VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SYSTÉM PRO OVĚŘENÍ MINIMÁLNÍCH POTŘEBNÝCH ZDROJŮ PRO BĚH APLIKACE

SYSTEM FOR VERIFYING THE MINIMUM RESOURCES REQUIRED TO RUN AN APPLICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JIŘÍ ŽÁK
AUTHOR

VEDOUCÍ PRÁCE doc. RNDr. PAVEL SMRŽ, Ph.D.
SUPERVISOR

BRNO 2022

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Žák Jiří**
Program: Informační technologie
Název: **Systém pro ověření minimálních potřebných zdrojů pro běh aplikace**
System for Verifying the Minimum Resources Required to Run an Application
Kategorie: Web
Zadání:

1. Seznamte se s vyhodnocovacími metrikami na potřebné zdroje systému, fungování operačního systému Linux a linuxového jádra.
2. Nastudujte existující řešení vyhodnocovacích metrik.
3. Navrhněte a implementujte program, která tato data změří a uloží k vyhodnocování.
4. Navrhněte a vytvořte implementaci, která tato data vyhodnotí a zjistí, jestli daný systém je vhodný pro běh aplikace.
5. Diskutujte možnosti využití programu v reálném prostředí.

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- funkční prototyp řešení

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Smrž Pavel, doc. RNDr., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstrakt

Cílem této bakalářské práce je vytvořit systém pro ověření minimálních potřebných zdrojů pro běh aplikace. Teoretická část se věnuje tématu vyhodnocovacích metrik počítačového výkonu a principu fungování operačního systému Linux. V praktická části je popsáno, jak byl vytvořen návrh a implementace celého systému, který využívá technologii BPF (Berkeley Packet Filter). Konec práce je završený testováním a vyhodnocením celé práce. Systém byl úspěšně nasazen v partnerské firmě BringAuto. Ukázalo se, že daný operační systém je dostatečně výkonný pro běh aplikací.

Abstract

The main goal of this bachelor thesis is to create a system for the verifying minimum resources required to run an application. The theoretical part deals with the topic of computer performance evaluation metrics and the principle of operation of the Linux operating system. The practical part describes how the design and implementation of the entire system, which uses BPF (Berkeley Packet Filter) technology, was created.The end of the work is completed by testing and evaluation of the whole work. The system was successfully deployed in the partner company BringAuto. It turned out, that the operating system is powerful enough to run applications.

Klíčová slova

Berkeley Packet Filter, Linux, linuxové jádro, systémové volání, vyhodnocovací metriky, C++, Python3, Git, Github

Keywords

Berkeley Packet Filter, Linux, linux kernel, system call, evaluation metrics, C++, Python3, Git, Github

Citace

ŽÁK, Jiří. *Systém pro ověření minimálních potřebných zdrojů pro běh aplikace.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Pavel Smrž, Ph.D.

Systém pro ověření minimálních potřebných zdrojů pro běh aplikace

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. RNDr. Pavla Smrže, Ph.D. Další informace mi poskytla partnerská firma BringAuto. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Jiří Žák
10. května 2022

Poděkování

Děkuji vedoucímu bakalářské práce panu doc. RNDr. Pavlu Smržovi, Ph.D. za trpělivost, ochotu, tipy a rady při konzultacích. Dále poděkování patří firmě BringAuto za poskytnutí odborné pomoci.

Obsah

1 Úvod **3**

2 Rozbor řešené problematiky **4**
2.1 Cíle měření výkonu . . . 4
2.2 Metriky výkonu . . . 5
2.2.1 Charakteristika metrik výkonu . . . 6
2.3 Měření výkonu . . . 7
2.4 Typy událostí . . . 7
2.5 Strategie měření . . . 8
2.6 Měření výkonu v posledních letech . . . 8
2.7 Trasování událostí . . . 9
2.8 Nástroje . . . 10
2.8.1 Berkeley Packet Filter . . . 10
2.8.2 bpftool - systémový nástroj . . . 12

3 Fungování operačního systému Linux **14**
3.1 Vrstvy linuxového systému . . . 14
3.1.1 Hardware . . . 14
3.1.2 Linuxové jádro . . . 14
3.1.3 Uživatelské procesy . . . 15
3.2 Sledovací bod . . . 16
3.3 Sonda jádra . . . 16
3.4 Sledování událostí . . . 17
3.5 Systémové volání . . . 18
3.6 BPF mapy . . . 19
3.6.1 Typy map . . . 20
3.7 BTF formát . . . 20
3.8 Systémový soubor - vmlinux . . . 20
3.9 Superuživatel . . . 21

4 Návrh systému **22**
4.1 Návrh generátoru . . . 23
4.2 Návrh sledovací aplikace . . . 24
4.2.1 Způsob ukládání dat . . . 25
4.3 Návrh evaluační aplikace . . . 26
4.3.1 Knihovna libevaluator . . . 26
4.3.2 Bezpečné provádění . . . 27

5 Implementace systému **28**
5.1 Nástroj na zjištění velikostí datových typů 28
5.2 Generátor 28
5.3 Sledovací aplikace 29
5.3.1 Překlad sledovací aplikace 30
5.3.2 Část aplikace v jádře 30
5.3.3 Část aplikace v uživatelském prostoru 33
5.4 Evaluační aplikace 34
5.4.1 Knihovna libevaluator 34
5.5 Git 35

6 Testování **36**
6.1 Výpočetní jednotky 36
6.1.1 Instalace potřebných knihoven 37
6.1.2 Jednotka s Ubuntu 37
6.1.3 Jednotka s Debianem 40
6.2 Problémy 43
6.2.1 Operační systém pro Raspberry Pi 43
6.2.2 Linuxové jádro OEM 43

7 Závěr **44**

Literatura **46**

A Příkazy **48**
A.1 Přeložení sledovací aplikace 48
A.2 Přeložení knihovny libevaluator pro evaulační aplikaci 48

Kapitola 1

Úvod

V dnešní době žijeme ve světě, který každý den ke svému běhu využívá informační technologie, ať už jde o průmyslová zařízení nebo osobní výpočetní techniku. Díky této provázanosti se nároky na tyto technologie mění, ať už jde o výpočetní výkon, spotřebu energie nebo speciální architekturu. Zároveň je zde důraz na cenu a efektivitu vynaložených financí. Průmyslové počítače mohou sloužit jako dobrý příklad toho, že je důležité, aby tyto počítače měly dostatečný výkon pro plnění účelu, ke kterému byly pořízeny, a aby zároveň nebyly moc drahé nebo nespotřebovávaly moc energie. Proto je dobré znát náročnost aplikací před jejich samotným nasazením v praxi a uzpůsobit hardware podle naměřených dat.

