

Plánovače procesů v Linuxu

Tomáš Veselý

Bakalářská práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Tomáš Veselý
Osobní číslo:	A21126
Studijní program:	B0613A140020 Softwarové inženýrství
Forma studia:	Kombinovaná
Téma práce:	Plánovače procesů v Linuxu
Téma práce anglicky:	Process Schedulers in Linux

Zásady pro vypracování

- Vytvořte literární rešerši na dané téma.
- Vysvětlete význam a funkci plánování procesů.
- Definujte, jak je plánovač procesů ovlivňován současnou architekturou desktopových CPU.
- Popište princip aktuálně využívaného plánovače CFS (Completely Fair Scheduler) a nového plánovače EEVDF (Earliest Eligible Virtual Deadline First).
- Proveďte srovnání plánovačů pro různé situace použití.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. BOVET, Daniel Pierre a CESATI, Marco, 2006. *Understanding the Linux kernel*. 3rd ed. Sebastopol: O'Reilly. ISBN 05-960-0565-2.
2. VOZÁR, Jiří, 2019. *Performance Testing of Linux Kernel Scheduler*. Online, Bakalářská práce, vedoucí Ing. Malík Viktor. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: <https://www.fit.vut.cz/study/thesis-file/21469/21469.pdf>. [cit. 2024-10-23].
3. LINUX KERNEL ORGANIZATION, 2024. *The Linux Kernel documentation*. Online. Dostupné z: <https://docs.kernel.org/>. [cit. 2024-10-23].
4. LEUNG, Joseph (ed.), 2004. *Handbook of Scheduling Algorithms, Models, and Performance Analysis*. Chapman & Hall. ISBN 1-58488-397-9.
5. GREGG, Brendan. *Systems performance: enterprise and the cloud*. Second edition. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2021. ISBN 01-368-2015-8.

Vedoucí bakalářské práce: **doc. Ing. Martin Sysel, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **27. října 2024**
Termín odevzdání bakalářské práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA BAKALÁŘSKÉ PRÁCE

Beru na vědomí, že

- odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení bakalářské práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – bakalářskou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na bakalářské práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....

podpis autora

ABSTRAKT

Bakalářská práce se zabývá problematikou plánování procesů v operačním systému Linux. Teoretická část popisuje základy plánování procesů. Je popsán vývoj a současný stav plánovacích algoritmů. Praktická část se zaměřuje na testování rozdílného chování plánovačů CFS a EEVDF z pohledu latence. Má za cíl objektivní posouzení vlastností obou algoritmů.

Klíčová slova: plánovač, proces, Linux, CFS, EEVDF

ABSTRACT

The bachelor thesis deals with the issue of process scheduling in the Linux operating system. The theoretical part introduces the basics of process scheduling. The development and current state of scheduling algorithms is described. The practical part focuses on testing the different latency behavior of CFS and EEVDF. It aims to objectively assess the properties of both algorithms.

Keywords: scheduler, process, Linux, CFS, EEVDF

Tímto chci poděkovat vedoucímu bakalářské práce panu doc. Ing. Martinu Syslovi, Ph.D. za jeho odborné vedení, cenné připomínky a za jeho čas při řešení dané problematiky.

OBSAH

ÚVOD.....	11
I TEORETICKÁ ČÁST	12
1 TEORETICKÝ ZÁKLAD PLÁNOVÁNÍ PROCESŮ	13
1.1 PROCES	13
1.1.1 Životní cyklus procesu	14
1.2 VLÁKNA.....	15
1.2.1 POSIX vlákna.....	16
1.3 PLÁNOVAČ PROCESŮ	17
1.4 PŘEPÍNÁNÍ KONTEXTU	18
1.5 PREEMPTIVNÍ PLÁNOVÁNÍ	18
1.6 PRIORITA A ČASOVÉ KVANTUM	19
1.7 TRÍDY PLÁNOVÁNÍ	19
1.7.1 Stop	20
1.7.2 EDF – Early Deadline First.....	20
1.7.3 RT – Real-Time.....	20
1.7.4 Třída „Fair“	21
1.7.5 Idle.....	22
2 ARCHITEKTURA CPU A JEJÍ VLIV NA PLÁNOVÁNÍ PROCESŮ	23
2.1 SIMULTÁNNÍ MULTITHREADING (SMT).....	23
2.2 NEROVNOMĚRNÝ PŘÍSTUP DO PAMĚTI (NUMA, CPU AFFINITY)	23
2.3 HETEROGENNÍ ARCHITEKTURA CPU (P-CORE / E-CORE).....	24
2.4 DYNAMICKÉ ŘÍZENÍ TAKTU (TURBO BOOST).....	24
2.5 BEZPEČNOSTNÍ HROZBY A MITIGACE.....	25
3 PLÁNOVAČE PROCESŮ.....	26
3.1 $O(N)$	26
3.2 $O(1)$	27
3.3 ALTERNATIVNÍ PLÁNOVAČE	28
3.3.1 RSDL.....	28
3.3.2 BFS.....	29
3.3.3 MuQSS.....	29
3.4 CFS.....	30
3.4.1 Princip CFS	31
3.4.2 Implementace CFS	31
3.4.3 Výpočetní složitost a efektivita využití výkonu.....	34
3.5 EEVDF	34
3.5.1 Princip EEVDF	35

3.5.2	Implementace EEVDF	37
II	PRAKTICKÁ ČÁST	39
4	PROVEDENÍ TESTŮ PLÁNOVAČŮ	40
4.1	PŘÍPRAVA TESTOVACÍHO PROSTŘEDÍ	40
4.2	POUŽITÉ NÁSTROJE	41
4.2.1	Hackbench	41
4.2.2	Cyclictest	42
4.3	POSTUP TESTOVÁNÍ A MĚŘÍCÍ METODIKA	43
4.3.1	Latence a počet jader	44
4.3.2	Latence s parametrem nice	49
4.3.3	Konzistence latence	51
4.4	ANALÝZA VÝSLEDKŮ	53
	ZÁVĚR.....	55
	SEZNAM POUŽITÉ LITERATURY	56
	SEZNAM OBRÁZKŮ	60
	SEZNAM TABULEK.....	61
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	62

ÚVOD

Plánování procesů patří mezi klíčové úkoly každého operačního systému a zásadním způsobem ovlivňuje jeho výkon, stabilitu a použitelnost. Jde o mechanismus, kterým operační systém rozhoduje o tom, kdy a na jak dlouho jednotlivé procesy získají přístup k procesoru. Vhodně zvolený plánovací algoritmus dokáže nejen optimalizovat využití systémových prostředků, ale i zajistit rychlou odezvu aplikací a s tím spojenou plynulost práce uživatelů. Operační systém Linux, který je dnes více než kdy jindy používán na desktopových nebo serverových platformách, disponuje mechanismy plánování procesů, které se právě v současné době dočkaly zásadních změn v podobě nového plánovače EEVDF. Postupný vývoj a zdokonalování těchto plánovacích algoritmů reaguje na rychlé změny v hardwarových architekturách i na nové požadavky aplikací, které dnes vyžadují například snadnou škálovatelnost prostředí.

Cílem této práce je představit problematiku plánování procesů v Linuxu jako celek. Od teoretických principů a základních pojmů až po praktické aspekty ovlivňující každodenní využívání operačního systému. Práce se zaměřuje zejména na moderní přístupy k problematice a aktuálně používané algoritmy plánování, které určují, jak efektivně dokáže Linux uspokojit náročné požadavky dnešních uživatelů.

I TEORETICKÁ ČÁST

1 TEORETICKÝ ZÁKLAD PLÁNOVÁNÍ PROCESŮ

Jednou z klíčových funkcí operačního systému je rozhodování, kolik, kdy a v jakém pořadí se přidělí procesorový čas procesům, které čekají na spuštění. Zvolený algoritmus plánování ovlivňuje propustnost systému (kolik procesů dokáže systém úspěšně dokončit za určitý časový úsek), režii samotného plánování, ale i odezvu uživatelských aplikací. Důležitou roli hraje prostředí, ve kterém je plánovač (scheduler) nasazen. Plánování procesů se chová rozdílně v desktopovém grafickém prostředí, které musí přijatelně rychle reagovat na akce uživatele a na serveru, kde interaktivní procesy nemají takovou prioritu. K tomuto rozhodování pomáhají vlastnosti procesů jako např. plánovací třída, do které proces spadá, dosavadní chování nebo jeho priorita. Operační systém umožňuje uživateli prioritu neboli váhu procesu nastavit. [22]

Aby se zabránilo situaci, kdy krátké úlohy musí dlouho čekat na dokončení delších úloh s vyšší prioritou, rozdělí se všechny úlohy na malé části a střídavě jsou spuštěny na procesoru tak, aby v daném časovém úseku každá úloha získala určitou část času procesoru. Tím je zajištěno, že úloha s krátkou dobou zpracování dokáže být dokončena a opustit systém v co nejkratším časovém horizontu. Cílem plánovače je tedy minimalizovat očekávaný součet časů potřebných k dokončení všech úloh. [22]

Následující část kapitoly se věnuje základní terminologii spojené s plánováním procesů a následný popis hlavní strategie plánování a analýza hlavních kritérií, které ovlivňují efektivitu plánovače.

1.1 Proces

Proces představuje základní jednotku, s níž operační systém pracuje a které přiděluje systémové zdroje. Lze si jej představit jako běžící program neboli konkrétní instanci spustitelného kódu. Každý proces má vlastní adresní paměťový prostor, vlastní identifikátor (PID) a soubor prostředků a vlastností, jimiž jsou souborové deskriptory nebo práva. Z pohledu plánovače jde o entitu, které přiděluje čas procesoru. Na druhou stranu z pohledu procesu je právě on ten jediný, kdo vlastní procesor. Provádění více úloh v systému a přepínání mezi nimi probíhá transparentně pro konkrétní proces. Tato abstrakční vrstva zajišťuje, že každý proces má řízený přístup k hardwaru a zároveň je izolován od ostatních procesů. [22] [23]

V systému Linux, se vytváření procesů provádí pomocí systémového volání `fork()` nebo `clone()`. Pomocí `clone()` je možné definovat sdílené prostředky s rodičovským procesem (která bude blíže specifikována v kapitole 1.2). Používá se ke zkopírování existujícího procesu a jeho následné změně pro účely nově vytvořené úlohy. Nově vzniklý proces se nazývá potomek (child) původního procesu, který je označen jako rodič (parent). Každý proces má přesně jednoho rodiče, s výjimkou procesu `init`, který je prvním procesem vytvořeným při zavedení jádra Linuxu. Nově vzniklý proces dostane vlastní PID z rozsahu 1 až 32762 a zdědí většinu prostředků rodičovského procesu (například proměnné prostředí nebo otevřené deskriptory souborů). Přesto však pracuje ve svém odděleném (virtuálním) paměťovém prostoru. Identifikátor PID se z důvodu malého rozsahu recykluje poměrně často a bude krátce po uvolnění použit pro proces jiný. [22]

1.1.1 Životní cyklus procesu

Za existence může proces nabývat různých stavů podle toho, co zrovna dělá a zda může být spuštěn. Linux rozlišuje několik základních stavů procesu, které si jádro udržuje proměnné `task_struct->state`. Proces může nabývat stavů:

- Spuštěný/spustitelný – `TASK_RUNNING`
- Pozastavený – `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE`
- Zastavený – `TASK_STOPPED`
- Zombie – `EXIT_ZOMBIE`, `EXIT_DEAD`

Spuštěný anebo spustitelný proces je buď právě vykonáván na CPU, nebo čeká ve frontě na přidělení CPU. Ve výpisu nástroje „ps“ se tento stav označuje příznakem „R“ (running/runnable). Stav `TASK_RUNNING` znamená, že proces nic neblokuje a je zařazen v runqueue plánovače. [23]

Pozastavený proces ve stavu `TASK_INTERRUPTIBLE` s příznakem „S“ („sleeping“) čeká na událost jako například vstup/výstup a dočasně neběží. Čekání procesu lze přerušit signálem. Tento stav nastane při volání blokující funkce jako například čtení z disku, kdy se proces uspí do doby, než je požadovaný úkol hotov. Po dokončení události se stav procesu změní zpět na běžící (`TASK_RUNNING`). [23]

Nepřerušitelně pozastavený proces čeká na dokončení určité operace, typicky I/O (vstup/výstup) úlohy, a nelze jej probudit signálem. Tento stav se využívá, v případě kdy probíhající operaci nelze bezpečně přerušit. Proces ve stavu `TASK_UNINTERRUPTIBLE`

s příznakem „D“, ignoruje signály a jakmile očekávaná operace skončí, jádro jej probudí. [23]

Proces označen jako TASK_STOPPED s příznakem „T“ („stopped“) není vykonáván. K zastavení může dojít pomocí signálu SIGSTOP tedy uživatelsky klávesovou zkratku Ctrl+Z nebo při ladění kdy je proces debuggerem zastaven pomocí dosažení požadované části kódu. V zastaveném stavu reaguje proces jen na některé signály jako SIGCONT, který jej může navrátit do stavu běžícího nebo SIGKILL, který jej ukončí. [23] [24]

Stavem zombie se označuje proces, který již ukončil svou činnost, ale stále má záznam v tabulce procesů. V takovém stavu proces zůstane, dokud rodič nepožádá jádro o jeho odstranění. Je označen příznakem „Z“ („zombie“). [23]

Proces může ukončit svůj běh několika způsoby. Dobrovolně sám pomocí systémového volání exit(), nebo je ukončen zvenčí signálem jako např. SIGKILL. Při ukončení procesu jádro uvolní téměř všechny jeho prostředky. Uvolní paměť procesu, uzavře soubory anebo jej i odebere z plánovače. Záznam o procesu (task_struct) není odstraněn okamžitě. Proces zůstává ve zmíněném zombie stavu a čeká, až jeho rodič převezme informaci o ukončení. Rodič by měl následně zavolat wait() k vyzvednutí statusu ukončeného potomka. Tím se dozví, s jakým kódem potomek skončil a jádro může dokončit úklid. Pokud rodič tuto povinnost nesplní, zombie proces zůstává v systému. Takový proces nezabírá téměř žádné prostředky, ale mnoho zombie procesů indikují chybu aplikace, kde rodič nezpracovává potomky. OS umožňuje, aby si opuštěné zombie procesy převzal init pokud rodič procesu skončí dříve než potomek. Proces init pravidelně volá wait() a uklízí tak zombie úlohy. [22]

1.2 Vlákna

V jiných operačních systémech se obvykle implementují vlákna jako odlehčený proces. Vlákna se vytvoří rychle v „odlehčeném“ stavu narozdíl od běžných „těžkých“ procesů, jejichž vytvoření vyžaduje více času. V Linuxu se vytvoření vlákna a procesu zásadně neliší. Jedná se o proces, který sdílí různé prostředky s jiným procesem. Hlavně se jedná o sdílení adresního prostoru rodičovského procesu s jeho vlákny. To je v Linuxu realizováno systémovým voláním clone() se specifickými příznaky (clone_flags), které určují sdílené zdroje. Nastavením příznaku „CLONE_VM“ dáváme funkci clone() pokyn, aby sdílel paměť rodičovského procesu s potomkem. Dalším příznakem je „CLONE_FS“ umožňující sdílení informací o souborovém systému nebo „CLONE_FILES“ otevřených souborů rodiče. [25]

Vlákná jsou rozlišena pomocí identifikačního čísla TID (thread identifier). Vlákno sdílí hodnotu TID s rodičovským procesem, tedy jeho PID. Dále rozlišujeme skupiny vláken pomocí TGID (thread group identifier). Místo explicitního konceptu vláken jako u jiných systémů používá Linux jednotný abstraktní pojem úloha (task), který reprezentuje plánovatelnou entitu. Všechna vlákna i běžné procesy jsou v jádru reprezentovány stejným datovým typem (task_struct). V kontextu plánovačů procesů není mezi procesem a vláknem rozdíl a chovají se identicky. [22]

1.2.1 POSIX vlákna

POSIX vlákna (pthreads) poskytuje standardizovaný způsob programování s vlákny napříč Unixovými systémy. V Linuxu je toto rozhraní implementováno knihovnou libpthread. Základní funkcí pro spuštění nového vlákna je pthread_create().

