022, Dostupné z: https://www.quora.com/How-are-semaphores-implemented-in-the-Linux-kernel
[23] Spin Locks [online]. 20.4.2022, Dostupné z: https://flylib.com/books/en/4.168.1.35/1/
[24] Mutex lock for Linux Thread Synchronization [online]. 20.4.2022, Dostupné z: https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

SEZNAM SYMBOLŮ A ZKRATEK

Zkratky:

GPIO	general-purpose input/ouput
RT	Real-time
OS	Operační systém
HAL	Hardware abstarction layer (hardwarová abstrakní vrstva)
RTAI	Real time application Interface
IRQ	Interrupt ReQuest (Požadavek na přerušení)
IRS	Interrupt service rutine (obslužná rutina přerušení)
FIFO	First in first out
RR	round-robin

SEZNAM PŘÍLOH

Příloha A - Postup při kompilaci Linux jádra pro Raspberry Pi 64 bit OS + PREEMPT_RT patch

1. V home vytvoření složky opt a ve složce opt složku aarch64
2. Do opt/aarch64 stažení binutils pomocí:
   wget https://ftp.gnu.org/gnu/binutils/binutils-2.37.tar.bz2
3. Rozbalení binutils tar xf binutils-2.37.tar.bz2
4. Instalace a konfigurace binutils: mkdir binutils-obj
5. cd binutils-obj
6. ../binutils-2.34/configure --prefix=/opt/aarch64 --target=aarch64- linux-gnu --disable-nls
7. make -j4
8. sudo make install
9. Nastavení proměnné PATH: export PATH=$PATH:/opt/aarch64/bin/
10. Stažení gcc: wget https://ftp.gnu.org/gnu/gcc/gcc-11.2.0/gcc-11.2.0.tar.xz
11. Rozbalení gcc: tar xf gcc-11.2.0.tar.xz
12. instalace gcc: mkdir gcc-out
13. cd gcc-out
14. ../gcc-8.4.0/configure --prefix=/opt/aarch64 --target=aarch64-linux-gnu --with-newlib --without-headers \
    --disable-nls --disable-shared --disable-threads --disable-libssp - - disable-decimal-float \
    --disable-libquadmath --disable-libvtv --disable-libgomp --disable-libatomic \
    --enable-languages=c
15. make all-gcc -j4
16. sudo make install-gcc
17. instalace libgcc: make all-target-libgcc
18. make install-target-libgcc
19. Stažení kernel zdroje: https://github.com/raspberrypi/linux/releases : verze: raspberrypi-kernel_1.20210527-1
20. Rozbalení kernel zdroje: tar xf raspberrypi-kernel_1.20200212-1.tar.gz
21. stažení soubru:
    git clone --depth=1 -b rpi-5.4.y https://github.com/raspberrypi/linux.git
22. vytvoření složek: kernel-rt, modules-rt a dtbs-rt
23. pomocí export nastavit proměné: ARCH=arm64, KERNEL=kernel8, INSTALL_MOD_PATH=~/opt/aarch64/modules-rt,

INSTALL_DTBS_PATH=~/opt/aarch64/dtbs-rt a CROSS_BUILD=aarch64-linux-gnu-

24. pro odpovídající verzi jádra stáhnout RT patch z: https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/
25. ve složce linux použít příkaz: `zcat patch-<verze patche>.patch.gz | patch -p1`
26. vytvoření defconfig: make O=../kernel-rt/ bcmrpi3_defconfig (Pro Raspberry Pi 4 bcm2711_defconfig)
27. Pro uprávu konfigurace: make O=../kernel-rt/ menuconfig : V general setup přidat k verzi jádra vlastní koncovku a v general setup -→ `Preemption Model zvolit Fully Preemptible Kernel (Real-Time)`
28. Sestavení kernelu: make -j4 Image modules dtbs O=../kernel-out/
29. Instalace modulů a DTBs: make O=../kernel-rt/ modules_install dtbs_install
30. Složky kernel-rt, modules-rt a dtbs-rt nahrát na Raspberry Pi
31. Zkopírovat:sudo cp ~/kernel-rt/arch/arm64/boot/Image /boot/kernel8.img, přesun do složky ~/modules-rt/lib a provést sudo cd -r <Verze jádra> /lib/modules, přesun do složky ~/dtbs-rt/overlays a provést sudo cp -d * /boot/overlays, pomocí cd ../broadcom se přesunout do složky broadcom a provést sudo cp -d bcm* /boot/
32. V /boot/config.txt nastavit kernel image: kernel=kernel8.img
33. Reboot