Cílem této bakalářské práce je navrhnout a implementovat systém, který sleduje a zjišťuje, zda je výpočetní platforma schopna spustit a nechat stabilně běžet všechny potřebné aplikace. Pomocí této funkce lze pak snadno vyměňovat výpočetní jednotky nebo aktualizovat operační systém bez obav o výkon a stabilitu prostředí. Toto je docíleno pomocí technologie Berkeley Packet Filter, která je dále popsána v textu práce.

Hlavní motivací pro vypracování této bakalářské práce je její reálné využití v praxi. Využití by šlo najít v partnerské firmě, která nasazuje operační systém Linux ve svých výpočetních jednotkách pro fungování autonomního systému. Zde je zásadní, aby bylo možné systém přizpůsobit hardwaru s omezenými paměťovými a výpočetními zdroji. Pomocí odhadu náročnosti běhu systému, lze optimalizovat výběr hardwaru pro daný úkol.

Současné nástroje umožňují sledovat pouze malé části linuxových operačních systémů. Tudíž je potřeba, aby byly všechny spuštěny najednou a nejlépe v jeden moment. Spuštění více programů najednou může vést k přidání další režie operačnímu systému a tím znehodnotit celé měření. Když pomineme tento fakt, tak i celkové porovnávání výsledků z jednotlivých nástrojů je těžko proveditelné. Každý z nástrojů využívá jiný způsob zpracování a ukládání dat a zjišťování, jak pracovat s výstupem jednotlivých nástrojů, by taktéž zabralo spoustu času.

V kapitole 2 je diskutováno měření výkonu počítačů, způsoby měření, následné testování systémů, nástroje využité k vypracování této bakalářské práce a způsoby, jakým se používají. Kapitola 3 se týká fungování operačního systému Linux. Jsou zde popsány všechny důležité informace, které byly využity při návrhu a samotné implementaci této práce. Návrhu a postupu při vypracování tohoto systému je věnována kapitola 4. Informace ohledně implementace s ukázkou zdrojového kódu jsou popsány v kapitole 5. Tato kapitola také obsahuje základní požadavky pro běh tohoto systému. V předposlední kapitole 6 této bakalářské práce je popsáno testování jednotlivých částí i celého systému.

Kapitola 2

Rozbor řešené problematiky

Faktické informace k rozboru problematiky pochází z [15]. Většina vědeckých a technických oborů má přesně definované nástroje a techniky pro měření nebo porovnávání jevů zájmu. V oblasti inženýrství a informatiky však existuje velmi malé množství způsobů, jak měřit něco tak zásadního, jako je výkon počítačů. Například rychlost automobilu lze měřit ve standardních jednotkách, jako je počet ujetých kilometrů za hodinu nebo počet ujetých metrů za sekundu. Díky využití standardních jednotek je umožněno snadné srovnání například rychlosti auta a letadla. Porovnávání výkonu různých systémů se ukázalo být ne tak jednoduché.

Problémy začínají nedostatečnou dohodou v tomto odvětví i se zdánlivě jednoduchými nápady, jako je nejvhodnější metrika pro měření výkonosti. V posledních několika desetiletích počítače zaznamenaly obrovské pokroky. Přesto je vyhodnocování výkonu systémů důležitou technologii pro výzkum v oblasti počítačů. Obecný problém rozvoje efektivní vyhodnocovací techniky lze vyjádřit jako hledání nejlepšího kompromisu mezi přesností a rychlostí. Tento kompromis závisí na použití vyhodnocovací metody a strategie. Správná efektivita může vést ke snížení nákladů při tvorbě systému pro běh aplikace.

Tato kapitola obsahuje přehled základních informací týkajících se počítačového výkonu, jeho měření a testování, a přehledu nástrojů.

2.1 Cíle měření výkonu

Informace o cílech měření výkonu pochází z [21]. Cíle měření výkonu závisí na zájmu, aplikaci (systému), zručnosti a znalosti člověka, který bude výkon měřit. Nicméně, základní cíle měření výkonu jsou vypsány níže.

Porovnání alternativních návrhů systému

Cílem je porovnat výkony různých systémů nebo návrhů komponent pro konkrétní aplikaci. Například výběr optimálního počtu procesorů v paralelním systému, velikost a počet disků, nebo výběr správného operačního systému. Cílem, v tomto případě, je najít nejlepší možné řešení konfigurace v uvažovaných operačních prostředcích.

Pořizování

V tomto případě je hlavním cílem najít nejefektivnější systém pro danou aplikaci. Je nezbytné zvážit přínos dražší verze daného systému, který poskytne jen malé zvýšení výkonu v porovnání s levnějším systémem.

Plánování kapacity

Plánování kapacity je speciální a zajímavé zejména pro správce zařízení, které zpracovávají data, a systémové administrátory. Cílem je se ujistit, zda budou k dispozici dostatečné zdroje pro splnění budoucích požadavků způsobem, který bude efektivní a neohrozí výkon.

Vylepšení dosavadního systému

Cílem je najít nejlepší sadu parametrů, která poskytne nejlepší výkon celého systému. Například použití rychlejšího disku nebo zvýšení velikosti vyrovnávací paměti může vést ke zvýšení výkonu.

Odladění výkonu

Někdy může nastat situace, kdy některé aplikace nebo systém fungují, ale pracují pomalu. Je tedy nezbytné analyzovat výkon a zjistit, proč program nebo systém nesplňuje dané očekávání. Pokud je příčina odhalena, může být vyřešena.

Ukázat, co systém dokáže

Cílem je ukázat uživatelům, co systém skutečně zvládne a dokáže. Toto je nezbytné pro vývoj nových počítačových komponentů, jako jsou například procesory.

2.2 Metriky výkonu

Informace o metrikách výkonu byly čerpány z [15]. Předtím, než bude pochopen jakýkoliv aspekt výkonu počítačového systému, je důležité, aby byly určeny zajímavé a užitečné věci, které budou měřeny. Základní charakteristiky počítačového systému, které je obvykle potřeba měřit, jsou:

- počet, kolikrát je událost vyvolána
- trvání nějakého časového intervalu
- velikost parametru

Například se bude počítat, kolikrát procesor vytvoří žádost na vstup a výstup. Také by mohlo být měřeno, jak dlouho vykonání každé žádosti trvá. Další užitečná hodnota může být určování počtu přenesených a uložených bitů. Z těchto naměřených hodnot může být odvozena skutečná hodnota, která bude využita pro popis výkonu systému. Tato hodnota se nazývá výkonnostní metrika.