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void*),
                  void *arg);
```

Obrázek 1 Implementace funkce pthread_create

(zdroj: [26])

První argument „pthread_t“ je ukazatel na proměnnou, kam funkce uloží identifikátor nově vytvořeného vlákna. Argument pthread_attr_t umožňuje specifikovat atributy vlákna. Pokud je nastaven na NULL, použijí se výchozí atributy. Mezi typické atributy patří například „stack size“ určující velikost zásobníku vlákna, „sched policy“ nebo „priority“ nastavující plánovací politiku a prioritu vlákna. Všechny se nastavují pomocnými funkcemi, kterými jsou pthread_attr_setdetachstate(), pthread_attr_setschedparam(), pthread_attr_setschedpolicy(), pthread_attr_setstack() a pthread_attr_setstacksize() před předáním atributu do pthread_create(). Argument „start_routine“ je ukazatel na funkci, která představuje počáteční rutinu vlákna. Nové vlákno tedy po vytvoření začne vykonávat danou funkci start_routine. V případě chyby při vytvoření vlákna, kdy návratová hodnota pthread_create() není rovná 0, není žádné vlákno vytvořeno a obsah thread proměnné je nedefinovaný. [26]

Vytvoření vláken a přepínání kontextu je zpravidla méně nákladné než u procesů, protože při přepnutí vláken v rámci jednoho procesu není nutné měnit celý kontext. Na

základě sdílení prostředků je komunikace mezi vlákny, jako předávání dat, mnohem jednodušší než komunikace mezi oddělenými procesy. To s sebou nese ovšem riziko nekorektního přístupu více vláken, tedy přístup k jedné sdílené proměnné více vláken najednou, což může vést k nekonzistentním datům či tzv. race condition a vyvstává potřeba synchronizace. Synchronizace zajišťuje správnou koordinaci vláken tak, aby si nepřekážela při přístupu ke sdíleným zdrojům a datům. Nejčastěji se v synchronizaci používá Mutual Exclusion Lock (Mutex), zámek pro vzájemné vyloučení. V POSIX je reprezentován typem `pthread_mutex_t`. Mutex umožňuje, aby v daném okamžiku pouze jedno vlákno vstoupilo do kritické sekce kódu. Kritickou sekcí se z pravidla označuje přístup ke sdíleným datům. Ostatní vlákna, která se pokusí uzamknout již zamčený mutex, se uspí do doby, než se mutex uvolní. Tím se zaručí, že kritickou sekci vykonává vždy jen jedno vlákno. [2]

Vlákno se může ukončit několika způsoby. Nejčastěji vlákno jednoduše dokončí svou funkci, čímž volá `pthread_exit()`. Alternativně může vlákno zavolat `pthread_exit()` kdykoli, čímž předčasně ukončí svou činnost a předá návratovou hodnotu. Vlákno také může být asynchronně zrušeno funkcí `pthread_cancel()` z jiného vlákna. Pokud dojde k ukončení celého procesu, jsou ukončena všechna vlákna daného procesu současně. Po ukončení vlákna nastává fáze úklidu zdrojů. Vlákno uvolní alokovanou paměť svého zásobníku a dalších datových struktur v jádře. [26]

1.3 Plánovač procesů

V každém okamžiku existují alespoň některé procesy, které jsou současně ve stavu, kdy je možné je spustit. Na každém jádru (vlákně) procesoru však může být současně spuštěn pouze jeden proces. Plánovač procesů je v Linuxu a obecně v každém operačním systému klíčovou součástí, jejímž úkolem je rozhodovat, který proces či vlákno bude spuštěno v daném okamžiku na procesoru. Pravidelně se přerušuje aktuálně běžící proces a je plánovačem vybírán jiný spustitelný, který jej nahradí. Doba, po kterou jeden proces běží, než je přerušen, se liší v závislosti na prioritě procesu a strategii plánování. [7]

Kromě samotného výběru procesů k vykonání je úkolem plánovače také zajistit, aby žádný proces nebyl dlouhodobě přehlížen. Bez dostatečné kontroly by totiž mohlo dojít k situaci, kdy určité procesy s vysokou prioritou zcela vytěsní ostatní méně prioritní procesy. Proto většina moderních plánovačů implementuje mechanismy prevence vyhladovění (starvation) jako například postupným zvyšováním priority čekajících procesů nebo

pravidelnou rotací běžících úloh. Výsledkem je spravedlivější rozdělení času CPU mezi všechny procesy a zajištění plynulého chodu celého systému. [7]

Díky rychlému střídání vykonávání procesů dosáhneme zdánlivého multitaskingu. Uživatel může současně například poslouchat hudbu, pracovat v textovém editoru, kompilovat program a stahovat soubory z internetu, přičemž z jeho pohledu vše probíhá současně a bez větších prodlev. To vše je možné právě na základě efektivního fungování plánovače procesů. [7]

1.4 Přepínání kontextu

Přepínání kontextu (context switching) je činnost, kdy se přeruší běh právě aktivního procesu nebo vlákna a přenese se řízení na jiný proces. Uloží se kontext přerušeného procesu do paměti, aby se k němu později mohl operační systém opět vrátit a pokračovat v místě přerušení. Jakmile je stav přerušovaného procesu uložen, načte se odpovídající stav pro nově vybíraný proces. Například načte uložené registry a převede správu paměti na adresní prostor nového procesu. Tento přesun z jednoho procesu na druhý s sebou nese režii (overhead). Čas a zdroje potřebné na uložení a obnovení stavu by mohly být využity ke skutečnému běhu programů. Z tohoto důvodu se moderní plánovače snaží minimalizovat zbytečně časté přepínání kontextu. [22]

1.5 Preemptivní plánování

Moderní linuxové systémy používají tzv. preemptivní plánování. To znamená, že systém může kdykoliv přerušit právě běžící proces či vlákno a dát prostor jinému, například s vyšší prioritou. Hlavní rozdíl oproti nepreemptivnímu plánování spočívá v tom, že u nepreemptivní metody se aktuálně běžící proces vzdá řízení například při dokončení úlohy, při požadavku na vstupně/výstupní operaci nebo vypršením časového kvanta, zatímco s preemptivním plánováním se může provést přepnutí kontextu kdykoli. To umožňuje průběžně reagovat na vzniklé situace, jako je požadavek procesu s vyšší prioritou či probuzení procesu, který dlouho čekal na I/O.

Preemptivní plánování je proto zvláště užitečné v interaktivních systémech, kde dochází k akcím uživatele, který vyžaduje rychlé odezvy od systému. V momentě, kdy uživatel vykoná akci např. stisknutí klávesy, může scheduler přerušit méně důležitý proces a okamžitě dát k dispozici procesor té aplikaci, která jej aktuálně vyžaduje. Zaručuje se tím lepší interaktivita a menší latence systému. [22]

1.6 Priorita a časové kvantum

Priorita procesu vyjadřuje důležitost daného procesu (vlákna) v očích plánovače, jenž podle této hodnoty rozhoduje jak často a kolik procesorového času (časového kvanta) procesu přidělí. Dopad priority závisí na použitém plánovacím algoritmu. [21]

Typicky platí, že plánovač každému běžícímu procesu nebo vláknu přidělí časové kvantum, tedy určitý časový úsek, dosahující až několik milisekund, během kterého má proces na CPU vyhrazený prostor k provedení svého kódu. Po vypršení kvanta dojde k přerušení (pokud proces mezitím neskončil nebo dobrovolně neusnul) a předá se CPU jinému procesu.

Bez speciální obsluhy je délka časového kvanta zásadním parametrem plánování, protože ovlivňuje odezvu systému a jeho celkovou propustnost. Pokud je kvantum příliš krátké, dochází k častému přepínání mezi procesy, což zvyšuje režii systému a může vést ke snížení výkonu. Naopak příliš dlouhé kvantum může způsobit, že interaktivní aplikace mají delší odezvu, což negativně ovlivňuje uživatelský komfort. Operační systémy proto dynamicky přizpůsobují délku kvanta na základě aktuální zátěže a charakteru běžících úloh. [21]

1.7 Třídy plánování

Linuxové plánování procesů disponuje několika plánovacími třídami (scheduling classes), které určují, jakým způsobem jsou procesy spravovány a ovlivňují například jejich prioritu. Mezi plánovací třídy patří:

- Stop
- EDF – „Early Deadline First“
- RT – „Real-Time“
- „Fair“ – Třída implementovaná plánovačem CFS a EEVDF
- Idle

Třídy RT a „Fair“ dále obsahují politiky zajišťující přesnější zařazení procesu a žádoucí chování plánování. [2] [4]

1.7.1 Stop

Plánovací třída Stop je v jádře operačního systému speciální interní mechanismus, který sám o sobě neslouží k plánování uživatelských úloh. Místo toho funguje jako nástroj, jenž umožňuje na určitém CPU přerušit běh všech ostatních procesů a vykonat zadanou funkci s nejvyšší prioritou. Používá se výhradně v systémech se symetrickým multiprocessingem (SMP). Vytváří jedno jádrové vlákno označované jako migration/N kde N označuje číslo procesoru. Tato třída je pak využívána jádrem při migraci procesů mezi procesory. [5]

1.7.2 EDF – Early Deadline First

EDF implementuje plánovací politiku SCHED_DEADLINE. Je navržena pro aplikace s přísnými časovými požadavky, jako jsou multimediální systémy, průmyslové řízení nebo jiné real-time úlohy. Zavádí mechanismus, který umožňuje úlohám nastavit své časové požadavky pomocí tří parametrů.

- runtime – maximální doba běhu úlohy v každém období,
- deadline – časový limit, do kterého musí být úloha dokončena,
- period – interval, ve kterém se úloha opakuje.

Každá úloha plánovaná pomocí SCHED_DEADLINE by měla obdržet "runtime" mikrosekund času na procesoru v každém "period" mikrosekundovém intervalu, přičemž tento čas musí být využit do "deadline" mikrosekund od začátku období. Úloha s nejbližším termínem dokončení má nejvyšší prioritu. Pokud úloha překročí svůj přidělený "runtime", je pozastavena až do začátku dalšího období, kdy se její "runtime" obnoví a "deadline" se posune o hodnotu "period". [2]

1.7.3 RT – Real-Time

Třída RT je určena pro „real-time“ aplikace, které vyžadují garantovaný a deterministický čas procesoru. Používá algoritmus plánování založený na prioritách, který zajišťuje, že procesy s vysokou prioritou vždy dostanou potřebný čas procesoru. Je optimalizován pro aplikace, jako jsou průmyslové řídicí systémy a embedded zařízení, kde je kritická nízká latence a předvídatelný výkon. Jednou z výzev je synchronizace úloh, které sdílejí prostředky. Bez správného řízení může dojít k jevu známému jako priority inversion (převrácení priority), kdy nízko prioritní úloha blokuje prostředek potřebný pro úlohu s vyšší prioritou, což může vést k porušení časových omezení. Řešení těchto problémů často

vyžaduje implementaci pokročilých synchronizačních protokolů. Plánování v reálném čase je velmi náročné a představuje především snahu co nejvíce se tomuto ideálu přiblížit. Zahrnuje politiky SCHED_FIFO, SCHED_RR. [2] [3] [4]

Proces s politikou SCHED_FIFO se vždy spustí před úlohami v jakékoliv jiné plánovací třídě či politice. SCHED_FIFO používá pevnou prioritu mezi 1 (nejnižší) a 99 (nejvyšší). Proces bude vykonáván, dokud sám nepřejde do režimu spánku nebo nezačne čekat na vstup anebo nebude proces s vyšší prioritou připraven ke spuštění. Pokud některá z těchto událostí nenastane, proces na daném procesoru poběží neomezeně dlouho a úlohy s nižší prioritou nedostanou šanci k běhu. [22]

Politika SCHED_RR je velmi podobná SCHED_FIFO, avšak s implementovaným časovým kvantem a „round-robin“ algoritmem. Po vypršení časového úseku, který byl procesu přiřazen, se proces přesune na konec fronty a naplánuje se spuštění dalšího bez ohledu na to, zda byl proces dokončen, nebo ne. Tím je zajištěno, že všechny úlohy dostanou podíl času procesoru a nenastane vyhladovění procesů. [2] [4]

1.7.4 Třída „Fair“

Třída „Fair“ a její politiky byly implementovány plánovačem CFS. Obecně se označuje jako „třída CFS“ avšak po zavedení plánovače EEVDF, který převzal implementaci třídy, není toto označení přesné. Má vždy menší prioritu než zmíněná třída RT. Tento typ plánování pracuje s hodnotou vruntime (virtuální čas běhu), případně s virtual deadline (virtuální termín dokončení), přičemž plánovač přiděluje procesorový čas tak, aby byl rozdělen spravedlivě mezi všechny běžící procesy. Zahrnuje politiky:

- SCHED_NORMAL – standardní plánování pro běžné úlohy
- SCHED_BATCH – dávkové plánování optimalizované pro procesy s nižší interaktivitou
- SCHED_IDLE – plánování pro úlohy s nejnižší prioritou, které mohou být spuštěny, pokud je procesor nečinný.

SCHED_NORMAL neboli SCHED_OTHER je výchozí plánovací politika používaná pro běžné uživatelské úlohy, které nevyžadují specifické real-time vlastnosti ani zvláštní optimalizaci pro dávkové zpracování. Součástí této politiky je také koncept hodnoty nice (ohleduplnost), který umožňuje uživateli manuálně ovlivnit prioritu procesu. Hodnota nice se pohybuje v rozmezí od -19 (nejvyšší priorita) do 20 (nejnižší priorita), přičemž

výchozí hodnota je 0. Hodnotu nice lze měnit pomocí systémových volání jako `setpriority` nebo `sched_setattr`, čímž lze určit, které procesy budou preferovány při přidělování CPU času. Pokud proces obdrží hodnotu nice 10, bude tomuto procesu méně často přidělován procesorový čas než proces s hodnotou nice 0. [2] [4]

Politika `SCHED_BATCH` je určena pro úlohy, které nejsou interaktivní a mohou být vykonávány na pozadí bez požadavku na rychlou odezvu. Tento režim je vhodný zejména pro výpočetně náročné úlohy, které běží delší dobu a nevyžadují okamžitou interakci s uživatelem. Oproti `SCHED_NORMAL` proces v tomto režimu nemá tak výrazné preemptivní chování jako při standardním plánování. To umožňuje vyšší propustnost systému a optimalizaci využití času procesoru pro úlohy, jako je kompilace kódu nebo zpracování velkých objemů dat jako například výpočty v oblasti vědeckých simulací nebo šifrování. Stejně jako u politiky `SCHED_NORMAL` lze i v rámci `SCHED_BATCH` upravit prioritu procesu pomocí hodnoty nice. [2] [4]

Politika `SCHED_IDLE` je určena pro procesy s nejnižší možnou prioritou, které mohou běžet pouze tehdy, když je procesor zcela nečinný a nemá žádné jiné úlohy ke zpracování. Tento režim je ideální pro procesy, které nemají žádné přísné požadavky na výkon a mohou být vykonávány zcela na pozadí bez jakéhokoliv zásahu do běžícího systému. Plánování v režimu `SCHED_IDLE` je často využíváno pro údržbové úlohy systému, jako je indexace souborů. Na rozdíl od `SCHED_NORMAL` a `SCHED_BATCH` zde hodnota nice neovlivňuje prioritu procesu. [2] [4]

Proces `SCHED_IDLE` je od verze jádra 5.4 vždy považován za nečinný. Pokud bude spustitelný jiný proces, který nespadá do `SCHED_IDLE`, plánovač jej okamžitě přeruší. [5]

1.7.5 Idle

Poslední třídou je Idle, která má nižší prioritou než zmíněná `SCHED_IDLE`. Tato třída je nejnižší v hierarchii priorit a podobně jako třída Stop neřídí žádné uživatelské úlohy. Udržuje pouze jeden proces na každé CPU, pojmenovaný `swapper/N` kde N označuje číslo procesoru. Tyto procesy nejsou v uživatelském prostoru viditelné. Jejich hlavním úkolem je šetřit energii tím, že v okamžicích nečinnosti systém uvedou do úsporného režimu. [5] [22]

2 ARCHITEKTURA CPU A JEJÍ VLIV NA PLÁNOVÁNÍ PROCESŮ

Moderní vícejádrové procesory přinesly do plánování úloh řadu nových výzev. Plánovač musí zohledňovat aspekty jako SMT, NUMA, cache hierarchii nebo i bezpečnostní opatření spojené se sdílením cache.

2.1 Simultánní multithreading (SMT)

SMT umožňuje dvěma či více logickým procesorům sdílet jedno fyzické jádro současně. To zvyšuje využití jádra, ale vlákna na stejném jádře si konkurují o sdílené zdroje jako je cache paměť. Důsledkem je, že dvě vlákna běžící na různých fyzických jádrech běží rychleji než dvě vlákna sdílející jedno SMT jádro. Plánovač proto musí rozhodovat, kdy úlohy “sloučit” na jedno jádro a kdy je rozložit na různá jádra. [9]

Linux řeší SMT prostřednictvím plánovacích domén a skupin odpovídajících topologií CPU. CFS preferuje, rozdělit procesy tak, aby nejprve využil všechna fyzická jádra před zaplněním druhých vláken na jádrech. To znamená, že na systému se 4 jádry (8 SMT vláken) bude nejprve vytíženo 4×1 vlákno na každém jádře; teprve když běží více než 4 procesy, začne plánovač využívat druhé vlákno na jádru. Tím se dosáhne blízké lineární škálovatelnosti výkonu až do počtu fyzických jader. Pokud by byly naplánovány dvě těžké úlohy na jedno SMT jádro a jiné jádro by zůstalo nečinné, došlo by ke zbytečnému zpomalení. Moderní plánovač tomu aktivně předchází pomocí vyvažování zátěže (load balancing). [1] [2] [19]

2.2 Nerovnoměrný přístup do paměti (NUMA, CPU affinity)

NUMA (Non-Uniform Memory Access) je úzce spjatý s fyzickým rozložením paměti. Ta je na čipu rozdělena mezi jednotlivé uzly, typicky odpovídající procesorům nebo skupinám jader, přičemž každý uzel má vlastní lokální paměť, ke které má rychlý přístup. Přístup k paměti jiného uzlu je však pomalejší a náročnější na propustnost, což vytváří nerovnoměrnou latenci a propustnost při přístupu k paměti. [10]

Moderní procesory mají víceúrovňové vyrovnávací paměti (cache). Například malou L1 (vyrovnávací paměť první úrovně) a L2 (vyrovnávací paměť druhé úrovně) cache pro každé jádro a sdílenou L3 (vyrovnávací paměť třetí úrovně). Plánovač musí brát ohled na tuto hierarchii, protože přepnutí úlohy na jiné jádro může znamenat ztrátu dat v cache a nutnost ji znovu plnit například na jiném socketu což snižuje výkon. Z toho důvodu Linux uplatňuje afinitu vůči CPU a snaží se ponechat proces běžet opakovaně na stejném jádře.