Postup převzat z: **https://www.tal.org/tutorials/raspberry-pi3-build-64-bit-kernel** a https://www.tal.org/tutorials/booting-64-bit-kernel-raspberry-pi-3

Příloha B - Zdrojový kód RT aplikace s RR a FIFO plánovací politikou

```
/*
 * RT_proj.c
 *
 * Copyright 2021  <pi@raspberrypi>
 *
 *  This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License as published
by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301, USA.
 *
 *
 */


#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <gpiod.h>
#include <limits.h>


struct period_info{
            struct timespec next_period;
            long period_ns;
};

static void inc_period(struct period_info *pinfo);
static void periodic_task_init(struct period_info *pinfo);
static void wait_rest_of_period(struct period_info *pinfo);
void *thread_func(void *data);

int main(int argc, char *argv[])
{
      struct sched_param param;
      pthread_attr_t attr;
      pthread_t thread;
      int ret;
      if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
      {
```

```
            printf("mlockall fauled: %m\n");
            exit(-2);
      }

      ret = pthread_attr_init(&attr);
      if(ret)
      {
            printf("init pthread attribute failed\n");
            goto out;
      }

      ret = pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
      if(ret)
      {
            printf("pthread stacksize failed\n");
            goto out;
      }

      ret = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
      if(ret)
      {
            printf("pthread setschedpolicy failed\n");
            goto out;
      }

      param.sched_priority = 80;
      ret = pthread_attr_setschedparam(&attr, &param);
      if(ret)
      {
            printf("pthread setschedparam failed\n");
      }

      ret             =             pthread_attr_setinheritsched(&attr,
PTHREAD_EXPLICIT_SCHED);
      if(ret)
      {
            printf("pthread setinheritsched faild/n");
            goto out;
      }

      ret = pthread_create(&thread, &attr, thread_func, NULL);
      if(ret)
      {
            printf("create pthread failed\n");
            goto out;
      }

      ret = pthread_join(thread, NULL);
      if(ret)
      {
            printf("join pthread failed\n");
      }

out:
      return ret;
}


static void wait_rest_of_period(struct period_info *pinfo)
```

```
{
            inc_period(pinfo);
            clock_nanosleep(CLOCK_MONOTONIC,   TIMER_ABSTIME,   &pinfo-
>next_period, NULL);
}

static void inc_period(struct period_info *pinfo)
{
            pinfo->next_period.tv_nsec += pinfo->period_ns;
            while(pinfo->next_period.tv_nsec >= 1000000000)
            {
                  pinfo->next_period.tv_sec++;
                  pinfo->next_period.tv_nsec -= 1000000000;
            }

}

static void periodic_task_init(struct period_info *pinfo)
{
      pinfo->period_ns = 1000000;
      clock_gettime(CLOCK_MONOTONIC, &(pinfo->next_period));
}

void *thread_func(void *data)
{
            struct period_info pinfo;

            periodic_task_init(&pinfo);
            const char *chipname="gpiochip0";
            struct gpiod_chip *chip;
            struct gpiod_line *lineRed;

            chip = gpiod_chip_open_by_name(chipname);
            lineRed = gpiod_chip_get_line(chip,18);
            gpiod_line_request_output(lineRed,"example1", 0);

            int i = 0;
            while(1)
            {
            if (i % 2 == 0)
                  gpiod_line_set_value(lineRed, 1);
            else
                  gpiod_line_set_value(lineRed, 0);


            wait_rest_of_period(&pinfo);
            ++i;
            if(i == 50000000)
                  break;
            }
            gpiod_line_release(lineRed);
            gpiod_chip_close(chip);

            return NULL;
}
```