Pokud měření výkonu bude zaměřeno na hodnotu času, počet nebo velikost, lze tuto hodnotu využít přímo jako metriku výkonu. Výběr vhodné metriky pro měření výkonu závisí na cílech pro konkrétní situace a na nákladech na shromažďovaní potřebných informací.

2.2.1 Charakteristika metrik výkonu

Existuje mnoho různých metrik, které jsou využity k popisu počítačového systémového výkonu. Některé z těchto metrik se běžně používají v celém tomto odvětví. Například jimi jsou MIPS[1] a MFLOPS[2]. Některé metriky jsou vytvořeny pro nové specifické situace tak, jak je potřeba. Zkušenosti ukázaly, že ne všechny metriky jsou vhodné. Tím je myšleno, že použití konkrétních metrik může vést k zavádějícím, až chybným závěrům.

Metrika výkonu, která splňuje všechna následující kritéria, je obecně užitečná pro umožnění porovnání různých měření. Tato kritéria byla vyvinuta pozorováním výsledků z mnoha analýz výkonnosti během let. Použití metrik výkonu, které nesplňují tyto požadavky, může často vést k chybným závěrům.

Linearita

Tato metrika říká, že hodnota metriky by měla být lineárně úměrná skutečnému výkonu systému. To znamená, že pokud hodnota metriky změní určitý poměr, tak by se měl změnit i skutečný výkon systému stejným poměrem. Například, pokud je dosavadní systém vylepšován na systém, který je dvakrát rychlejší, bude očekáváno, že stejná metrika bude také dvakrát rychlejší. Avšak ne všechny typy metriky splňují tyto požadavky. S těmito typy nelineárních metrik nemusí být nutně něco špatného. Lineární metriky jsou atraktivnější a intuitivnější při interpretaci výkonu systému.

Spolehlivost

Metrika výkonu je považována za spolehlivou, pokud systém vždy překoná jiný systém, když odpovídající metriky naznačují, že by to systém měl zvládnout. I když se toto zdá být zřejmé, tak některé z běžně používaných metrik nejsou schopné splnit tento požadavek. Konkrétně nějaký procesor má větší MIPS než jiný procesor. To nemusí nutně znamenat, že zdánlivě pomalejší procesor provede nějaký program pomaleji. Taková metrika je v podstatě k ničemu a nemá smysl ji využít k měření výkonu.

Opakovatelnost

Metrika výkonu je považována za opakovatelnou, pokud je hodnota metriky stejná při provádění stejného experimentu. Z tohoto vyplývá, že metrika je deterministická.

Snadnost měření

Pokud metriku výkonu není snadné změřit, je nepravděpodobné, že bude skutečně použita. Čím více je obtížnější změřit metriku přímo, nebo ji odvodit z jiných hodnot, tím pravděpodobnější je, že metrika bude nesprávně určena. Špatně naměřená metrika je obecně horší než špatná metrika.

---

[1] zkratka pro výpočetní výkonnost počítače podle počtu milionů zpracovaných celočíselných údajů za sekundu, které je daný procesor schopný vykonat.

[2] zkratka pro počet operací v pohyblivé řádové čárce za sekundu (FLoating-point Operations Per Second). Používá se i jako měřítko výpočetní výkonnosti počítačů.

Konzistence

Konzistentní metrika je taková metrika, jejíž jednotky a definice jsou stejné v různých systémech a různých konfiguracích stejného systému. Pokud jednotky metrik nejsou stejné, nelze tuto metriku použít při porovnávání výkonů systémů a tato metrika je nekonzistentní.

2.3 Měření výkonu

Měření výkonu je také základním elementem počítačové vědy. Dobré měření výkonu může vést k dobrému návrhu systému, jeho využití a zvýšení jeho efektivity. Naopak špatné měření výkonu může vést k nedostatečnému využití a nesplnění účelu systému [5]. Hlavní faktory, které ovlivňují výkon počítačového systému, jsou:

- návrh systému
- implementace systému
- pracovní zátěž, které je systém vystaven

K tomu také patří schopnost umět porozumět životnosti počítačového systému a jeho výpočetní techniky. Před nákupem nové výpočetní techniky je nutné zvážit tyto aspekty: jaký operační systém poběží na této výpočetní technice a jaké na něj budou požadavky [4].

Měřící technika hodnocení výkonu počítačů je považována za nejvíce důvěryhodnou, ale zároveň za nejvíce drahou. Může být implementována na skutečný systém, nebo na prototypovou verzi systému, který bude sestaven. Měření výkonu systému zahrnuje monitorování systému, zatímco je v pracovní zátěži, nebo je nad ním spuštěna sada testovacích aplikací [21].

Základem je, aby člověk, který bude měřit, rozuměl těmto hlavním konceptům, než začne měřit:

- systémové aplikace
- metriky výkonu, které budou měřeny
- hardwarové vybavení systému
- zátěž pro skutečnou aplikace
- způsob prezentace výsledků měření

Měření počítačového výkonu by nemělo být omezeno pouze na údaje o výsledku a ukazatele efektivity. Naopak by mělo obsahovat více informací o příchozí pracovní zátěži a získat přehled o příčinách výsledků [10].

2.4 Typy událostí

V měřící technikách se často měří jiné výkonnostní metriky. Toto závisí na aplikaci nebo systému, které budou testovány. Z pohledu typu události se metriky kategorizují do následujících skupin.

Metriky počtu vyvolaných událostí

Tato třída zahrnuje metriky, které jednoduše počítají počet vykonání nějaké konkrétní události. Například počet paketů, které jsou zahozeny či počet vynechání vyrovnávací paměti.

Profily

Profily v počítačových systémech představují souhrnnou metriku pro charakterizaci celého systému nebo chování aplikace. Například stupeň paralelismu představuje celkový počet aktivních procesorů v paralelním počítačovém systému v okamžiku vykonávání konkrétního aplikace.

Metriky pomocných událostí

Pomocí pomocných metrik se zaznamenávají hodnoty sekundárních systémových parametrů, když se vykoná konkrétní událost.

2.5 Strategie měření

O strategii pro sledování metrik výkonu je možné rozhodnout na základě použití výše uvedené klasifikace typu událostí. Hlavní strategie jsou následující.