Výjimkou je situace, kdy původní jádro nestíhá a jiné je volné. Pak dojde k migraci za účelem vyvážení zátěže, ale i to se děje s ohledem na topologii cache. [10]

Plánovací domény zrcadlí cache hierarchii a fyzickou strukturu procesoru. Jádra sdílející cache, například L3, jsou ve společné doméně, v níž je migrace relativně levná. Mezi vzdálenějšími doménami (jiný socket) je migrace omezená. Moderní plánovače udržují pro každý běžící proces informaci k identifikaci jádra, na kterém byl proces spuštěn naposledy (CPU affinity hint) a při plánování se nejprve pokouší použít toto jádro. [1] [2] [10]

2.3 Heterogenní architektura CPU (P-core / E-core)

Moderní CPU (Intel Alder Lake a novější) kombinují výkonná jádra (P-core) a efektivní jádra (E-core) v jednom systému. Tato heterogenní architektura znamená, že ne všechna jádra mají stejný výpočetní výkon ani stejnou energetickou náročnost. Plánovač proto musí rozpoznat rozdíl v “kapacitě” jader a úlohy na ně přiřazovat inteligentně. Například náročný jednovláknový proces by měl běžet na výkonném P-core, zatímco méně náročné služby mohou běžet na E-core, které má nižší spotřebu.

Linux na toto reagoval rozšířením plánovače o Capacity Aware Scheduling a Energy Aware Scheduling (EAS). Plánovač si udržuje metriky, kolik výkonu může dané jádro poskytnout (CPU capacity). Moderní plánovače jsou díky tomu schopné zohlednit heterogenní CPU a přiřazovat úlohy s ohledem na kapacitu. Podpora pro big.LITTLE architektury, která kombinuje výkonná a úsporná procesorová jádra v jednom systému na čipu mobilních zařízení byla vyvinuta v Linuxu pro ARM a nyní se aplikuje i na x86 hybridní čipy.

2.4 Dynamické řízení taktu (Turbo Boost)

Moderní procesory dynamicky mění svůj takt a napětí podle vytížení a teploty, aby maximalizovaly výkon a zároveň splňovaly limity spotřeby. Například Intel technologie „Turbo Boost“ a pro AMD hardware „Precision Boost“, dokáže dočasně zvýšit frekvenci jednoho nebo více jader procesoru, pokud je dostatečná tepelná rezerva. Tyto mechanismy znamenají, že počet aktivních jader a jejich zátěž ovlivňuje, jak rychle které jádro běží. Jedno vytížené jádro může dosáhnout vyšší frekvence, zatímco při zátěži všech jader se frekvence sníží kvůli limitům. Plánovač má tedy vliv na výkon nepřímo tím, jak rozhodne rozprostřít úlohy na jádra procesoru. Může volit mezi strategií rozložení procesů na více jader (spread),

tím snížit možnost turbo frekvencí, nebo naopak využít menší množství aktivních jader (pack) s vyšším taktům a ponechat ostatní v úsporném režimu. [12]

Linuxový scheduler v moderním kernelu spolupracuje s podsystémem CPUfreq pro řízení frekvence. Konkrétně governor „schedutil“ využívá údaje od plánovače o vytížení CPU a počítá optimální frekvenci na základě zátěže a třídy úlohy. To znamená, že plánovače jako CFS a EEVDF průběžně měří, kolik CPU času úloha potřebuje, a governor podle této informace zvyšuje nebo snižuje takt. Například rychle zvedne frekvenci, když přibude zátěž, a naopak ji sníží při nečinnosti. [12]

2.5 Bezpečnostní hrozby a mitigace

Důležitým faktorem, jsou bezpečnostní slabiny moderních CPU a jejich následný dopad na plánovač. Zranitelnosti jako Spectre a Meltdown ukázaly, že sdílení zdrojů procesy může vést k únikům informací mezi nimi. Plánovač se ocitl v roli „strážce“, který může pomoci omezit nebezpečí tím, jak izoluje úlohy na CPU.

Jedním z opatření je omezení SMT. Někteří uživatelé na citlivých systémech SMT vypínají, aby se zabránilo sdílení jádra a tím i přístupu k paměti dvěma různými procesy. Jemnější přístup v Linuxu představuje „core scheduling“. Jde o rozšíření plánovače, které umožňuje seskupit procesy do důvěryhodných skupin. Každá skupina dostane identifikátor (tag) a jen procesy se stejným tagem mohou běžet současně na vláknech jednoho fyzického jádra. Např. všechny procesy jednoho uživatele nebo jednoho virtuálního stroje (VM) mohou mít stejný tag a běžet spolu, ale nikdy nesdílejí jádro s procesy jiné skupiny. To účinně brání úniku dat z cache, i když SMT zůstává zapnuto. [11]

3 PLÁNOVAČE PROCESŮ

Historický vývoj plánovačů procesů sahá od prvních batch systémů, kdy bylo cílem co nejefektivněji využít tehdy drahý hardware, až po time-sharing operační systémy, které přinesly preemptivní plánování a umožnily současný běh mnoha uživatelských programů. V raných systémech se používala relativně jednoduchá prioritní schémata a algoritmy pro přidělování CPU, přičemž se začaly zavádět sofistikovanější mechanismy jako dynamické úpravy priorit na základě dosavadního chování procesu. [2]

Od 7. března 1995 ve verzi 1.2 jádro Linuxu využívalo pro řízení běžících procesů round-robin algoritmus. Díky jednoduchosti bylo vkládání a odebírání procesů velmi rychlé a efektivní. Pozdější verze 2.2 zavedená 26. ledna 1999, přenesla koncept plánovacích tříd, což umožnilo implementovat různé politiky pro procesy vyžadující speciální obsluhu. Současně se objevila podpora symetrického multiprocessingu (SMP), čímž se Linux otevřel širším možnostem nasazení na vícejádrových či víceprocesorových systémech. [2]

S nástupem vícejádrových procesorů se do popředí dostala problematika vyrovnávání zátěže, zatímco potřeba rychlé odezvy interaktivních aplikací podpořila intenzivnější využívání preempece. V prostředí Linuxu se tento vývoj odrazil zejména v přechodu od jednodušších plánovačů ke scheduleru $O(1)$. Do jádra Linuxu následoval plánovač CFS a až nyní nově uvedený EEVDF. [2]

3.1 $O(n)$

Verze jádra Linuxu 2.4 vydána 4. ledna 2001 zavedla plánovač $O(n)$, který byl z několika pohledů lepší než plánovač s jednoduchým round-robin algoritmem, který byl doposud používán. Děлил čas na epochy (epoch), během kterých měl každý proces ve frontě možnost k běhu. Proces dostal předem určený časový úsek, kterého mohl využít. Takzvaný time slice. Pokud proces svůj podíl procesorového času nevyčerpal celý, například začal čekat na vstup uživatele, polovina nevyužitého času se mu přičetla do další epochy. [6]

Plánovač využíval pro výběr úlohy ke spuštění algoritmus se složitostí $O(n)$. Prozkoumával celý seznam spustitelných procesů. Tato metoda sice nebyla složitá, ale měla omezující výkon a nestavěla na novějších možnostech hardwaru, jako jsou vícejádrové procesory, ani nebyla příliš výhodná pro real-time systémy. Škálovatelnost byla velký problém. Čas, za který plánovač $O(n)$ naplánuje proces, se lineárně škáloval s počtem procesů. V případě velkého počtu spustitelných procesů strávil plánováním následující

úlohy velké množství času, což mělo za následek, že méně času bylo přiděleno na samotný běh procesů. [8]

3.2 O(1)

Jádrem 2.6 v roce 2003 byl nasazen plánovač O(1), jehož autorem je Ingo Molnár, dokázal plánovat úlohy v konstantním čase neohledně na množství procesů, které jsou právě v systému připraveny ke spuštění. Do linuxového jádra se dostala celá řada inovací, které zlepšily efektivitu a přinesly snahu o vyšší interaktivitu systému. V první řadě se zavedly priority s rozsahem 0 až 139, kde nižší číslo znamenalo vyšší prioritu. Rozmezí 0 až 99 bylo vyhrazeno pro úlohy v reálném čase (real-time), zatímco hodnoty 100 až 139 připadly běžným, tzv. normálním procesům. Priorita úlohy současně určila velikost časového kvanta (timeslice), a mohly tak úlohy s vyšší prioritou běžet na procesoru déle. Další novinkou bylo časné přerušování (preemptce). Jakmile se proces ocitl ve stavu „TASK_RUNNING“, jádro okamžitě porovnálo jeho prioritu s prioritou aktuálně běžícího procesu. Pokud byla vyšší, došlo ihned k vyvolání plánovače, který nově připravené úloze předal CPU. Pro real-time procesy nově platila statická priorita, jež se v průběhu jejich činnosti neměnila. Oproti tomu běžné (normální) procesy získaly dynamickou prioritu, vypočítávanou podle míry jejich interaktivity. Jakmile skončilo přidělené časové kvantum, systém jejich prioritu přepočítal a na základě dosavadního chování určil, nakolik byla daná úloha interaktivní. Interaktivní procesy tím pádem získávaly od plánovače vyšší prioritu, aby systém zajistil rychlou odezvu uživatelsky orientovaných aplikací a minimalizoval prodlevy způsobené čekáním na procesor. [21]

O(1) pracoval s aktivním a expirovaným polem, které obsahovaly 140 lineárních seznamů korespondující s každou prioritou. Aktivní pole obsahovalo procesy připravené ke spuštění s nevyužitým časovým kvantem, kterým plánovač postupně přiděloval čas procesoru. Expirované pole se plnilo procesy, které již přidělené časové kvantum využily. Jakmile bylo aktivní pole prázdné, tedy všechny procesy byly v expirovaném poli, ukazatele na tyto pole se vyměnily. Expirované pole se tedy stalo aktivním a naopak. Složitost O(1) byla dosažena průchodem bitmapy o 140 bitech. Každý jeden bit nastavený na hodnotu „1“ znamenal, že seznam s odpovídající prioritou obsahuje spustitelný proces. [21]

Problémy O(1) byly zřejmé ve zpracování interaktivních procesů nebo procesů vázaných na vstup/výstup operace (IO-bound). Plánovač obsahoval velké množství těžko

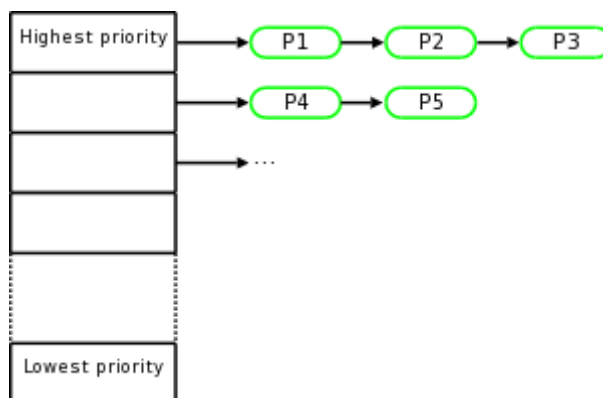
udržitelného kódu, kterým se snažil dynamicky upravit prioritu takových úloh podle průměrné doby čekání, avšak tyto výpočty byly příliš náročné a často nepřesné. [8] [21]

3.3 Alternativní plánovače

Významnějším alternativním plánovačem se věnoval australský vývojář Con Kolivas, který se soustředil na problematiku latence a interaktivity v desktopových systémech. I přes to, že jeho návrhy byly označovány za výrazné zlepšení oproti tehdejším algoritmům, nebyl žádný přijat do hlavní větve jádra Linuxu.

3.3.1 RSDL

Rotating Staircase Deadline Scheduler (RSDL) byl představen v roce 2007. Con Kolivas kritizoval stávající plánovač $O(1)$ za problémy zpracování interaktivních úloh. $O(1)$ způsoboval při vysoké zátěži příliš dlouhé čekání procesů na CPU, což vedlo ke zpoždění v odezvě systému. Hlavní myšlenkou RSDL je rotující schodišťový model, kde jsou seznamy procesů seřazeny podle priority do úrovně neboli schodů. Každému procesu jsou přiděleny dvě časové kvóty. První je časový úsek (timeslice), po který je procesu dovoleno být spuštěn. Po vypršení tohoto časového úseku, je vybrán následující proces stejné priority pomocí round-robin algoritmu. Druhá kvóta vyjadřuje maximální dobu spuštění procesu na dané úrovni. Po využití tohoto časového limitu, je proces přerušen a přesunut na nižší úroveň. Úlohy s vyšší prioritou dostaly více příležitostí k běhu pádem přes větší počet úrovní. Další pojistkou před uvíznutím se stala globální lhůta pro celou úroveň. Ta znamenala, že nehlédě na vyčerpání kvót jednotlivých procesů, bude celý seznam dané priority přesunut o úroveň níže. Po běhu všech dostupných úloh jsou procesy navráceny na úroveň korespondující jejich původní prioritě. Tento model zaručoval, že žádná úloha nemohla vyhladovět a zároveň se zajistilo deterministické chování i při vysokém zatížení CPU. RSDL byl testován komunitou a považován za významné zlepšení oproti $O(1)$ plánovači, ale nebyl přijat do hlavní větve jádra. Namísto něj byl vybrán plánovač CFS. Kolivas po odmítnutí RSDL oznámil odchod z vývoje linuxového jádra. [15]



Obrázek 2 Schodová priorita plánovače RSDL

(zdroj:[15])

3.3.2 BFS

Po neúspěchu RSDL při integraci do jádra se Con Kolivas vrátil v roce 2009 s dalším návrhem plánovače Brain Fuck Scheduler (BFS) jako reakci na rostoucí složitost a serverově orientované chování aktuálně používaného plánovače CFS. Cílem BFS bylo minimalizovat latenci, zjednodušit kód a zvýšit interaktivitu systémů. Oproti tradičním linuxovým plánovačům využívá jednoduchý mechanismus a struktury. BFS obsahuje jednu společnou frontu pro všechny procesy. Každému procesu je přiřazen virtuální deadline, který určuje pořadí, ve kterém budou procesy naplánovány k běhu. Proces s nejbližším deadline je vybrán jako další k provedení. Jedním z problémů BFS byla škálovatelnost, tedy použití na systémech s více CPU. Takový problém provází nejen BFS a autor Con Kolivas na tuto skutečnost poukazuje ve svých poznámkách: [32]

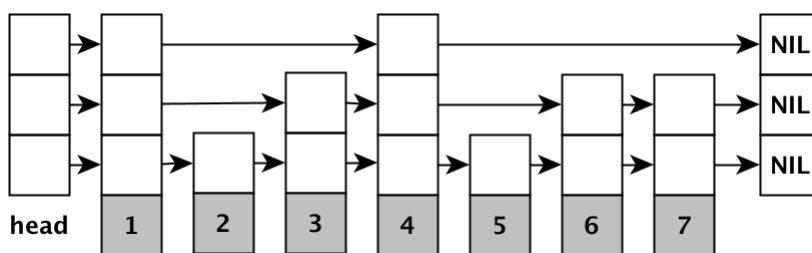
„I don't own the sort of hardware that is likely to suffer from using it, so I can't find the upper limit. Based on first principles about the overhead of locking, and the way lookups occur, I'd guess that a machine with 16 CPUs or more would start to have exponentially less performance (thanks Ingo for confirming this). Note that the number of logical CPUs is what affects BFS' scalability, not the physical ones. By that I mean that a hyperthreaded CPU that is a quad core hyperthreaded is 8 EIGHT logical CPUs. So it is NOT the same as a quad core without hyperthreading.“

(zdroj: [32])

3.3.3 MuQSS

Posledním plánovačem, který Con Kolivas představil se stal MuQSS (Multiple Queue Skiplist Scheduler). Ten se prezentuje jako jednoduchý a efektivní plánovač, který se znovu zaměřuje na nízkou latenci a vysokou interaktivitu. Narozdíl od BFS, který používal jednu

globální frontu, MuQSS implementuje samostatné fronty pro každý procesor. Každá fronta je implementována jako skiplist. Ten je tvořen vrstveným linkovaným seznamem. První úroveň obsahuje všechny prvky. Každá vyšší úroveň obsahuje pouze některé prvky. Umožňuje efektivní operace vkládání, mazání a vyhledávání s časovou složitostí $O(\log n)$. To zajišťuje rychlé plánování i při velkém počtu procesů. MuQSS zachovává algoritmus plánování z BFS, který upřednostňuje úlohy podle jejich virtuálního deadline. Tento přístup zajišťuje předvídatelné chování. [30]



Obrázek 3 Skiplist

(zdroj: [31])

3.4 CFS

Po přelomu tisíciletí sílila potřeba lepší správy procesorového času s důrazem na spravedlnost a interaktivitu. Vývojář Con Kolivas experimentoval s alternativními plánovači zaměřenými na desktopovou odezvu. Jeho návrh Rotating Staircase Deadline Scheduler prokázal, že férové plánování je možné a zlepšuje odezvu. Tento přístup inspiroval Ingo Molnára, autora původního $O(1)$ plánovače, k vytvoření nového plánovače nazvaného Completely Fair Scheduler (CFS). V dubnu 2007 jej poprvé zveřejnil na Linux Kernel Mailing List (LKML) jako kompletní přepsání plánovacího subsystému a verzi jádra 2.6.23 následně nahradil dosavadní $O(1)$ scheduler. Důležitou myšlenku, na níž je CFS z velké části postaven, je snaha v reálném prostředí napodobit ideální procesor, který by mohl všechny úlohy provádět paralelně a každé z nich přidělovat přesně stejnou rychlost ($1/X$, kde X je počet spouštěných procesů). Procesor, který spouští dvě úlohy, věnuje každé poloviční výkon, přičemž vypadají, jako by běžely současně, avšak s poloviční rychlostí. Ve skutečnosti taková dokonalá paralelizace na jednom reálném CPU neexistuje, ale CFS se tento princip snaží co nejlépe simulovat. Plánovač CFS přinesl zásadní změny v tom, jak Linux přistupuje k řízení procesů, a stal se standardním plánovacím algoritmem. Začlenění CFS do oficiálního linuxového jádra vyvolalo kritiku ze strany vývojářské komunity

například kvůli inspiraci plánovačem RSDS nebo i kvůli preferenci serverových prostředí na úkor interaktivních desktopových systémů. [2]

CFS přes 15 let obsluhoval procesy na principu spravedlivého dělení času. Postupem času však vyvstaly požadavky, které CFS řešil jen pomocí různých heuristik a záplat. Zejména potřeba lépe zacházet s odezvou pro interaktivní a krátce běžící úlohy. CFS neposkytoval přímý způsob, jak úloha může vyjádřit požadavek na latenci. Hodnoty „nice“ upravovaly pouze podíl času CPU, nikoli požadavek rychlosti reakce. Proto vznikly „latency nice“ záplaty, které měly umožnit nastavení latence citlivým procesům a tím přednostně předběhnout ostatní. Tyto patche fungovaly, ale přidávaly do CFS další komplexitu. [29]

3.4.1 Princip CFS

V CFS je klíčovou hodnota vruntime. Ta vyjadřuje, kolik času již proces dostal k dispozici na procesoru. Uchovává se pro každý proces v proměnné `p->se.vruntime` v nanosekundách. Na ideálním hardwaru, kde by všechny úlohy mohly běžet souběžně, by se vruntime u všech procesů navyšoval také společně. Ve skutečnosti však CFS pouze aproximuje tento ideál a neustále volí k běhu ten proces, který má momentálně nejmenší hodnotu vruntime, a tím se snaží udržet rozdělení procesorového času co nejrovnoměrnější. [2]

3.4.2 Implementace CFS

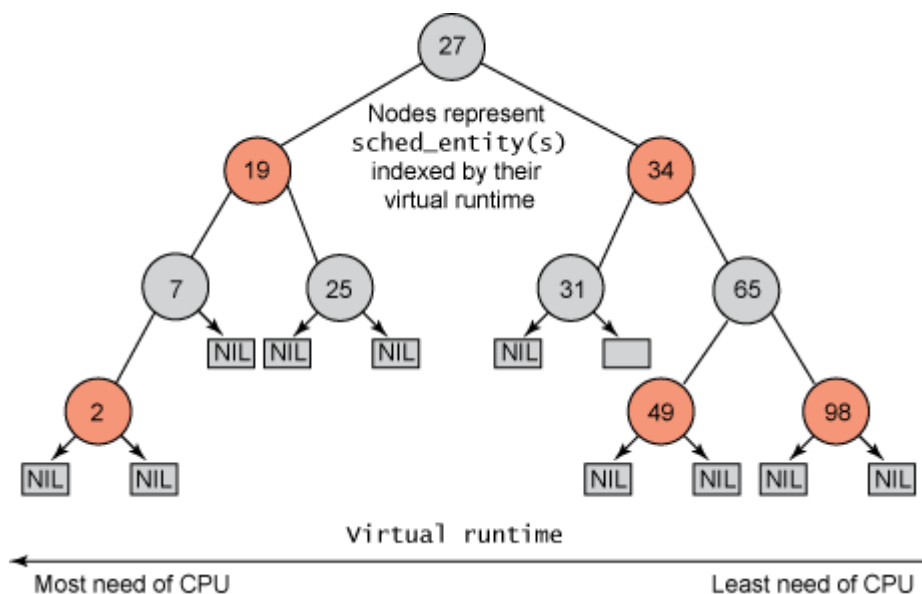
Výrazným rysem CFS je nahrazení starých datových struktur fronty časově seřazeným stromem, takzvaný RB-strom (red-black tree), ve kterém si hodnoty vruntime pro jednotlivé procesy udržuje. Jedná se o samovyvažovací binární strom, což znamená, že žádná cesta ve stromě není nikdy více než dvakrát delší než každá jiná. Procesy jsou v RB-stromě seřazeny od nejnižší hodnoty vruntime až po nejvyšší zleva doprava. Plánovač následně vybírá právě ten proces uložený v nejvíce levé pozici. [2]

Pro porovnání procesů při vkládání do RB-stromu se používá funkce `entity_before` přítomná v „`kernel/sched/fair.c`“, která rozhoduje na základě vruntime.