Příloha C - Zdrojový kód RT aplikace s Deadline plánovací politikou

```
/* RT_proj.c
 *
 * Copyright 2021  <pi@raspberrypi>
 *
 *  This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License as published
by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301, USA.
 *
 *
 */

#define _GNU_SOURCE_
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <gpiod.h>
#include <limits.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <time.h>

#define gettid() syscall(__NR_gettid)
#define SCHED_DEADLINE 6

/* XXX use the proper syscall numbers */
#ifdef __x86_64__
#define __NR_sched_setattr           314
#define __NR_sched_getattr           315
#endif

#ifdef __i386__
#define __NR_sched_setattr           351
#define __NR_sched_getattr           352
#endif

#ifdef __arm__
```

```
#define __NR_sched_setattr           380
#define __NR_sched_getattr           381
#endif

struct sched_attr {
     __u32 size;

     __u32 sched_policy;
     __u64 sched_flags;

     /* SCHED_NORMAL, SCHED_BATCH */
     __s32 sched_nice;

     /* SCHED_FIFO, SCHED_RR */
     __u32 sched_priority;

     /* SCHED_DEADLINE (nsec) */
     __u64 sched_runtime;
     __u64 sched_deadline;
     __u64 sched_period;
};

int sched_setattr(pid_t pid, const struct sched_attr *attr, unsigned
int flags)
{
     return syscall(__NR_sched_setattr, pid, attr, flags);
}

int sched_getattr(pid_t pid, struct sched_attr *attr, unsigned int
size, unsigned int flags)
{
     return syscall(__NR_sched_getattr, pid, attr, size, flags);
}

void *run_deadline(void *data)
{
     struct sched_attr attr;
     int ret;
     unsigned int flags = 0;

     printf("deadline thread started [%ld]\n", gettid());

     attr.size = sizeof(attr);
     attr.sched_flags = 0;
     attr.sched_nice = 0;
     attr.sched_priority = 0;


     attr.sched_policy = SCHED_DEADLINE;
     attr.sched_runtime = 20*1000*1000;
     attr.sched_deadline = 50*1000*1000;
     attr.sched_period = 100*1000*1000;
     //attr.sched_period = attr.sched_deadline = 120*1000*1000;

     ret = sched_setattr(0, &attr, flags);
     if (ret < 0) {
             perror("sched_setattr");
             exit(-1);
     }
```

```
      const char *chipname="gpiochip0";
            struct gpiod_chip *chip;
            struct gpiod_line *lineRed;

            chip = gpiod_chip_open_by_name(chipname);
            lineRed = gpiod_chip_get_line(chip,18);
            gpiod_line_request_output(lineRed,"example1", 0);

            int i = 0;
     while (1) {
            if (i % 2 == 0)
                  gpiod_line_set_value(lineRed, 1);
            else
                  gpiod_line_set_value(lineRed, 0);

            ++i;
            if(i == 5000000000)
                  break;
     }
       gpiod_line_release(lineRed);
       gpiod_chip_close(chip);
     printf("deadline thread dies [%ld]\n", gettid());
     return NULL;
}


/*
struct period_info{
            struct timespec next_period;
            long period_ns;
};
*/
//static void inc_period(struct period_info *pinfo);
//static void periodic_task_init(struct period_info *pinfo);
//static void wait_rest_of_period(struct period_info *pinfo);
//void *thread_func(void *data);

int main(int argc, char *argv[])
{
      pthread_t thread;
    printf("main thread [%ld]\n", gettid());

    pthread_create(&thread, NULL, run_deadline, NULL);

    pthread_join(thread, NULL);


      printf("main dies [%ld]\n", gettid());

    return 0;
/*
      if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1)
      {
            printf("mlockall fauled: %m\n");
            exit(-2);
      }
      ret = pthread_attr_init(&attr);
      if(ret)
```

```
	{
		printf("init pthread attribute failed\n");
		goto out;
	}

	ret = pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
	if(ret)
	{
		printf("pthread stacksize failed\n");
		goto out;
	}

	ret = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
	if(ret)
	{
		printf("pthread setschedpolicy failed\n");
		goto out;
	}

	//param.sched_priority = 80;
	ret = pthread_attr_setschedparam(&attr, &param);
	if(ret)
	{
		printf("pthread setschedparam failed\n");
	}

	ret=pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
	if(ret)
	{
		printf("pthread setinheritsched faild/n");
		goto out;
	}

	ret = pthread_create(&thread, &attr, thread_func, NULL);
	if(ret)
	{
		printf("create pthread failed\n");
		goto out;
	}

	ret = pthread_join(thread, NULL);
	if(ret)
	{
		printf("join pthread failed\n");
	}

out:
	return ret;
	* */
}

/*
static void wait_rest_of_period(struct period_info *pinfo)
{
		inc_period(pinfo);
		clock_nanosleep(CLOCK_MONOTONIC,   TIMER_ABSTIME,   &pinfo-
>next_period, NULL);
}
```

```
static void inc_period(struct period_info *pinfo)
{
            pinfo->next_period.tv_nsec += pinfo->period_ns;
            while(pinfo->next_period.tv_nsec >= 1000000000000)
            {
                  pinfo->next_period.tv_sec++;
                  pinfo->next_period.tv_nsec -= 1000000000000;
            }

}
*/
/*static void periodic_task_init(struct period_info *pinfo)
{
      pinfo->period_ns = 100000000;
      clock_gettime(CLOCK_MONOTONIC, &(pinfo->next_period));
}
*/
/*
void *thread_func(void *data)
{
            //struct period_info pinfo;

            //periodic_task_init(&pinfo);
            const char *chipname="gpiochip0";
            struct gpiod_chip *chip;
            struct gpiod_line *lineRed;

            chip = gpiod_chip_open_by_name(chipname);
            lineRed = gpiod_chip_get_line(chip,18);
            gpiod_line_request_output(lineRed,"example1", 0);

            int i = 0;
            while(1)
            {
            if (i % 2 == 0)
                  gpiod_line_set_value(lineRed, 1);
            else
                  gpiod_line_set_value(lineRed, 0);

            ++i;
            if(i == 5000000000)
                  break;
            }
            gpiod_line_release(lineRed);
            gpiod_chip_close(chip);

            return NULL;
}
*/
```