Událostmi řízená strategie

Událostmi řízená strategie zaznamenává informace potřebné pro vypočítání nějaké metriky, nastane-li událost, která je spojená s danou metrikou. Takovou metrikou může například být počet vynechání vyrovnávací paměti. Toto číslo by mělo být aktualizováno vždy, když nastane příslušná událost. Po ukončení měření počítadlo vrátí svůj obsah. Výhoda této strategie je, že režie potřebná k monitorování událostí, které jsou s tím spojené, se utrácí pouze, pokud k události dojde. Tato vlastnost je však nevýhodná, pokud se události vyskytují často.

Sledovací strategie

Tato strategie se opírá o záznam sledování více událostí než pouze jediné. Tato strategie vyžaduje více úložného prostoru ve srovnání s událostmi řízenou strategii.

Nepřímá strategie

Toto schéma se využívá při měření výkonů, které nelze měřit přímo. V takovém případě je důležité najít metriku, ze které lze dané požadované metriky vyvodit.

Vzorkovací strategie

Vzorkovací strategie zaznamenává potřebné stavy systému ke zjištění metrik, které byly zvolené. Z tohoto vyplývá, že vzorkovací frekvence určuje celkovou režii a kvalitu měření. Vzorkovací frekvence se určuje podle požadovaných událostí.

2.6 Měření výkonu v posledních letech

Vzhledem k tomu, že samotné operační systémy se vyvíjí každý rok, tak i nástroje, které se používají k jejich měření, se posunuly o krok dále.

Například technologie *Berkeley Packet Filter* se v posledních letech rozšířila i do Windows od Microsoftu. Systémová volání, která používají *io_uring*, mohou přijímat a posílat

data asynchronně. Tímto se urychlí jejich volání, což může vést ke zrychlení celého operačního systému. Musí být však podporována běhovým prostředím a knihovnami [9]. K měření celkového výkonu počítačových clusterů lze použít balíček *HPL*. Tento balíček je přenosná implementace vysoce výkoného benchmarku pro počítače s distribuovanou pamětí [19].

2.7 Trasování událostí

Informace o trasování událostí byly čerpány z [21]. Trasování událostí se obecně skládá z uspořádaného seznamu událostí a jejich souvisejících proměnných.

Tyto zachycené informace se shromažďují pomocí profilovacích nástrojů a poskytují reprezentaci celkového provedení úkolu. Události trasování mohou být uspořádané instrukce provedené programem nebo systémem, sekvence adres a podobně. Události mohou být také zastoupeny v několika úrovních podrobností jako trasování samostatných událostí, jejich proměnných nebo jejich procedur. To přidává další režii zpracování. Proto je důležité poskytnout přepínače pro zapnutí nebo vypnutí trasování podle potřeby.

Trasování lze zkoumat a analyzovat a tím charakterizovat chování systému nebo programu. V simulacích, jako je simulování komunikace sítě nebo simulování vyrovnávací paměti, lze k řízení simulačních programů použít trasovací stopy. Při použití velkých trasovacích stop se často používá komprese pro urychlení procesu simulace.

Trasování lze také využít pro ověření simulačních modelů nebo vyladění algoritmů správy zdrojů. Simulace řízená trasováním má výhody, jako jsou:

- spravedlivé srovnání alternativních schémat,
- lze je snadněji ověřit,
- důvěryhodnost,
- blíží se ke skutečným podmínkám,
- podrobnost.

Systém, který je sledován, se skládá ze tří částí, které jsou pro trasování klíčové. První část je program, nebo systém, který generuje stopy. Druhá část je spotřebitel, který stopy analyzuje, třetí část je soubor, do kterého se stopy ukládají.

Jeden z hlavních problémů při generování trasování je velký objem dat vytvořený v relativně krátké době. Proto existují techniky, které se používají ke snížení velikosti dat. Tyto techniky jsou následující.

Komprese dat

Tato technika provádí kompresi jednotlivé stopy, čímž zmenší jejich celkový objem. Její potenciál je až 25% původní velikosti. Nevýhoda využití této techniky je čas potřebný ke kompresi a dekompresi stop.

Odběr vzorků stop

Cílem je uložit jen relativně malou část sledovacích sekvencí rozptýlených ve shromážděných či generovaných sledovacích datech. Tímto se uloží pouze malé části trasovaných dat. Nevýhoda je, že neexistuje žádný teoretický základ, který by pomohl rozhodnout o velikosti každého vzorku, nebo o použité vzorkovací frekvence. Tato technika se využívá při simulaci vyrovnávací paměti.

Abstraktní provedení

Toto schéma rozděluje trasování na dvě části. První je analýza programu, který má být trasován, a rozdělení jeho trasování na malé části, ze kterých se později reprodukuje celé trasování. Druhou částí je převedení malých částí trasovaných dat na úplná trasovaná data. Nevýhodou tohoto schématu je, že zpomaluje provádění sledovaného programu. Výhodou je, že toto schéma může snížit objem uložených dat, a to díky zaznamenávání informací pouze o změnách.

2.8 Nástroje

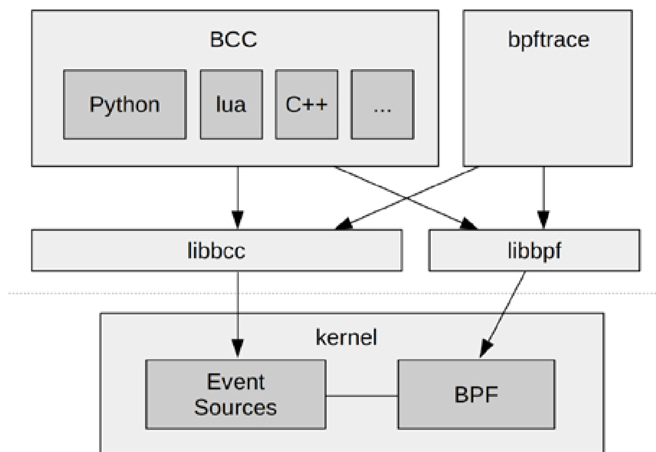
Tato sekce slouží jako přehled nástrojů, které byly využité při tvorbě této bakalářské práce.

2.8.1 Berkeley Packet Filter

Informace o Berkeley Packet Filter jsou převzaty z [8]. Berkeley Packet Filter, nebo také BPF, je technologie, která byla původně vyvinuta jako nástroj na zachytávání paketů. Poté zaznamenala zásadní změny a byla zahrnuta do linuxového jádra. Takto se z BPF stal univerzální nástroj pro spouštění, který lze využít pro mnoho věcí včetně vytváření nástrojů na analyzování výkonu. Rozšíří schopnosti jádra bez nutnosti měnit zdrojový kód jádra, díky spouštění mini programů připojených na událostech.