```
static inline bool entity_before(const struct sched_entity *a,
                                const struct sched_entity *b)
{
    return (s64)(a->vruntime - b->vruntime) < 0;
}
```

Obrázek 4 Implementace funkce entity_before CFS

(zdroj: [20])



Obrázek 5 Red-Black Tree

(zdroj: [8])

V rámci této struktury udržuje také hodnotu `rq->cfs.min_vruntime`, která se neustále navyšuje a představuje nejmenší vruntime z přítomných procesů. Hodnotou této proměnné se inicializuje vruntime každého nově spuštěného procesu. Slouží k tomu, aby se právě spuštěné úlohy umísťovaly co nejvíce doleva v RB-stromu, a zároveň zohledňovaly doposud vykonanou práci. Nebudou tedy výrazně zvýhodněny oproti již běžícím procesům. [2]

Při každém dobrovolném uvolnění procesoru, či při přerušení způsobeném hodinami plánovače, aktualizuje vruntime právě běžící úlohy. Jakmile tato hodnota vruntime přesáhne určitou hranici, za níž má jiný proces menší hodnotu vruntime, dojde k preempci a střídání úloh.

Namísto dřívějšího přímého využívání priorit při určení pořadí používá CFS tyto hodnoty jako modifikátor navyšování vruntime. Základním vstupem je hodnota nice, kterou lze nastavit v rozsahu -19 až 20. Nižší hodnota nice prodlužuje dobu, po kterou se proces

může zdržet na CPU. Naopak vyšší (kladné) nice se projeví rychlejším nárůstem vruntime a tím i kratšími časovými úseky na procesoru. [8] Každá změna hodnoty nice o jeden stupeň mění procesorový čas procesu přibližně o 10 %. Tento efekt je kumulativní, což znamená, že pokud jeden proces zvýší hodnotu nice o 1 stupeň a druhý jej sníží o 1 stupeň, jejich vzájemný rozdíl v přiděleném čase CPU se zvětší přibližně o 25 %. [2] Výpočet vruntime se provádí podle vzorce:

$$\frac{T}{\frac{1024}{X}} = vruntime$$

kde T vyjadřuje jednotky času strávené procesem na CPU a X jako váha procesu. [13]

Výchozí váha procesu je definována pro nice rovno 0 jako hodnota 1024. Ostatní hodnoty se od ní odvozují pomocí násobení nebo dělení faktorem 1,25. Překladová tabulka je k nalezení v souboru „kernel/sched/core.c“.