Příloha D - Zdrojový kód RT aplikace s RR a FIFO plánovací politikou pro měření latence na základě příchozího přerušení

```
/*
 * RT_proj.c
 *
 * Copyright 2021  <pi@raspberrypi>
 *
 *  This program is free software; you can redistribute it and/or
modify
 * it under the terms of the GNU General Public License as published
by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
 * MA 02110-1301, USA.
 *
 *
 */


#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <gpiod.h>
#include <limits.h>
#include <stdbool.h>

struct period_info{
            struct timespec next_period;
            long period_ns;
};


void *thread_func(void *data);

int main(int argc, char *argv[])
{
      struct sched_param param;
      pthread_attr_t attr;
      pthread_t thread;
      int ret;
```

```
      if(mlockall(MCL_CURRENT | MCL_FUTURE) == -1){
            printf("mlockall fauled: %m\n");
            exit(-2);
      }

      ret = pthread_attr_init(&attr);
      if(ret){
            printf("init pthread attribute failed\n");
            goto out;
      }

      ret = pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
      if(ret){
            printf("pthread stacksize failed\n");
            goto out;
      }

      ret = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
      if(ret){
            printf("pthread setschedpolicy failed\n");
            goto out;
      }

      param.sched_priority = 90;
      ret = pthread_attr_setschedparam(&attr, &param);
      if(ret){
            printf("pthread setschedparam failed\n");
            goto out;
      }

      ret             =             pthread_attr_setinheritsched(&attr,
PTHREAD_EXPLICIT_SCHED);
      if(ret){
            printf("pthread setinheritsched faild/n");
            goto out;
      }

      ret = pthread_create(&thread, &attr, thread_func, NULL);
      if(ret){
            printf("create pthread failed\n");
            goto out;
      }

      ret = pthread_join(thread, NULL);
      if(ret){
            printf("join pthread failed\n");
      }

out:
      return ret;
}

void *thread_func(void *data){
            const char *chipname="gpiochip0";
            struct gpiod_chip *chip;
            struct gpiod_line *lineRed;

            chip = gpiod_chip_open_by_name(chipname);
```

```
            lineRed = gpiod_chip_get_line(chip,18);
            gpiod_line_request_output(lineRed,"example1", 0);

            int i = 0;
            int READ;
            bool run = true;

            while(run){
                  FILE *frw = fopen("//dev//etx_device","r+");
                  if(frw == NULL){
                        printf("Open file failn \n");
                        return NULL;
                  }
                  fscanf(frw, "%d", &READ);
                  if (i % 2 == 0)
                        gpiod_line_set_value(lineRed, 0);
                  else
                        gpiod_line_set_value(lineRed, 1);

                  ++i;
            fclose(frw);
                  if(i == 5000000000)
                        run = false;
            }

      gpiod_line_release(lineRed);
      gpiod_chip_close(chip);

      return NULL;
}
```