BPF je flexibilní a efektivní technologie složená z instrukční sady a pomocných funkcí. BPF instrukce nejprve prochází ověřovatelem, který zajistí bezpečnost a také to, že program nespadne, nebo nepoškodí jádro.

BPF je k dispozici pro většině unixových operačních systémů a extended BPF je také k dispozici pro Microsoft Windows. Je podporován mnoha programovacími jazyky, jako je například C, C++, GO, Rust, Lua nebo Python.



Obrázek 2.1: BCC, bpftrace, and BPF. Převzato z [8].

extended Berkeley Packet Filter - eBPF

Informace o extended Berkeley Packet Filteru byly čerpány z [7]. Rozšířený Berkeley Packet Filter povoluje spouštět programy v rámci operačního systému a přidávat další funkce za

běhu. eBPF se používá pro poskytování vysoce výkonných sítí a vyvažování zátěže v moderních datových centrech a cloudových nativních prostředích, pro získávání podrobných dat o pozorovatelnosti zabezpečení při nízké režii, také pomáhá vývojářům aplikací sledovat aplikace, mimo to poskytuje přehledy pro řešení problémů s výkonem a mnoho dalšího.

Funkce

Přehled základních funkcí, které BPF nabízí.

Bezpečnost

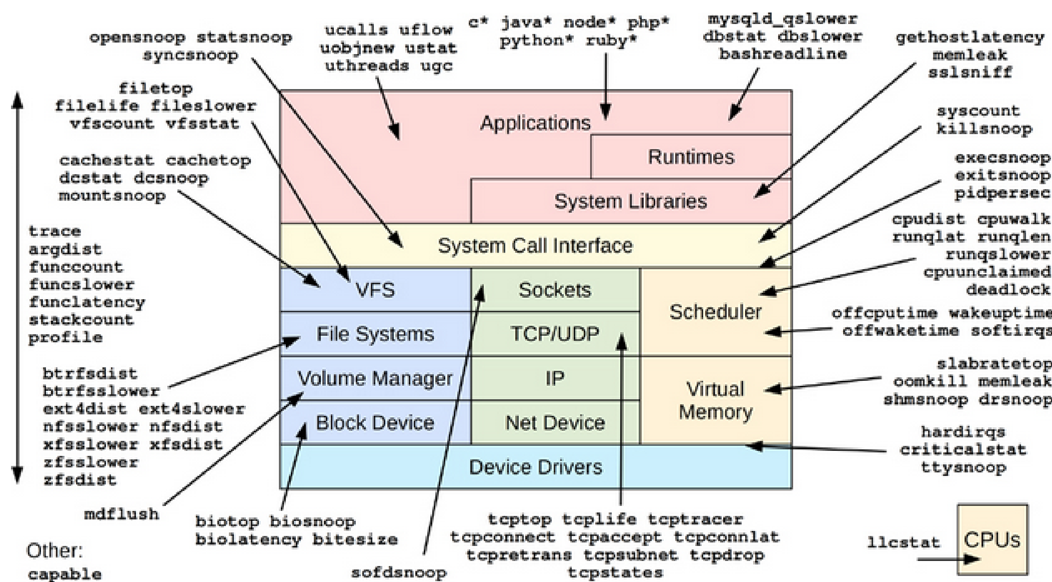
Pochopení všech systémových volání s kombinací na všechny síťové operace přináší nové přístupy k zabezpečení systému. BPF umožňuje kombinovat kontrolu všech aspektů a viditelnost za účelem tvorby bezpečnějších systémů fungující na lepší úrovni kontroly.

Sledování síťového provozu

BPF poskytuje rozhraní k datové lince, což umožňuje odesílání a přijímání paketů spojové vrstvy. BPF podporuje filtrování paketů a umožňuje uživatelským procesům, aby si samy vyfiltrovaly pakety, které chtějí obdržet a které naopak zahodit. Toto může vést ke zvýšení výkonu operačního systému, protože se tím zamezí kopírování nepotřebných paketů.

Sledování a profilování

Možnost připojovat BPF programy ke sledovacím bodům a bodům jádra umožňuje bezprecedentní sledování samostatného systému a běhového chování aplikací. Díky schopnosti pohledu na systém a aplikaci lze sledovat výkon systému. Tyto pohledy je možné i kombinovat pro ještě lepší pohled. Pokročilé provedení umožňuje extrahování pouze smysluplných dat místo exportovaní obrovského množství dat, jak to u podobných systémů bývá provedeno. Na obrázku 2.2 je vyobrazeno, jaké všechny možné části jádra systému lze sledovat a to i s příklady, co obsahují.



Obrázek 2.2: Přehled sledovatelných systémových částí. Převzato z [8].

2.8.2 bpftool - systémový nástroj

Bpftool se využívá ke spravování BPF programů. Tento program obsahuje sadu podprogramů, které zajišťují generování kódu pro BPF programy, správu a manipulaci BPF programů, tvorbu BPF iterátorů, nahrávání programu do jádra a mnoho dalšího [6].

Zobrazení BPF objektů

Tento příkaz ukáže všechny BPF objekty načtené v systému.

```
$ bpftool prog list
```

Ukázka možného výstupu:

```
3: cgroup_device tag 47dd357395126b0c gpl
       loaded_at 2022-04-19T11:53:08+0200 uid 0
       xlated 504B jited 309B memlock 4096B
4: cgroup_skb tag 6deef7357e7b4530 gpl
       loaded_at 2022-04-19T11:53:08+0200 uid 0
       xlated 64B jited 54B memlock 4096B
5: cgroup_skb tag 6deef7357e7b4530 gpl
       loaded_at 2022-04-19T11:53:08+0200 uid 0
       xlated 64B jited 54B memlock 4096B
6: cgroup_device tag b73cbcf8b8c71a5b gpl
       loaded_at 2022-04-19T11:53:08+0200 uid 0
       xlated 496B jited 307B memlock 4096B
7: cgroup_skb tag 6deef7357e7b4530 gpl
       loaded_at 2022-04-19T11:53:08+0200 uid 0
       xlated 64B jited 54B memlock 4096B
8: cgroup_skb tag 6deef7357e7b4530 gpl
```

```
       loaded_at 2022-04-19T11:53:08+0200 uid 0
       xlated 64B jited 54B memlock 4096B
9: cgroup_device tag ee0e253c78993a24 gpl
       loaded_at 2022-04-19T11:53:09+0200 uid 0
       xlated 416B jited 255B memlock 4096B
```

Generování souboru

Tento příkaz přečte soubor vmlinux a vygeneruje hlavičkový soubor vmlinux.h, jenž obsahuje typové definice, které jádro používá.

```
$ bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

Generování kostry souboru

Pomocí tohoto příkazu lze vygenerovat BPF kostru hlavičkového souboru pro programovací jazyk C při zadaném vstupním objektovém souboru. Tento objektový vstupní soubor musí být sestaven se speciálními argumenty.