```
const int sched_prio_to_weight[40] = {
    /* -20 */    88761,    71755,    56483,    46273,    36291,
    /* -15 */    29154,    23254,    18705,    14949,    11916,
    /* -10 */    9548,    7620,    6100,    4904,    3906,
    /* -5 */    3121,    2501,    1991,    1586,    1277,
    /* 0 */    1024,    820,    655,    526,    423,
    /* 5 */    335,    272,    215,    172,    137,
    /* 10 */    110,    87,    70,    56,    45,
    /* 15 */    36,    29,    23,    18,    15,
};
```

Obrázek 6 Překladová tabulka váhy procesu

(zdroj: [13])

Například procesu s hodnotou nice 0, který byl spuštěn 1000 jednotek času, bude přičteno číslo 1000 k dosavadnímu vruntime. Procesu s nice rovno -1 bude přičteno 802 dle výpočtů:

$$\frac{1000}{\frac{1024}{1024}} = 1000$$

$$\frac{1000}{\frac{1024}{820}} = 802$$

To znamená, že proces s hodnotou nice -1 získal z pohledu následujícího plánování o 198 jednotek času méně než proces s hodnotou nice 0 a dostane více času na procesoru. [13] [14]

3.4.3 Výpočetní složitost a efektivita využití výkonu

Časovou složitost plánování CFS je $O(\log N)$, přičemž N je počet procesů čekajících ke spuštění. Samotný výběr úlohy s nejmenším vruntime může proběhnout v konstantním čase, tedy $O(1)$, ale znovuzaražení procesu zpět do vyváženého RB-stromu zabere právě $O(\log N)$ operací. [8]

Ve vícejádrovém prostředí vytváří samostatnou frontu pro každé jádro a pravidelně vyrovnává zátěž mezi vytíženějšími a méně vytíženými jádry. Přitom se musí zohlednit topologie systému, protože přesun procesu mezi jádry může znamenat i přesun dat z vyrovnávací paměti, což může v některých případech degradovat výkon více, než kdyby proces zůstal na vytíženějším jádru. CFS využívá plánovací domény. Podle rozdílu v zátěži pak rozhoduje, zda úlohy migrovat z jedné skupiny do druhé. Plánovač pravidelně kontroluje a upravuje rovnoměrné rozložení výkonu od nejnižší a nejlevnější domény až po nejvyšší a nejdražší. Tímto způsobem se snaží optimálně využít výpočetní výkon a zároveň minimalizovat negativní efekty spojené s migrací procesů napříč různými jádry či procesory. [8]

3.5 EEVDF

Jako reakce na nedostatky CFS přišel plánovač EEVDF (Earliest eligible virtual deadline first), jehož koncept byl popsán již v akademické literatuře v 90. letech. [16] Myšlenku EEVDF pro Linux znovu oživil vývojář Peter Zijlstra v roce 2023. Cílem bylo dosáhnout spravedlivého plánování včetně zohlednění latence úloh, a tím odstranit potřebu dodatečných záplat které přinesl CFS. Scheduler EEVDF byl začleněn do hlavní vývojové větve jádra Linuxu ve verzi 6.6 v říjnu 2023 a stal se novým výchozím plánovačem pro „Fair“ třídu procesů a tím pro politiky SCHED_NORMAL, SCHED_BATCH a SCHED_IDLE. EEVDF stojí na myšlence „virtuálních termínů“ (virtual deadline) na rozdíl od CFS, který se rozhodoval pouze na základě vruntimes. Tím se snaží docílit lepší obsluhy interaktivních aplikací. Dle poznámek k vydání kernelu 6.6 slibuje EEVDF lepší výkon a spravedlnost plánování při menší závislosti na křehkých heuristikách než CFS.

Vývojáři však upozorňují, že takto zásadní změna může v okrajových případech způsobit i regresní chování, na jehož doladění se nadále pracuje. [33]

3.5.1 Princip EEVDF

EEVDF je, podobně jako CFS, plánovač založený na virtuálním čase, který se snaží spravedlivě rozdělit CPU čas mezi všechny běžící úlohy podle jejich váhy. Při n současně aktivních procesech by každý měl ideálně získat $\frac{1}{n}$ dostupného času CPU. Na rozdíl od CFS však EEVDF zavádí několik nových konceptů k řízení pořadí a preempece. [17] [33]

Každý proces má svůj vlastní vruntime podobně jako v CFS. Roste úměrně skutečnému času běhu procesu a je normalizován jeho váhou. Na základě toho EEVDF průběžně vyčísľuje hodnotu lag (zpoždění procesu). Lag je definován jako rozdíl mezi tím, kolik času měl proces ideálně dostat podle spravedlivého podílu, a tím, kolik reálně dostal. Pokud proces zaostává za ideálem, jeho lag je kladný a je mu dlužen čas na CPU. Naopak pokud již běžel déle, než odpovídá spravedlivému podílu, lag vyjde záporný. [17] [33]

Eligible (způsobilý) se v EEVDF nazývá takový proces, jehož lag je větší nebo rovno 0. Tedy žádný nebo kladný dluh. Proces s negativním lag není dočasně považován za způsobilý k běhu, protože již vyčerpal více než spravedlivý čas a musí nejprve počkat, až se jeho záporný lag zvýší minimálně na nulu, což nastane plynutím času, kdy běží jiné úlohy. EEVDF se snaží udržovat lag všech aktivních procesů co nejblíže nule. Důležitou vlastností je, že součet lag hodnot všech aktivních úloh je v každém okamžiku nula. Pokud jeden proces dostal čas navíc, jiní musí být pozadu o stejnou sumu. Tato vlastnost zajišťuje konzistenci spravedlnosti. [14]

Eligible time (čas způsobilosti) úlohy je právě ten časový okamžik, kdy se její lag zvýší na nulu neboli kdy úloha dorovná náskok a opět se stane způsobilou k naplánování. [14]

Každé úloze EEVDF periodicky přiděľuje určité časové kvantum běhu (slice). EEVDF počítá slice dle váhy procesu, a navíc zohledňuje jeho požadavky na latenci. Pro každou úlohu, která je přípustná k běhu, pak plánovač stanoví její virtuální deadline (virtual deadline) neboli virtuální termín dokončení běhu. Virtuální deadline se vypočítá jako součet eligible time a délky slice daného procesu. Intuitivně to znamená nejpozdější čas, kdy by tato úloha měla (ideálně) obdržet své přiděľené kvantum běhu. Například má-li proces slice 10 ms a stane se způsobilým za 20 ms, pak jeho virtuální deadline bude za 30 ms od

aktuálního času. U procesu, který je již způsobilý nyní (eligible time je rovno 0), je virtuální deadline rovno právě délce jeho slice. [17] [33]

Plánovač při každém výběru následujícího procesu porovná virtuální deadline všech způsobilých úloh a vybere k běhu tu, která má nejbližší (nejmenší) virtuální deadline. Tím je realizováno pravidlo Earliest Eligible Virtual Deadline First tedy nejbližší přípustný termín první. Žádný proces s nárokem na CPU (kladný lag) se nemusí obávat vyhladovění. Vždy dostane přednost před těmi procesy, které už svůj čas na CPU využily. [17] [33]

EEVDF umožňuje zohlednit různou citlivost úloh na latenci pomocí parametru latency nice (priorita latence). Jde o hodnotu v rozsahu -20 až 19, která určuje, jak krátké slice má proces dostávat. Úloha s nižší hodnotou latency nice, požaduje nižší latenci, chce rychle reagovat, a proto jí plánovač přidělí menší slice. Oproti tomu úloha s vyšším latency nice, je tolerantní k latenci a dostane delší slice. [18] Důležité je, že tato úprava nemění celkový podíl CPU dané úlohy, pouze rozdělí běh procesu na více menších úseků. Například dva procesy se stejnou váhou obdrží za delší období stejný procesorový čas, ale ten s přísnějšími požadavky na odezvu ho bude dostávat po menších dávkách. Protože virtuální deadline se počítá jako součet eligible time a slice, platí, že procesy s menším slice budou mít bližší deadliny a tudíž přednost v běhu. Latency nice tak elegantně zajišťuje, že interaktivní úlohy dostanou CPU rychle po probuzení, aniž by jim bylo nutné uměle zvyšovat podíl CPU (nice hodnoty CFS) a dostanou pouze menší “dílký” času, které je upřednostní. Tím EEVDF řeší problém latence přirozeně v rámci algoritmu, namísto dodatečných heuristik jako v CFS. [17] [33]

Pro příklad výběru procesu je možné představit si tři procesy („A“, „B“ a „C“) soutěžící o čas na CPU.

	A	B	C
Lag	0 ms	0 ms	0 ms

Obrázek 7 Příklad výpočtu lag, část 1.

Všechny úlohy začínají s lag rovné 0 ms, jsou tedy všechny způsobilé k běhu. „A“ bude vybráno a využije 30 ms.

	A	B	C
Lag	-20 ms	10 ms	10 ms

Obrázek 8 Příklad výpočtu lag, část 2.

Za dobu 30 ms měla každá úloha nárok na 10 ms. „A“ využilo 30 ms a tím získalo -20 ms lag. Ostatním procesům byl přidělen kladný lag 10 ms. Dále bude spuštěno například „C“.

	A	B	C
Lag	-10 ms	20 ms	-10 ms

Obrázek 9 Příklad výpočtu lag, část 3.

V tento moment je jediný eligible proces „B“ a bude vybrán jako následující. [33]

3.5.2 Implementace EEVDF

Implementace EEVDF v jádře Linuxu navazuje na infrastrukturu CFS, ale mění klíčové části algoritmu. Každý logický procesor má vlastní frontu úloh (runqueue). Každý proces má v interní struktuře uložené váhové parametry a nově také parametr `latency_nice` vyjadřující požadavek na latenci. EEVDF také využívá RB-strom, ale na rozdíl od CFS porovnává virtuální deadline. Operace vložení nové úlohy či návratu úlohy do fronty a odebrání mají složitost $O(\log n)$, kde n je počet procesů, stejně jako u CFS. Upravenou funkci použitou k porovnání při vkládání do RB-stromu můžeme vidět v „kernel/sched/fair.c“:

```
static inline bool entity_before(const struct sched_entity *a,
                                const struct sched_entity *b)
{
    /*
     * Tiebreak on vruntime seems unnecessary since it can
     * hardly happen.
     */
    return (s64)(a->deadline - b->deadline) < 0;
}
```

Obrázek 10 Implementace `entity_before` EEVDF

(zdroj: [20])

Jako příklad zjednodušení můžeme uvést funkci „pick_next_entity“, která rozhodovala o způsobu vybrání následujícího procesu. V porovnání s CFS nemusí funkce procházet několik kontrol okrajových případů. [20]

```
static struct sched_entity *  
pick_next_entity(struct rq *rq, struct cfs_rq *cfs_rq)  
{  
    struct sched_entity *se;  
    if (sched_feat(PICK_BUDDY) &&  
        cfs_rq->next && entity_eligible(cfs_rq, cfs_rq->next)) {  
        /* ->next will never be delayed */  
        WARN_ON_ONCE(cfs_rq->next->sched_delayed);  
        return cfs_rq->next;  
    }  
    se = pick_eevdf(cfs_rq);  
    return se;  
}
```

Obrázek 11 Implementace pick_next_entity EEVDF

(zdroj: [20])

V kódu je možné vidět několikrát reference na CFS. Jak bylo zmíněno výše, EEVDF je založeno na infrastruktuře předchozího plánovače, a tedy pojmenování proměnných a funkcí nadále zůstává.

II PRAKTICKÁ ČÁST

4 PROVEDENÍ TESTŮ PLÁNOVAČŮ

Praktická část byla zaměřena na provedení testů, které porovnávají vlastnosti plánovače CFS a jeho nástupce EEVDF. Testy vychází z výsledků prezentovaných v emailové komunikaci vývojářů linuxového jádra, konkrétně v diskusi na Linux Kernel Mailing List (LKML). Zde byly představeny výsledky testů, mimo jiné i výsledné latence plánovačů měřené nástrojem `cyclicttest` při současném zatížení procesoru pomocí nástroje `hackbench`. Cílem praktické části práce je zopakování těchto měření s doplněním o dodatečné zjištění. [27]

Za objekt testování byl zvolen rozdíl v latenci interaktivních procesů, rozdíl v latenci interaktivních procesů s různými hodnotami „nice“ a konzistence latence interaktivních procesů. Jedná se o hodnoty, které se dle prezentovaných vlastností plánovačů dají předpovědět, opakovatelně změřit, porovnat a objektivně poukázat na změnu chování.

4.1 Příprava testovacího prostředí

Byla připravena dvě totožná prostředí v podobě dvou virtuálních strojů (VM) na virtualizační platformě Proxmox. Procesorem tohoto hypervisoru byl AMD EPYC 9754 (128 jader s dvěma vlákny na jádro, tedy 256 vláken). Použitím tohoto procesoru bylo možné také otestovat, zda prostředí s velkým množstvím jader (i vláken) má na výsledek vliv a přiblížit se zmíněným testům prezentovaným na LKML, kde byly použity více nespécifikované ARM procesory dohromady s 60 jádry a 8 vlákny na jádro. [27] Provádění testů ve virtualizovaném prostředí výrazně zjednodušilo a zrychlilo jednotlivé iterace testů.

Na obou VM byl nainstalován OS Fedora 41. Ten byl zvolen z důvodu existence obou testovaných verzí jádra v již stabilním vydání OS. Na prvním virtuálním stroji bylo zkompilováno a nainstalováno jádro verze 6.5.13 s výchozím plánovačem CFS. Na druhé VM bylo obdobně připraveno jádro verze 6.6. Před samotnou kompilací bylo nutné do každého systému doinstalovat potřebné nástroje a knihovny. To bylo provedeno pomocí následujícího příkazu:

```
dnf install ncurses-devel flex bison gcc numactl-devel \
openssl-devel-engine openssl dwarves elfutils-libelf-devel
```

Po instalaci potřebného softwaru byly ze stránek `kernel.org` staženy zdrojové kódy příslušných verzí jádra. Konfigurace obou jader byla převzata z výchozí konfigurace distribučního kernelu přítomného společně s OS Fedora 41. Konfigurační šablona byla zkopírována do adresáře se staženým zdrojovým kódem jádra.

```
cp /boot/config-$(uname -r) .config
```

Tato konfigurace mohla být v případě potřeby upravena pomocí nástroje „make menuconfig“, nicméně v rámci této práce byly ponechány výchozí hodnoty odpovídající systému Fedora 41. Následně proběhla kompilace jádra, modulů, jejich instalace a instalace samotného jádra.

```
make -j$(nproc)
make modules_install
make install
```

Po restartu systému byla příslušná verze jádra zvolena v nabídce GRUB (GRand Unified Bootloader). Takto připravená prostředí na dvou oddělených virtuálních strojích byla následně využita pro srovnávací testy plánovačů.

Při testování bylo vycházeno z předpokladu, že virtualizované prostředí ovlivní absolutní hodnoty naměřených latencí, avšak tento vliv bude stejný pro oba plánovače tedy jejich vzájemné srovnání zůstane zachováno. Tento předpoklad byl následně také ověřen. Na obou strojích bylo zajištěno, aby v průběhu provádění testů nebyla výpočetní kapacita CPU ovlivněna jinou zátěží a veškerý výkon fyzického procesoru tak mohl být věnován pouze testovacím úlohám.

4.2 Použité nástroje

K provedení testů byly použity nástroje z balíku rt-tests, který představuje standardní sadu testovacích nástrojů pro měření latence plánovače a ověřování vlastností systému v podmínkách reálného času případně i jiných plánovacích třídách. Tento balík je spravován vývojáři linuxového jádra a je volně dostupný v repozitáři projektu Linux kernel. Obsahuje několik specializovaných nástrojů určených primárně pro testování reálné odezvy procesů, mezi které patří například cyclicttest, hackbench, pi_stress či rt-migrate-test. V rámci této práce byly využity zejména nástroje cyclicttest pro měření latence plánování procesů a hackbench k simulaci intenzivní zátěže. Tato kombinace umožňuje spolehlivě a opakovatelně testovat rozdíly v chování plánovačů procesů. [28]

4.2.1 Hackbench

Pro generování zátěže byl použit nástroj hackbench. Hackbench je jednoduchý benchmark a zároveň zátěžový test. Jeho hlavní úlohou je vytvořit určitý počet dvojic procesů, které si mezi sebou posílají zprávy, a měřit, jak dlouho trvá výměna těchto zpráv. V praxi hackbench

spustí velké množství procesů, které neustále odesílají a přijímají malé bloky dat, čímž generují zatížení procesoru.

V linuxové komunitě se jedná o běžně používaný mikrobechmark. Hackbench sice nereprezentuje reálnou uživatelskou aplikaci, ale vytváří extrémní případ intenzivní komunikace vláken, který je vhodný pro porovnání plánování. V provedených testech hackbench kontinuálně generoval zátěž na pozadí, zatímco byla měřena latence jiného procesu pomocí nástroje cyclicttest. Tímto způsobem lze odhalit, jak si plánovač poradí s interaktivní úlohou měření latence běžící souběžně s masivní zátěží.

Použitím přepínače `-g` je nastaven počet skupin spuštěných procesů. Společně s příkazem „`nproc`“ se mohou dynamicky měnit počty skupin procesů rovný počtu dostupných jader, tedy aby hackbench dokázal spolehlivě vytižit procesor. Přepínač `-l` určuje počet cyklů a pro účely této práce stanovuje časový interval, po který bude hackbench spuštěn. [28]

Prezentovaná data jsou výsledkem průměrů získaných z minimálně pěti identických testů. Tento postup minimalizoval možnost chyby měření.

4.2.2 Cyclicttest

Pro měření latence byl použit nástroj cyclicttest. Cyclicttest měří čas probuzení procesu, tj. čas, který stráví ve frontě aktivních procesů čekajících na CPU. Prakticky funguje tak, že spustí jedno či více vláken, která usínají a mají se probudit v předem stanovených intervalech. Po probuzení vlákno zaznamená, o kolik později oproti ideálnímu času (okamžik požadavku o probuzení) se skutečně probudilo. Cyclicttest průběžně sbírá statistiky těchto odchylek a vypisuje minimální, průměrnou a maximální naměřenou latenci. Ve výchozím stavu zobrazuje hodnoty v jednotkách μs .

Cyclicttest je využíván zejména v testování real-time systémů k ověřování, zda splňuje požadavky latence. V kontextu této práce byl však použit v prostředí běžného uživatelského procesu ve třídě `SCHED_NORMAL`. Výhodou nástroje cyclicttest je, že poskytuje konzistentní výsledky v různé podobě a umožňuje spouštění s různými parametry, které zásadně ovlivňují způsob měření a umožňují testovat různé aspekty chování plánovače. Použitím přepínače `-t` se spustí měření na každém dostupném jádře s částečně odlišnými podmínkami jako např. odlišný čas plánovaného probuzení. Tento skok v intervalu probuzení definuje přepínač `-d`. Použitím „`--interval`“ definujeme počáteční interval probuzení prvního měřícího vlákna.