Příloha E - Linux modul pro obsluhu přerušení

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/wait.h>
#include <linux/device.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>

#define ISR 24
#define MY_GPIO_INT_NAME "RPi"
#define MY_DEV_NAME "IRQ_Module"
#define SLEEP_BY_READ

MODULE_LICENSE ("GPL");

static short int trigger_input = 0;
static volatile unsigned long flags = 0;
static int sleepy_major = 0;

static DECLARE_WAIT_QUEUE_HEAD(wq);

static struct class *dev_class;
static struct cdev etx_cdev;
dev_t dev = 0;

// Reader awake

static irqreturn_t Input_ISR (int irq, void* data){
      local_irq_save(flags);
      flags = 1;
      wake_up_interruptible(&wq);
      local_irq_restore(flags);
      return IRQ_HANDLED;
}

ssize_t  sleepy_read  (struct  file  *filp,  char  __user  *buf,  size_t
count, loff_t *pos){
#ifdef SLEEP_BY_READ
      flags = 0;
      wait_event_interruptible(wq, flags != 0);
      flags = 0;
#endif
      return 0; /* EOF */
}

static int sleepy_open (struct inode *inode,struct file *filp){
#ifndef SLEEP_BY_READ
      flags = 0;
      wait_event_interruptible(wq, flags != 0);
      flags = 0;
```

```
#endif
      return 0; /* EOF */
}

struct file_operations sleepy_fops = {
      .read =  sleepy_read,
      .open =  sleepy_open,
      .owner = THIS_MODULE
};

int __init device_init(void){
      int ret = 0;
// Init char. device
      if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
            pr_err("Cannot allocate major number for device\n");
             return -1;
        }
    pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

      cdev_init(&etx_cdev, &sleepy_fops);
      if((cdev_add(&etx_cdev,dev,1)) < 0)
                  goto r_class;

        /*Creating struct class*/
    if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
            pr_err("Cannot create the struct class for device\n");
            goto r_class;
        }

        /*Creating device*/
    if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
            pr_err("Cannot create the Device\n");
            goto r_device;
        }
    pr_info("Kernel Module Inserted Successfully...\n");
   // Init GPIO
      ret = gpio_is_valid(ISR);
      if (ret < 0)
            goto clean;
      ret = gpio_request(ISR,"Input");
      if (ret < 0)
            goto clean;
      trigger_input = gpio_to_irq(ISR);
      if(trigger_input < 0)
            goto clean;
      if(request_irq(trigger_input,Input_ISR,IRQF_TRIGGER_RISING,
MY_GPIO_INT_NAME, MY_DEV_NAME))
            goto clean;
      return 0;
clean:
      printk(KERN_DEBUG "insert fail\n");
      return -1;
r_class:
      unregister_chrdev_region(dev,1);
      return -2;
r_device:
      class_destroy(dev_class);
      return -3;}
```

```
void __exit clean_module(void){
      free_irq(trigger_input, MY_DEV_NAME);
      gpio_free(ISR);
      unregister_chrdev_region(dev,1);
      device_destroy(dev_class,dev);
      class_destroy(dev_class);
}

module_init(device_init);
module_exit(clean_module);
```