```
$ bpftool gen skeleton track.bpf.o > track.skel.h
```

Kapitola 3

Fungování operačního systému Linux

Aby bylo možné s operačním systémem Linux pracovat, je potřeba vědět, jak vlastně celý systém funguje. Při tvorbě běžných aplikací není potřeba znát operační systém Linux nějak dokonale, ovšem pracuje-li aplikace v prostoru jádra, je nutností vědět, jak linuxový operační systém funguje, a poté vyhodnotit, zda aplikace nemůže nějakým způsobem poškodit tento systém. Aplikace, které jakkoliv pracují v prostou jádra, musí být spuštěny s příkazem sudo. Tímto způsobem aplikace získá možnost pracovat v prostoru jádra bez omezení. Tato kapitola se věnuje základnímu přehledu o fungování Linuxu, z jakých částí se skládá a jak s ním lze pracovat.

3.1 Vrstvy linuxového systému

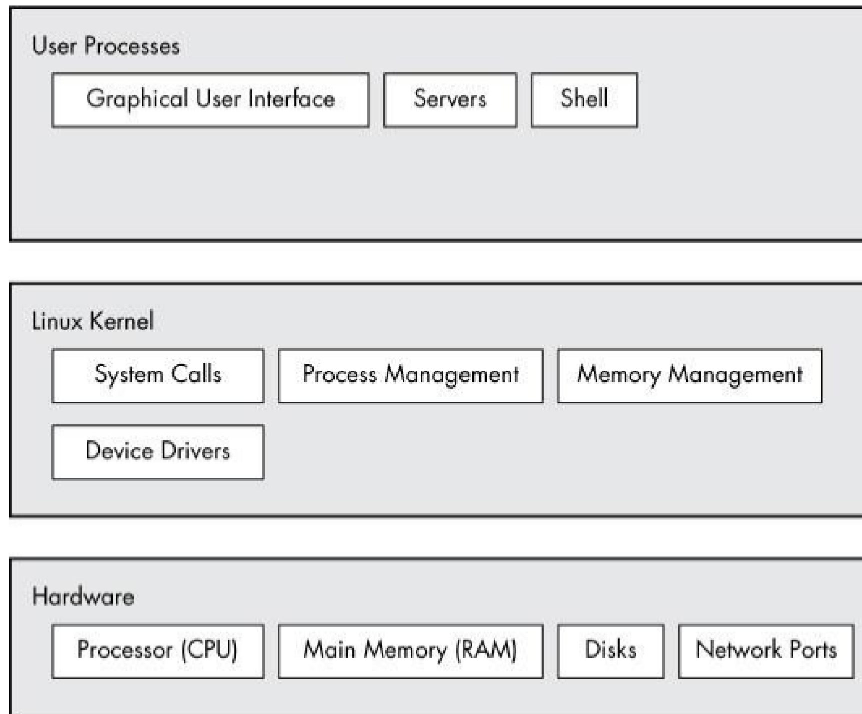
Informace o vrstvách linuxového systému jsou čerpány z [25]. Linuxový systém se skládá ze tři hlavních úrovní. Tyto úrovně jsou vyobrazeny na obrázku níže současně s jejich komponenty, ze kterých se skládají. Jak je již zmíněno výše, aplikace mohou pracovat v uživatelském prostoru, nebo v prostoru jádra.

3.1.1 Hardware

Hardware tvoří základ každé výpočetní jednotky. Zahrnuje paměť, jeden nebo více jednotek procesorů pro práci s pamětí. Dále obsahuje hlavní disk a případně i síťového rozhraní. Tato úroveň je nejnižší z těchto tří úrovní.

3.1.2 Linuxové jádro

Prostřední úrovní je linuxové jádro, které je zároveň jádrem operačního systému. Jádro je umístěné v paměti, která se také nazývá prostor jádra (angl. kernel space). Je zde uložen a vykonáván kód jádra, který taktéž řídí procesor a říká mu, co má dělat. Jádro především funguje jako rozhraní mezi hardwarem a jakýmkoliv spuštěným programem. Zároveň jádro spravuje hardware [16].



Obrázek 3.1: Obecná organizace systému Linux. Převzato z [25].

3.1.3 Uživatelské procesy

Informace o uživatelských procesech byly čerpány z [17]. Běžící programy, které jádro spravuje, tvoří nejvyšší úroveň systému. Je také nazýván uživatelským prostorem. Uživatelský prostor(angl. user space) je sada míst, kde běží uživatelské procesy(všechno ostatní kromě jádra). Jádro řídí aplikace v tomto prostoru tak, aby se nepletly mezi sebou. Procesy, které běží v uživatelském prostoru, mají omezenou část paměti a zároveň nemají přístup do prostoru jádra. Procesy běžící v uživatelském prostoru mohou přistupovat do prostoru jádra jedině přes rozhraní vystavené prostorem jádra, kterým se také říká systémová volání. Pokud proces provede systémové volání, tak se do jádra odešle systémové přerušení a jádro poté obslouží příslušný proces. Jádro pokračuje ve své práci až po dokončení přerušení.

Rozdíly

Kód běžící v prostoru jádra má neomezený přístup k hlavní paměti a procesoru. Toto je velice silné, ale zároveň velmi nebezpečné privilegium. Procesy, které zde běží, mohou snadno havarovat celý systém a tím ho nenávratně poškodit.

Naopak v uživatelském prostoru je přístup do paměti omezený na malou podmnožinu a bezpečné operace v procesoru. Jestliže proces z nějakého důvodu udělá chybu a havaruje, jeho následky jsou omezené a jádro je může vyčistit.

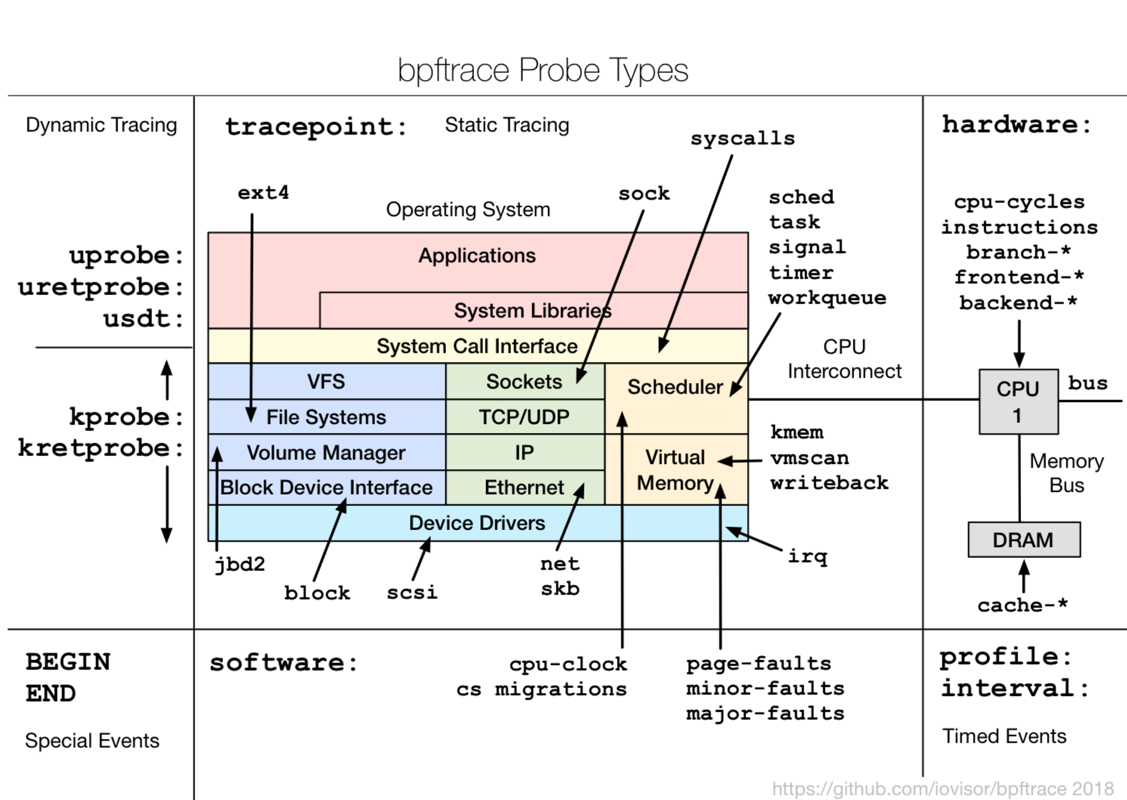
Proces v uživatelském prostoru může také způsobit poškození. Když bude mít vyšší oprávnění než jiné procesy, může třeba přepsat data uložená na disku. Proto je vždy důležité pracovat obezřetně.

3.2 Sledovací bod

Sledovací bod(angl. tracepoint) umístěný ve zdrojovém kódu poskytuje tzv. statický háček pro volání funkce(sondy) poskytnutou za běhu aplikace. Sledovací bod může být zapnutý, pokud je k němu připojena nějaká sonda, nebo vypnutý, jestliže k němu žádná sonda připojená není. Když je sledovací bod zapnutý, volá poskytnutou funkci pokaždé, když je sledovací bod spuštěn v kontextu provádění volajícího. Když poskytnutá funkce ukončí své provádění, vrací se k volajícímu (pokračuje z místa sledovacího bodu). Lze je využít pro sledování systému nebo výkonu [3].