```
T: 0 ( 3081) P: 0 I:1000 C: 99437 Min: 22 Act: 58 Avg: 334 Max: 21385
T: 1 ( 3082) P: 0 I:4000 C: 29082 Min: 18 Act: 58 Avg: 507 Max: 24459
T: 2 ( 3083) P: 0 I:7000 C: 16957 Min: 40 Act: 138 Avg: 503 Max: 20651
T: 3 ( 3084) P: 0 I:10000 C: 11964 Min: 18 Act:3613 Avg: 573 Max: 27819
```

Obrázek 12 Příklad výstupu „cyclicttest -t -d 3000“

(zdroj: vlastní)

Z výstupu výše je možné vyčíst informace jako: počet jader systému (4), jednotlivé intervaly probuzení pod položkou „I“, počet cyklů probuzení, kterými vlákno již proběhlo „C“, minimální naměřená latence „Min“, latence naměřená posledním cyklem „Act“, průměrná naměřená latence „Avr“ a maximální naměřená latence „Max“. [28]

4.3 Postup testování a měřící metodika

Při testování byly kombinovány výše zmíněné nástroje tak, aby byl nasimulován scénář intenzivní zátěže systému a současně měřena latence interaktivního procesu. Cílem bylo zjistit, jak se liší chování plánovače CFS a EEVDF v extrémních podmínkách. Konkrétněji jaké latence vykazuje úloha čekající na CPU (cyclicttest) při současném běhu velkého množství konkurenčních procesů vytěžujících CPU (hackbench). Každý scénář byl testován variací následujícího bash scriptu „hackbench_cyclicttest.sh“:

```
#!/bin/bash
nice=$1
nice -n0 ./hackbench -g$(nproc) -l100000000000 &
PID=$!
sleep 1

nice -n$nice ./cyclicttest -D 5m
kill $PID
```

Obrázek 13 Testovací script hackbench_cyclicttest.sh

(zdroj: vlastní)

Script nejprve spustil program hackbench na pozadí pomocí „&“. Tomuto procesu byla nastavena hodnota nice 0, aby představoval standardní zátěž, která není upřednostněna či upozaděna oproti ostatním uživatelským procesům. Hackbench byl spuštěn s parametrem -l 100000000000 (10 miliard cyklů). Ten zajistil, že poběží téměř neomezeně dlouho, dokud nebude test latence dokončen. Následně byl uložen identifikátor PID hackbench procesu do

proměnné. Ta se při celkovém ukončení testu využila s nástrojem kill pro ukončení vytvořené zátěže.

Po jedné vteřině od spuštění hackbench, kdy již stabilně zatěžoval systém, bylo spuštěno samotné měření latence. Tomuto procesu byla nastavena hodnota nice převzatá z prvního parametru scriptu zvoleného při jeho spuštění. Například „hackbench_cyclictest.sh 16“ spustí cyclictest s hodnotou nice = 16. Dále sbíral data po stanovenou dobu (zde 5 minut). Důležité je, že cyclictest běžel ve stejné plánovací třídě jako hackbench tedy SCHED_NORMAL, takže oba podléhaly stejným podmínkám. Pro obecné měření byl script spuštěn s parametrem 0, tedy hackbench i cyclictest byly spuštěny s hodnotou nice = 0.

Během testování některých scénářů byl postupně zvyšován počet virtuálních procesorů (vCPU) alokovaných testovací VM v rozmezí od 4 až po 255. Bylo zajištěno, aby technologie SMT hypervisoru nebyla virtuálním strojem využívána pokud to nebude nezbytné (testovací VM bude využívat méně než 128 vCPU). Pozorovatelná změna v naměřených hodnotách byla očekávána od 128 jader výše, protože procesor AMD EPYC 9754 disponuje právě 128 fyzickými jádry a 256 vláken. Hackbench automaticky přizpůsoboval zátěž počtu dostupných CPU. Díky použití -g\$(nproc) využil všechna dostupná jádra, tedy každý test s vyšším počtem CPU znamenal více současně běžících úloh například pro 16 CPU hackbench spustil 640 procesů, pro 255 CPU 10200 procesů.

4.3.1 Latence a počet jader

Prvním ze scénářů je obecné měření latence interaktivních procesů s různým počtem dostupných jader. Testy probíhaly za použití bash skriptu popsaneho výše, který byl upraven tak, že nástroj cyclictest byl spouštěn následujícím způsobem:

```
cyclictest -D 5m -t -a -d X
```

Zde parametr -t a -a aktivoval jedno testovací vlákno na každé jádro systému a parametr -d X definoval změnu intervalu probouzení testovacích vláken. Za „X“ byla použita taková hodnota, aby první vlákno bylo probouzeno každou 1 ms a poslední každých 10 ms. Pro systém vlastníci 4 CPU byl interval nastaven na hodnotu 3000 udávající čas v μ s.

```

T: 0 ( 2278) P: 0 I:1000 C: 92840 Min: 10 Act: 78 Avg: 453 Max: 23619
T: 1 ( 2279) P: 0 I:2290 C: 45450 Min: 10 Act: 60 Avg: 645 Max: 74950
T: 2 ( 2280) P: 0 I:3580 C: 31141 Min: 11 Act: 63 Avg: 726 Max: 28350
T: 3 ( 2281) P: 0 I:4870 C: 23673 Min: 10 Act: 58 Avg: 752 Max: 74728
T: 4 ( 2282) P: 0 I:6160 C: 19049 Min: 12 Act: 63 Avg: 732 Max: 19857
T: 5 ( 2283) P: 0 I:7450 C: 15864 Min: 18 Act:6376 Avg: 789 Max: 72099
T: 6 ( 2284) P: 0 I:8740 C: 13578 Min: 12 Act: 59 Avg: 810 Max: 70793
T: 7 ( 2285) P: 0 I:10030 C: 11908 Min: 12 Act: 63 Avg: 768 Max: 32224

```

Obrázek 14 Výstup cyclicttest pro kernel 6.5.13 disponující 8 CPU

(zdroj: vlastní)

```

T: 0 ( 5254) P: 0 I:1000 C: 92671 Min: 12 Act: 82 Avg: 525 Max: 44748
T: 1 ( 5255) P: 0 I:2290 C: 48001 Min: 9 Act: 134 Avg: 523 Max: 31848
T: 2 ( 5256) P: 0 I:3580 C: 32580 Min: 9 Act: 182 Avg: 537 Max: 45479
T: 3 ( 5257) P: 0 I:4870 C: 24360 Min: 11 Act: 60 Avg: 553 Max: 45131
T: 4 ( 5258) P: 0 I:6160 C: 19307 Min: 13 Act: 60 Avg: 555 Max: 38232
T: 5 ( 5259) P: 0 I:7450 C: 16032 Min: 11 Act:2655 Avg: 555 Max: 38620
T: 6 ( 5260) P: 0 I:8740 C: 13670 Min: 9 Act: 112 Avg: 554 Max: 37763
T: 7 ( 5261) P: 0 I:10030 C: 11945 Min: 10 Act: 61 Avg: 536 Max: 26282

```

Obrázek 15 Výstup cyclicttest pro kernel 6.6 disponující 8 CPU

(zdroj: vlastní)

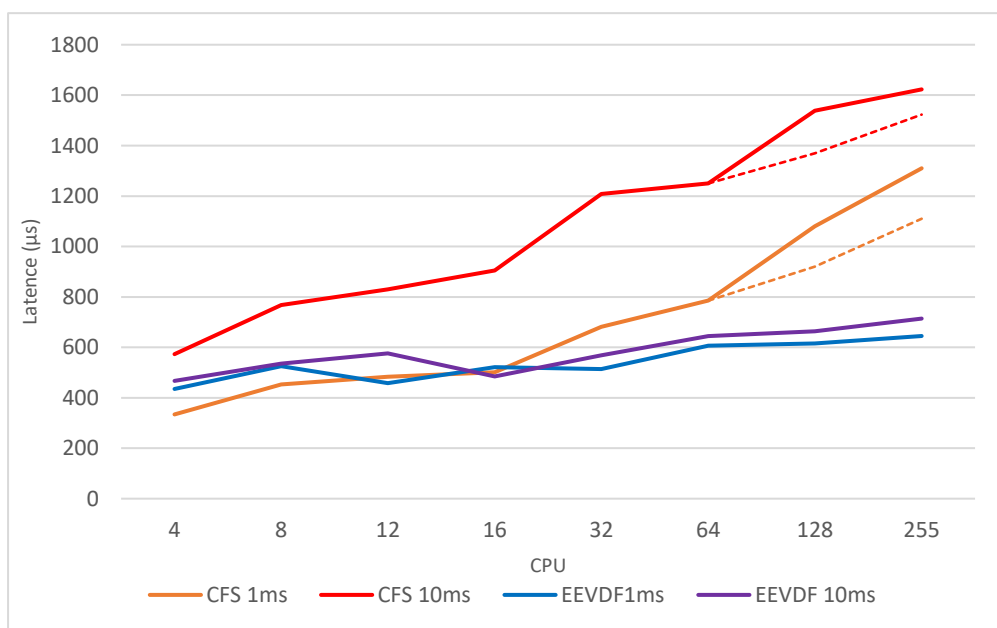
Z výstupu je patrné, že v případě vlákna s nejkratším intervalem 1 ms došlo u plánovače EEVDF oproti CFS k regresi, konkrétně o cca 16 %. Naopak pro vlákno s intervalem 10 ms bylo u plánovače EEVDF zaznamenáno významné snížení latence, a to o cca 30 % oproti CFS. Na základě těchto počátečních výsledků byl proveden rozsáhlejší experiment, při kterém byly měřeny latence obou plánovačů s různým počtem vCPU v rozmezí od 4 až do 255. Průměrné výsledky z několika měření byly zapsány do tabulky obsahující informaci o počtu jader, latenci pro vlákno s intervalem probuzení 1 ms a vlákno s intervalem probuzení 10 ms pro CFS a EEVDF v μ s:

CPU	CFS 1ms	CFS 10ms	EEVDF 1ms	EEVDF 10ms
4	334	573	435	467
8	453	768	525	536
12	483	830	458	576
16	501	905	522	485
32	682	1208	514	569
64	785	1250	606	645
128	1080	1538	616	664
255	1310	1623	645	714

Tabulka 1 Naměřené průměrné hodnoty latence v závislosti na počet CPU

(zdroj: vlastní)

Následně byla data zanesena do grafu. Znázorňuje absolutní hodnoty naměřených latencí (μs) při různém počtu CPU.



Obrázek 16 Naměřené průměrné latence (μs) v závislosti na počtu CPU

(zdroj: vlastní)

Z grafu je patrné, že plánovač EEVDF obecně vykazuje konzistentně nižší latence oproti plánovači CFS, zejména při intervalu 10 ms. Naopak při velmi krátkých intervalech probuzení (1 ms) jsou rozdíly mezi plánovači při nízkém počtu jader (do 16) minimální. Pohybují se v mezích chyby měření, případně vychází mírně lépe ve prospěch plánovače CFS. Rozdíl se postupně zvětšuje s rostoucím počtem jader a při nejvyšších počtech (128 a více) je patrné výrazné zhoršení latencí u plánovače CFS i v situaci krátkých intervalů. Zde

je patřičné podotknout, že nastává saturace fyzického procesoru hypervisoru a některé jeho procesy mohou mít vyšší prioritu oproti testovací VM. Také dochází k využití SMT, které má na výsledek ještě větší dopad. To může mít za následek ovlivnění měření a reálný růst latence by byl očekáván více lineární. Ten je v grafu značen přerušovanou čarou, avšak jedná se o spekulace nepodložené naměřenými hodnotami.

Následně byl vypočten poměr latencí obou plánovačů pro 1ms a 10ms.

$$\frac{EEVDF}{CFS}$$

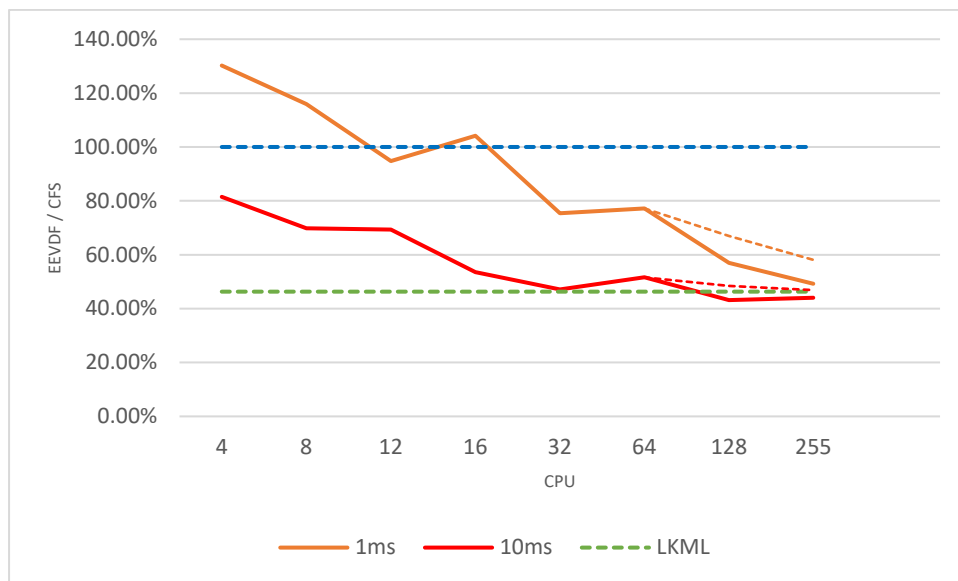
A v podobě procent zapsán do tabulky:

CPU	1ms	10ms
4	130,24%	81,50%
8	115,89%	69,79%
12	94,82%	69,40%
16	104,19%	53,59%
32	75,37%	47,10%
64	77,20%	51,60%
128	57,04%	43,17%
255	49,24%	43,99%

Tabulka 2 Procentuální poměr latencí EEVDF ku CFS v závislosti na počet CPU

(zdroj: vlastní)

Tato data byla zanesena do grafu se zvyšujícím počtem CPU:

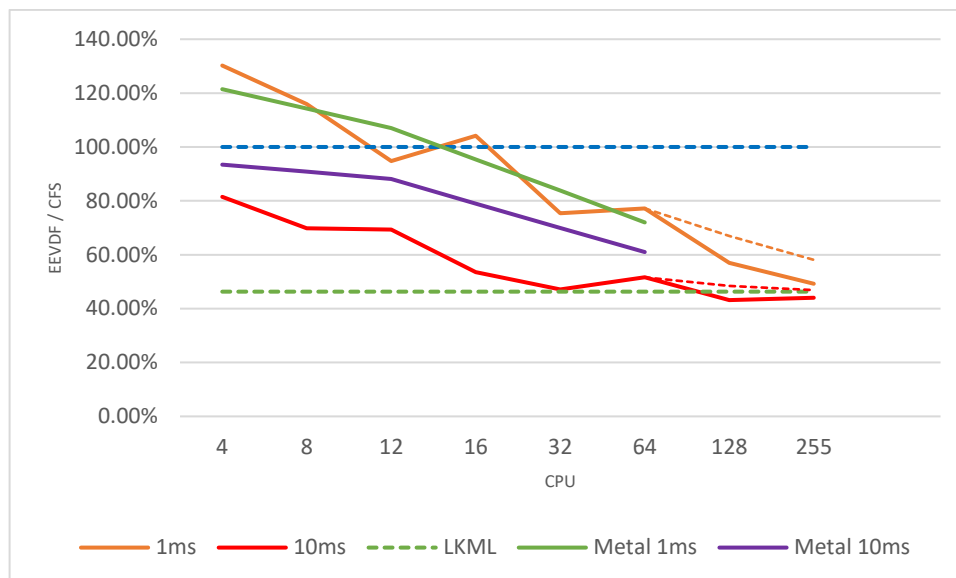


Obrázek 17 Procentuální poměr latencí EEVDF ku CFS v závislosti na počet CPU

(zdroj: vlastní)

V této reprezentaci je hodnota nižší než 100 % příznivá pro EEVDF, naopak hodnota vyšší než 100 % indikuje lepší výsledky plánovače CFS. Modrá přerušovaná čára odpovídá této hranici, kde latence obou plánovačů jsou totožné. Zelenou přerušovanou čarou je znázorněna referenční hodnota latencí publikovanou vývojáři na webu LKML, která byla naměřena na systému disponujícím 480 logickými jádry. Z uvedeného procentuálního srovnání je patrné, že plánovač EEVDF poskytuje zřetelně lepší výsledky latence oproti CFS při delších intervalech probouzení jako je 10 ms. Při intervalech 1 ms se hodnoty latence přibližují k rovnosti obou plánovačů a v případě nízkého počtu jader (do 16 jader) vykazuje EEVDF vyšší latence než CFS. Přítomná je i korekce při vyšším počtu jader zmíněna výše označena přerušovanou čarou v barvě příslušných dat.

Testy byly následně provedeny na nevirtualizovaných prostředích. Použit byl procesor Intel Celeron J4125 se 4 jádry bez SMT, AMD Ryzen 5 5600 disponující 6 jádry a celkově s 12 vlákny a systémy s jedním a dvěma procesory Intel Xeon E5-2697A v4 zastupující 32 a 64 logických jader. Výsledky kopírovaly dřívější zjištění. Obecně se virtualizované prostředí projevuje o 34 % vyšší latencí. Dále rozdíl výsledku měření s intervalem 1ms a 10ms, je méně výrazný. Z toho je možné usoudit, že provádění testů na virtualizovaném prostředí je validní a v některých případech i méně náchylné na odchylky v měření z důvodu výraznějších rozdílů. Po vložení dat reprezentující nevirtualizované prostředí („Metal 1ms“ a „Metal 10ms“) do grafu poměrů obou plánovačů je pozorovatelný podobný trend jako u virtualizovaného prostředí.



Obrázek 18 Porovnání poměru latencí s nevirtualizovaným prostředím v závislosti na počet CPU

(zdroj: vlastní)

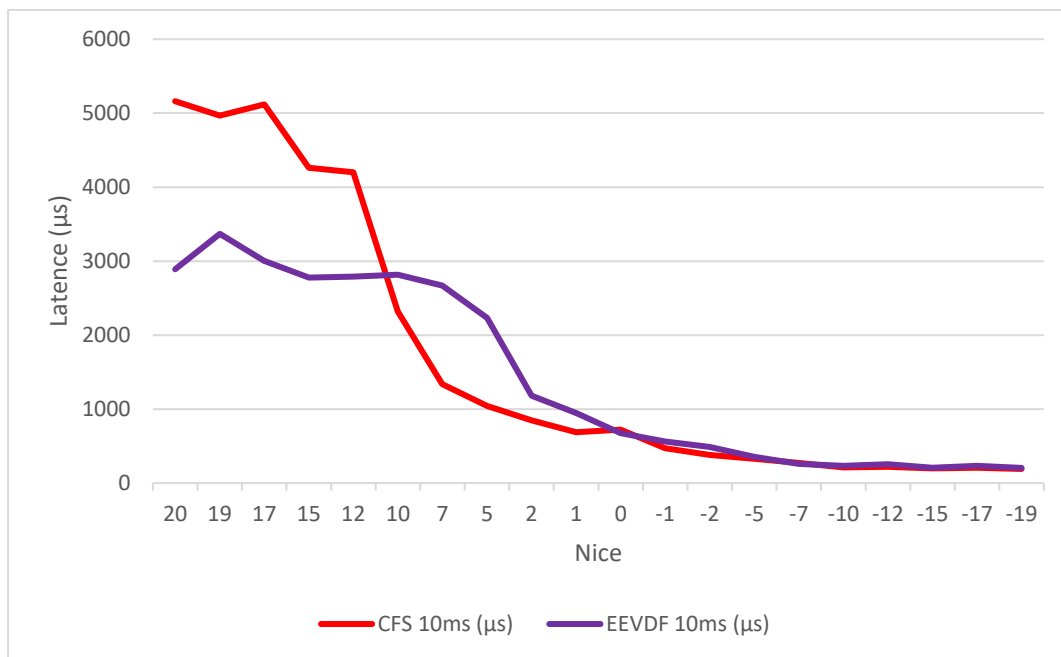
Na základě výše uvedeného lze konstatovat, že implementace plánovače EEVDF přináší výrazné zlepšení latence pro interaktivní aplikace s delšími intervaly probuzení. Nicméně pro aplikace s intervaly probuzení do 1 ms je třeba počítat s možnou regresí, zejména při nižším počtu jader.

4.3.2 Latence s parametrem nice

Následující testy byly zaměřeny na měření latence s různými hodnotami nice pro proces cyclicttest. Příkaz spouštěcí cyclicttest byl modifikován, aby bylo spuštěno měření s jedním vláknem a intervalem probuzení 1 ms a 10 ms po dobu 2 minut:

```
cyclicttest -D 2m --interval=1000
cyclicttest -D 2m --interval=10000
```

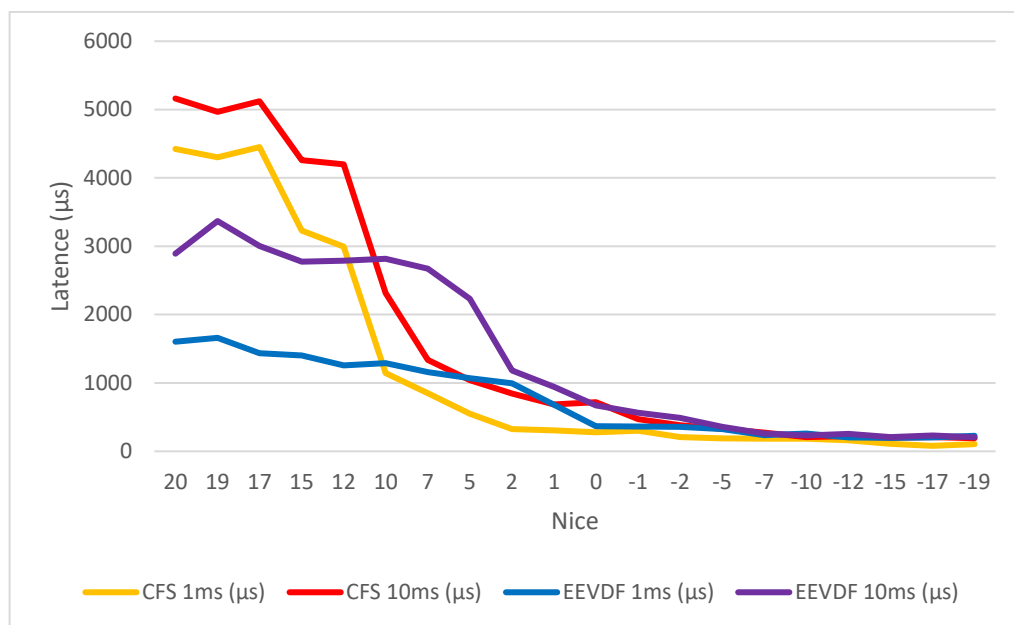
Tím se simulovaly scénáře, kdy se jedná o latence běžné úlohy, latence upřednostněné úlohy a latence úlohy se sníženou prioritou. Měření bylo provedeno na systémech s různým počtem jader. Jejich výsledky poukazyvaly na obdobné chování, avšak nejvýraznější rozdíl byl pozorovatelný při malém počtu CPU. Dále tedy bude odkazováno na měření uskutečněné se 4 jádry. Testy byly několikrát zopakovány a představené hodnoty jsou výsledkem jejich průměru. Ty byly následně zaneseny do grafu:



Obrázek 19 Průměrné latence v závislosti na hodnotách nice (interval probuzení 10 ms)

(zdroj: vlastní)

Při hodnotách nice blíží se 0, se opakuje scénář měření předcházející kapitoly. Se zvyšující se prioritou (nižší hodnota nice) se rozdíl v latencích mezi plánovači minimalizuje a oba dosahují velmi podobných výsledků, které se pohybují v mezích chyby měření a rozdíly mezi plánovacími algoritmy jsou v těchto krajních případech prakticky nerozlišitelné. Při snižující se prioritě (zvyšující se hodnota nice) je pozorovatelný dřívější nárůst a ustálení latence pro EEVDF. CFS reaguje později a ustálení se pohybuje na vyšších hodnotách latence. Data pro interval 1 ms kopíruje obdobný trend.



Obrázek 20 Průměrné latence v závislosti na hodnotách nice (interval probuzení 1 ms a 10 ms)

(zdroj: vlastní)

Je vhodné zmínit, že použitím skriptu uvedeného v kapitole 4.3 se záměrně předešlo standardního chování funkce autogroup. Za běžných okolností autogroup automaticky seskupuje procesy spuštěné z různých terminálů do samostatných skupin. Pokud by nástroje hackbench a cyclicttest byly spuštěny samostatně každý v jiném terminálu, každý proces by skončil v jiné autogroup skupině a plánovač by tím ignoroval nastavené hodnoty nice. Snažil by se stále spravedlivě přiřadit procesorový čas mezi tyto skupiny, ne mezi procesy jako takové. Jinými slovy, priority by se určovaly podle skupin, a nikoliv podle hodnot nice procesů, jejichž vliv měl být ověřen. Spuštěním obou nástrojů ze stejného skriptu bylo zajištěno, že všechny testované procesy patřily do stejné autogroup, čímž plánovač respektoval nastavené hodnoty nice, což umožnilo změřit jejich vliv na výsledné latence. [4]

4.3.3 Konzistence latence

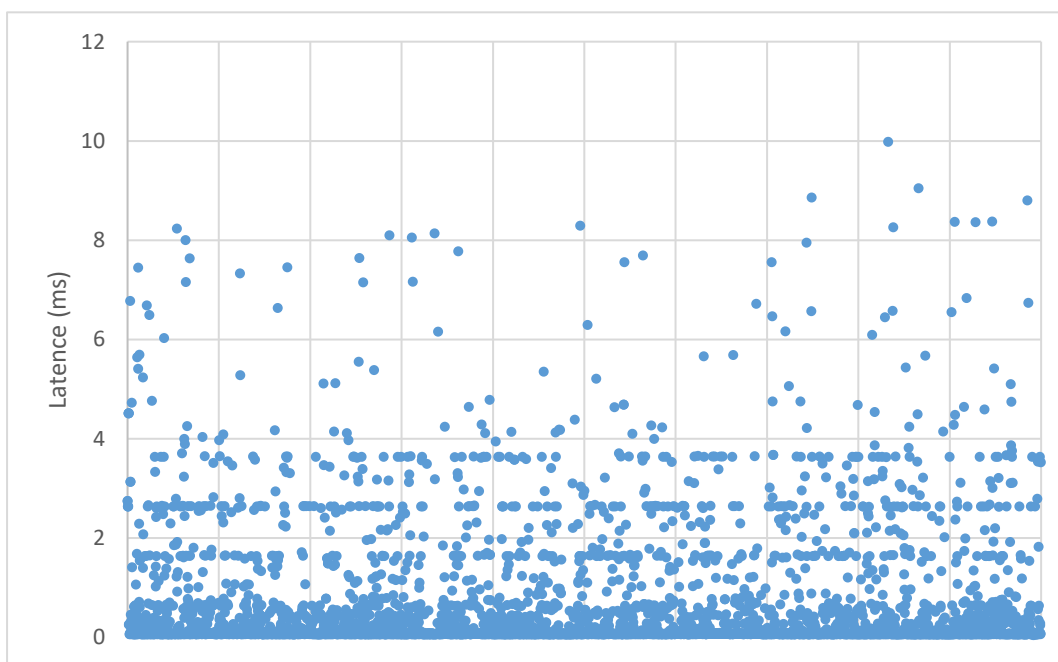
Následující vlastností k měření byla konzistence latence, která je snad i důležitějším ukazatelem interaktivity a plynulosti systému pro jeho uživatele. Jedná se o snahu minimalizovat výrazné výkyvy latence do vyšších hodnot. Objektivně je možné naměřit reakční dobu lidského mozku na vizuální podnět pohybující se v blízkosti 100 ms. Následná doba reakce na tento stimulus se může pohybovat od 150 ms do 200 ms. [34] Pro plánovače procesů je hranice přijatelné latence výrazně nižší. Například Con Kolivas použil pro BFS přijatelnou hranici latence 7 ms. Tvrdí, že nad touto hranicí je pro uživatele možné všimnout

si neplynulosti. [30] Z informací, které vycházely z hlášených problémů, CFS byly očekávány větší odchylky latence než u plánovače EEVDF.

Při měření byl cyclicttest spuštěn s parametrem `--interval=10000`, což odpovídá periodě probuzení každých 10 ms. Měření probíhalo po dobu deseti vteřin na systému disponující 12 jádry a současně byl systém zatížen nástrojem `hackbench`. Z každého cyclicttest cyklu probuzení byla pomocí nástroje „`awk`“ extrahována hodnota aktuální latence „`Act`“, tedy okamžitý rozdíl mezi očekávaným a reálným časem probuzení vlákna. Použitý příkaz:

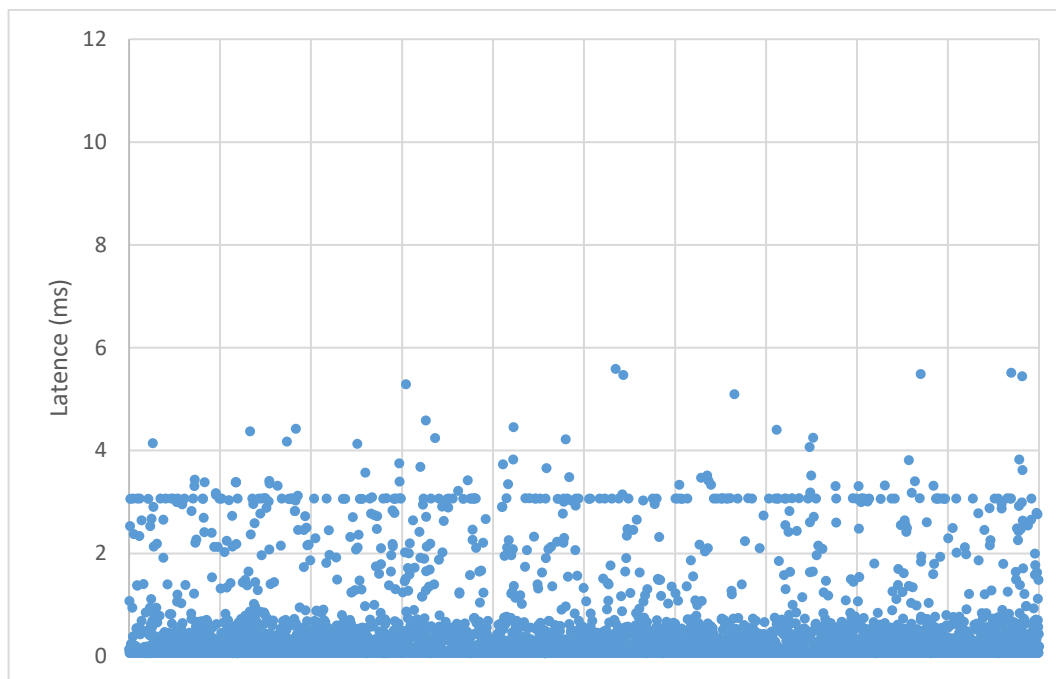
```
cyclicttest -D 60s --interval=10000 | \
awk '{for(i=1;i<=NF;i++) if($i=="Act:") print $(i+1)}'
```

Tyto hodnoty byly následně zaneseny do grafu pro vizuální porovnání.



Obrázek 21 Jednotlivé naměřené hodnoty latence CFS (ms)

(zdroj: vlastní)



Obrázek 22 Jednotlivé naměřené hodnoty latence EEVDF (ms)

(zdroj: vlastní)

Při analýze grafů bylo možné pozorovat několik rozdílů. U plánovače CFS se v grafu výrazněji projevoval vyšší rozptyl hodnot latence, tedy menší konzistence v čase. Naopak u plánovače EEVDF byly hodnoty latence celkově nižší a graf vykazoval vyšší konzistenci, tedy menší rozptyl hodnot. Toto zjištění odpovídá hlášeným vlastnostem plánovačů.

4.4 Analýza výsledků

Plánovač EEVDF se pro malý počet jader a krátký interval probuzení vláken (1 ms) projevuje vyšší či srovnatelnou latencí s CFS. Rozdíl nastává při delších intervalech probuzení (10 ms), kde EEVDF jednoznačně vykazuje zlepšení. Výsledky prezentované na LKML, které poukazovaly na přibližně 54% zlepšení, byly prováděny na stroji obsahující 480 logických jader. S přihlédnutím k výraznějšímu rozdílu plánovačů u virtualizovaného prostředí a trendu latence se zvyšujícím počtem jader je možné tuto informaci potvrdit.

Pozorovatelný rozdíl je také při obsluze hodnot nice. Zde je patrný zásadnější dopad na latenci při použití CFS a vysokých nice hodnot.

Z měření konzistence latence se EEVDF pro několik opakovaných měření projevuje menším rozptylem hodnot poukazující na vyšší konzistentní chování.

V této části byla porovnána poslední verze jádra obsahující algoritmus CFS (6.5.13) a první oficiální vydání jádra s EEVDF (6.6). Na implementaci a vyladění tohoto nového scheduleru se nadále pracuje a již existují další iterace, které představují nová zlepšení. Cílem bylo posoudit reálný přínos přechodu na EEVDF. Použitím novějších verzí jádra se do výsledků představují další proměnné, které není snadné identifikovat. Použitím těchto aktuálnějších vydání jádra se mohou některé výsledky lišit.

Celkové výsledky praktické části potvrzují, že implementace plánovače EEVDF přináší reálný přínos z hlediska odezvy systému. Je nutné podotknout, že testovací praktiky nereprezentují zátěž reálných aplikací, ale pouze její simulaci. Tento způsob měření byl zvolen z důvodu opakovatelnosti a konzistence výsledků. Jiný způsob zátěže, jako například kompilace kódu, bude mít vliv na výsledek jinými požadavky na systém.

ZÁVĚR

Cílem této bakalářské práce bylo podrobně se seznámit s problematikou plánování procesů v operačním systému Linux, přičemž důraz byl kladen na popis a porovnání současně využívaných plánovacích algoritmů CFS a nového plánovače EEVDF. V rámci teoretické části práce byly představeny základní koncepty plánování procesů, historický vývoj plánovacích algoritmů v Linuxu a jejich současný stav.

Praktická část byla zaměřena na experimentální testování vlastností obou plánovačů, konkrétně jejich latence při současném zatížení systému. Nejprve bylo nutné připravit prostředí v podobě jak desktopového, tak serverového hardwaru. Výsledky měření, provedené pomocí nástrojů `cyclicttest` a `hackbench` potvrdily, že EEVDF dosahuje v některých případech výrazně lepších výsledků ve smyslu nižších a stabilnějších latencí.

Získané poznatky vedou k závěru, že implementace plánovače EEVDF do linuxového jádra přináší zásadní výhodu pro aplikace, které vyžadují nízkou latenci. Naopak plánovač CFS, ačkoliv je stále vhodný pro běžné využití s obecnými pracovními zátěžemi, může v určitých situacích dosahovat horších výsledků v oblasti odezvy.

SEZNAM POUŽITÉ LITERATURY

- [1] VOZÁR, Jiří. Performance Testing of Linux Kernel Scheduler. Online, Bakalářská práce, vedoucí Ing. Malík Viktor. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií. 2019. Dostupné z: <https://www.fit.vut.cz/study/thesis-file/21469/21469.pdf> [cit. 2024-10-23].
- [2] LINUX KERNEL ORGANIZATION. The Linux Kernel documentation. Online. 2024. Dostupné z: <https://docs.kernel.org> [cit. 2024-10-23].
- [3] LEUNG, Joseph ed. Handbook of Scheduling Algorithms, Models, and Performance Analysis. Chapman & Hall. 2004. ISBN 1-58488-397-9.
- [4] KERRISK, Michael. sched(7) - Linux manual page. man7.org. Online. 2024. Dostupné z: <https://man7.org/linux/man-pages/man7/sched.7.html> [cit. 2025-05-26].
- [5] KUMAR, Viresh. Fixing SCHED_IDLE. LWN.net. Online. 2019. Dostupné z: <https://lwn.net/Articles/805317/> [cit. 2025-05-26].
- [6] LOVE, Robert. Introducing the 2.6 Kernel. Linux Journal. Online. 2003. Dostupné z: <https://www.linuxjournal.com/article/6530> [cit. 2025-05-26].
- [7] BOVET, Daniel Pierre a CESATI, Marco. Understanding the Linux kernel. 3rd ed. Sebastopol: O'Reilly. 2006. ISBN 05-960-0565-2.
- [8] JONES, M. Inside the Linux 2.6 Completely Fair Scheduler. IBM Developer. Online. 2018. Dostupné z: <https://developer.ibm.com/tutorials/l-completely-fair-scheduler/> [cit. 2025-05-26].
- [9] SHEN, John. a LIPASTI, Mikko. Modern processor design: Fundamentals of superscalar processors. Waveland Press. 2005. ISBN 1-4786-0783-1
- [10] RED HAT, Inc. Red Hat Enterprise Linux 9.0: Monitoring and managing system status and performance. Online. 2025. Dostupné z: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/pdf/monitoring_and_managing_system_status_and_performance/Red_Hat_Enterprise_Linux-9-Monitoring_and_managing_system_status_and_performance-en-US.pdf [cit. 2025-05-26].
- [11] CORBET, Jonathan. Core scheduling. LWN.net. Online. 2019. Dostupné z: <https://lwn.net/Articles/780703/> [cit. 2025-05-26].

- [12] PENG, Liu a XIAOCHUN, Li. Using Processor Performance P-States with Linux on Intel-based ThinkSystem Servers. Lenovo Press. Online. 2024. Dostupné z: <https://lenovopress.lenovo.com/lp1946-using-processor-performance-p-states-with-linux-on-intel-based-servers> [cit. 2025-05-26].
- [13] TOLVARDS, Linus. a kol. kernel/sched/core.c. Linux kernel source tree. Online. 2025. Dostupné z: <https://github.com/torvalds/linux/blob/master/kernel/sched/core.c> [cit. 2025-05-26].
- [14] YU, Chun. Thinking about EEVDF. Online. 2024. Dostupné z: <https://chunyu.sh/blog/thinking-about-eevdf/> [cit. 2025-05-26].
- [15] CORBET, Jonathan. Group scheduling and alternatives. LWN.net. Online. 2007. Dostupné z: <https://lwn.net/Articles/224865/> [cit. 2025-05-26].
- [16] STOICA Ion. WAHAB Hussein. Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. Online. 1996. Dostupné z: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=805acf7726282721504c8f00575d91ebfd750564> [cit. 2025-05-26].
- [17] CORBET, Jonathan. An EEVDF CPU scheduler for Linux. LWN.net. Online. 2023. Dostupné z: <https://lwn.net/Articles/925371/> [cit. 2025-05-26].
- [18] SHAH, Parth a ZIJLSTRA, Peter. [RFC][PATCH 12/15] sched: Introduce latency-nice as a per-task attribute. Linux Kernel Mailing List. Online. 2023. Dostupné z: <https://lkml.indiana.edu/hypermail/linux/kernel/2305.3/08692.html> [cit. 2025-05-26].
- [19] GREGG, Brendan. Systems performance: enterprise and the cloud. Second edition. Addison-Wesley professional computing series. Boston: Addison-Wesley, 2021. ISBN 01-368-2015-8
- [20] TOLVALDS, Linus. a kol. kernel/sched/fair.c. Linux kernel source tree. Online. 2025. Dostupné z: <https://github.com/torvalds/linux/blob/master/kernel/sched/fair.c> [cit. 2025-05-26].
- [21] JYOTHISH, J. et al. On the Fairness of Linux O(1) Scheduler. International Journal of Simulation: Systems, Science & Technology. Online. 2014. Dostupné z: <https://ijssst.info/Vol-15/No-3/data/3857a668.pdf> [cit. 2025-05-26].

- [22] JELÍNEK, Lukáš. Jádru systému Linux: kompletní průvodce programátora. Computer Press. 2008. ISBN 978-80-251-2084-2.
- [23] BABAR, Yogesh. Understanding Linux Process States. Red Hat. Online. 2012. Dostupné z: https://access.redhat.com/sites/default/files/attachments/processstates_20120831.pdf [cit. 2025-05-26].
- [24] Free Software Foundation. The GNU C Library Reference Manual: Job Control Signals. Online. 2023. Dostupné z: https://www.gnu.org/software/libc/manual/html_node/Job-Control-Signals.html [cit. 2025-05-26].
- [25] KERRISK, Michael. clone(2) - Linux manual page. man7.org. Online. 2025. Dostupné z: <https://man7.org/linux/man-pages/man2/clone.2.html> [cit. 2025-05-26].
- [26] KERRISK, Michael. pthread_create(2) - Linux manual page. man7.org. Online. 2024. Dostupné z: https://man7.org/linux/man-pages/man3/pthread_create.3.html
- [27] HEGDE S. a ZIJLSTRA P. [RFC][PATCH 00/17] sched: EEVDF using latency-nice. Linux Kernel Mailing List. Online. 2023. Dostupné z: <https://lore.kernel.org/lkml/a79014e6-ea83-b316-1e12-2ae056bda6fa@linux.vnet.ibm.com> [cit. 2025-05-26].
- [28] KACUR, J. a WILLIAMS, C. rt-tests repository. Online. 2025. Dostupné z: <https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git> [cit. 2025-05-26].
- [29] GUITTOT, Vincent. [PATCH 1/6] sched: Introduce latency-nice as a per-task attribute. Linux Kernel Mailing List. Online. 2022. Dostupné z: <https://lkml.org/lkml/2022/3/11/829> [cit. 2025-05-26].
- [30] KOLIVAS, Con. MuQSS - The Multiple Queue Skiplist Scheduler by Con Kolivas. Online. 2016. Dostupné z: <http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt> [cit. 2025-05-26].
- [31] EEO, Jun. An Analysis of Skip Lists. Online. 2016. Dostupné z: <https://eugene-eeo.github.io/blog/skip-lists.html> [cit. 2025-05-26].
- [32] KOLIVAS, Con. FAQs about BFS. Online. 2016. Dostupné z: <http://ck.kolivas.org/patches/bfs/bfs-faq.txt> [cit. 2025-05-26].

- [33] CORBET, Jonathan. Completing the EEVDF scheduler. LWN.net. Online. 2024. Dostupné z: <https://lwn.net/Articles/969062/> [cit. 2025-05-26].
- [34] THORPE, Simon. The Speed of Categorization in the Human Visual System. Online. 2009. Dostupné z: <https://www.sciencedirect.com/science/article/pii/S0896627309001718> [cit. 2025-05-26].

SEZNAM OBRÁZKŮ

Obrázek 1 Implementace funkce <code>pthread_create</code>	16
Obrázek 2 Schodová priorita plánovače RSDL	29
Obrázek 3 Skiplist.....	30
Obrázek 4 Implementace funkce <code>entity_before</code> CFS	32
Obrázek 5 Red-Black Tree.....	32
Obrázek 6 Překladová tabulka váhy procesu	33
Obrázek 7 Příklad výpočtu lag, část 1.....	36
Obrázek 8 Příklad výpočtu lag, část 2.....	37
Obrázek 9 Příklad výpočtu lag, část 3.....	37
Obrázek 10 Implementace <code>entity_before</code> EEVDF	37
Obrázek 11 Implementace <code>pick_next_entity</code> EEVDF.....	38
Obrázek 12 Příklad výstupu „ <code>cyclicttest -t -d 3000</code> “	43
Obrázek 13 Testovací script <code>hackbench_cyclicttest.sh</code>	43
Obrázek 14 Výstup <code>cyclicttest</code> pro kernel 6.5.13 disponující 8 CPU.....	45
Obrázek 15 Výstup <code>cyclicttest</code> pro kernel 6.6 disponující 8 CPU.....	45
Obrázek 16 Naměřené průměrné latence (μ s) v závislosti na počtu CPU	46
Obrázek 17 Procentuální poměr latencí EEVDF ku CFS v závislosti na počet CPU.....	48
Obrázek 18 Porovnání poměru latencí s nevirtualizovaným prostředím v závislosti na počet CPU.....	49
Obrázek 19 Průměrné latence v závislosti na hodnotách nice (interval probuzení 10 ms) .	50
Obrázek 20 Průměrné latence v závislosti na hodnotách nice (interval probuzení 1 ms a 10 ms).....	51
Obrázek 21 Jednotlivé naměřené hodnoty latence CFS (ms)	52
Obrázek 22 Jednotlivé naměřené hodnoty latence EEVDF (ms)	53

SEZNAM TABULEK

Tabulka 1 Naměřené průměrné hodnoty latence v závislosti na počet CPU 46

Tabulka 2 Procentuální poměr latencí EEVDF ku CFS v závislosti na počet CPU 47

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

BFS	Brain Fuck Scheduler
CFS	Completely Fair Scheduler
CPU	Central Processing Unit
EAS	Energy Aware Scheduling
EDF	Early Deadline FirstEV
EEVDF	Earliest Eligible Virtual Deadline First
GRUB	GRand Unified Bootloader
I/O	Input/Output
LKML	Linux Kernel Mailing List
MuQSS	Multiple Queue Skiplist Scheduler
NUMA	Non-Uniform Memory Access
OS	Operační Systém
PID	Process Identifier
POSIX	Portable Operating System Interface
RB-tree	Red-Black Tree
RSDL	Rotating Staircase Deadline Scheduler
RT	Real-Time
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
TGID	Thread Group Identifier
TID	Thread Identifier
VM	Virtual Machine
ms	milisekunda
vCPU	Virtual CPU
μs	mikrosekun