3.3 Sonda jádra

Informace o sondách jádra pochází z [11]. Sonda jádra umožní dynamicky proniknout do rutin jádra a sbírat informace o ladění a výkonu bez přerušení. Existují dva typy sond, přičemž první se nazývá kprobe a druhá kretprobe. Jak každá z nich funguje je popsáno níže.



Obrázek 3.2: Přehled rutin, které lze sledovat sondami jádra. Převzato z [23].

Jak kprobe funguje

Jakmile je kprobe registrována, vytvoří kopii probované instrukce a nahradí první bajty instrukcí přerušení. Když procesor zasáhne instrukci bodu přerušení, předá řízení kprobe s adresou struktury a provede přidanou funkci. Poté, co je funkce provedena, tak se pokračuje následující instrukcí za bodem sondy. Kprobes mohou sondovat běžící jádro, a tím

i změnit sadu registrů a ukazatele instrukce. Operace s nimi proto vyžaduje maximální pozornost a znalosti počítačové architektury.

Jak kretprobe funguje

Při registraci kretprobe se vytvoří kprobe u vstupu do funkce a uloží kopii zpáteční adresy. Nakonec ještě nahradí zpáteční adresu skokem na uživatelsky specifikovaný kus kódu. Jakmile je provedena instrukce návratu, vyvolá se sonda a spustí uživatelsky specifikovaný, přidaný kus kódu.

Optimalizace

Pokud je jádro sestavené s příznakem *CONFIG_OPTPROBES=y* a parametr *debug.kprobes_optimization* je nastaven na 1, tak se pokusí použít instrukce skoku místo instrukcí přerušení pro každou sondu jádra a tím snížit režii probe-hit. Před pokusem o tuto optimalizaci se vloží obyčejná sonda založené na přerušení, takže, když se nepovede vložit optimalizovanou verzi sondy, sonda tam stále bude.

Kontrola bezpečnosti

Než se provede optimalizace a přidání sondy do jádra, tak se vykoná kontrola bezpečnosti. Kontrola zkontroluje, zda oblast, která bude nahrazena instrukcí skoku, leží v rámci jedné funkce (Instrukce skoku má více bajtů, proto může překrývat více instrukcí.). Dále analyzuje celou funkci a ověří konkrétně:

- funkce neobsahuje žádný nepřímý skok
- funkce neobsahuje žádnou instrukci, která by mohla vyvolat výjimku
- neexistuje žádný skok do optimalizované části

Černá listina

Sondy jádra nemohou sondovat samotné jádro, ale pouze jeho rutiny. Existují totiž některé funkce, které sondy nemohou sondovat. Kdyby sondy zachycovaly tyto funkce, mohlo by to způsobit rekurzivní past (Dvojitou poruchu), nebo by se nikdy nezavolal obslužný program vnořené sondy. Sondy jádra spravují tyto funkce na černou listinu. Funkce lze přidávat na černou listinu. Sondy jádra kontrolují adresu sondy s černou listinou, a pokud je adresa na černé listině, bude odmítnuta se zaregistrovat.

3.4 Sledování událostí

Informace o sledování událostí pochází z [14]. Sledovací body lze využít k připojení funkce k určité události bez vytváření vlastních modulů. Ne všechny sledovací body lze sledovat pomocí systému sledování událostí. K některým musí vývojář poskytnout části kódu, které definují, jaké informace mají být sledovány a případně ukládány.

Formát

Každá trasovatelná událost má soubor *format*, který obsahuje popis každého pole dané události. Tyto informace lze poté využít v další analýze. Dále také zobrazuje formátovací řetězec pro tisk události v textovém režimu, název události a ID pro profilování.

Pole v souboru *format* má tvar: *field:field-type field-name; offset:N;velikost:N;* Offset je posun pole v záznamu trasování a velikost je velikost dané datové položky určená v bajtech.

Příklad události (systémové volání) openat:

```
name: sys_enter_openat
ID: 623
format:
    field:unsigned short common_type; offset:0; size:2; signed:0;
    field:unsigned char common_flags; offset:2; size:1; signed:0;
    field:unsigned char common_preempt_count; offset:3; size:1;signed:0;
    field:int common_pid; offset:4; size:4; signed:1;

    field:int __syscall_nr; offset:8; size:4; signed:1;
    field:int dfd; offset:16; size:8; signed:0;
    field:const char * filename; offset:24; size:8; signed:0;
    field:int flags; offset:32; size:8; signed:0;
    field:umode_t mode; offset:40; size:8; signed:0;

print fmt: "dfd: 0x%08lx, filename: 0x%08lx, flags: 0x%08lx, mode:
0x%08lx", ((unsigned long)(REC->dfd)), ((unsigned long)(REC->filename)),
((unsigned long)(REC->flags)), ((unsigned long)(REC->mode))
```

Spouštěče událostí

Sledované události jsou vyvolávány spouštěči, kteří jsou připojeni k daným událostem. Spouštěče mohou být jednoduché příkazy v příkazové řádce či aplikaci. Kdykoliv je vyvolána sledovaná událost, která má připojený nějaký spouštěč, spustí se sada příkazů přidružených k dané události. Ke spouštěči může být navíc připojen filtr. Daná sada příkazů bude spuštěna pouze, pokud projde daným filtrem. Pokud nemá žádný filtr, tak projde vždy.

K jedné sledované události může být připojen libovolný počet spouštěčů, s výhradou omezení, které mohou mít jednotlivé příkazy.

Spouštěče událostí jsou implementovány v tzv. *měkkém* režimu. To zapříčiní, že jakmile se k trasované události připojí jeden či více spouštěčů, tak se aktivuje, i když vlastně není povolen, ale je zakázán v *měkkém* režimu. To znamená, že sledovací bod bude zavolán, ale nebude sledován, pokud to není skutečně povoleno. Tímto schématem lze vyvolávat spouštěče pro události, které nejsou povoleny, a také je jím umožněno použít aktuální implementace filtru událostí při podmíněném vyvolávání spouštěčů.

3.5 Systémové volání

Informace o systémových volání jsou převzaty z [18]. V moderních operačních systémech poskytuje jádro sadu rozhraní, pomocí které procesy běžící v uživatelském prostoru můžou interagovat se systémem. Tato rozhraní poskytují aplikacím přístup k hardwaru, což je mechanismus, díky kterému se dají vytvářet nové procesy, ale také umožňují komunikovat se stávájícími a vyžádat si další zdroje od operačního systému. Existence těchto rozhraní a fakt, že aplikace si nemohou přímo dělat to, co chtějí, je klíč ke stabilnímu systému.

Systémová volání jsou typicky přístupná přes funkce v C knihovně a mají předem definovaná chování. Mají nula a více argumentů a mohou mít i návratové hodnoty. Obvykle v datovém typu *long*. Nula je většinou značení pro úspěch a záporná hodnota značí chybu. Pokud systémové volání vrátí chybu, vloží speciální chybnou hodnotu do globální proměnné zvané *errno*. Hodnotu lze přeložit do lidsky čitelné podoby pomocí funkce *perror()*. Každé systémové volání má konvenci pojmenování, že začíná s *sys_*.

Komunikace s jádrem

Systémová volání představují vrstvu mezi hardwarem a uživatelskými procesy. Tato vrstva je primárně pro tři účely.

První účel poskytuje abstraktní hardwarovou vrstvu pro uživatelský prostor. Když aplikace zapisuje nebo čte z disku, neví, s jakým diskem vlastně pracuje, nebo jaký je souborový systém, na kterém je soubor uložen.

Druhý účel je na poskytnutí stability a zabezpečení. Tento účel zabraňuje, aby aplikace nesprávně používaly hardware, nebraly zdroje jiným aplikacím či nepřistupovaly k souborům, ke kterým nemají povolení.

Poslední účel umožňuje zpracovávat poskytnutý virtualizovaný systém. Kdyby aplikace měly možnost přistupovat ke systémovým zdrojům tak, jak se jim zachce, bylo by nemožné zpracovávat více procesů současně či zajistit stabilitu a bezpečnost operačního systému. V linuxových operačních systémech jsou systémová volání jediným legálním vstupním bodem k jádru pro uživatelské procesy.

3.6 BPF mapy

Informace o BPF mapách byly čerpány z [2] a [13]. Technika předávání zpráv k vyvolání chování v programu se široce používá v softwarovém inženýrství. Program může změnit chování jiného programu odesíláním zpráv. Toto také umožňuje výměnu informací a zajištění komunikace mezi těmito programy. Fascinující aspekt BPF je, že kód běžící v jádře a program, který načetl uvedený kód (běží v uživatelském prostoru), spolu mohou komunikovat pomocí předávání zpráv.

BPF mapy ukládají klíče a hodnoty v jádře. Jakýkoliv BPF program k nim může přistupovat, pokud o nich ví. Programy, které běží v uživatelském prostoru, mohou také přistupovat k těmto mapám pomocí deskriptorů souboru. Do map lze uložit jakýkoliv druh dat, pokud je předem známá jejich velikost. Jádro používá klíče a hodnoty jako binární bloby a nezajímá se o to, co je zde uloženo.

Verifikátor BPF obsahuje zabezpečení, které zajistí bezpečný způsob vytváření mapy a přístup k ní. Toto je pak zaručeno vždy, když se přistupuje k datům.

Mapa je definována:

- typem
- maximálním počtem elementů
- velikostí klíče v bytech
- velikostí hodnoty v bytech

```c
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 10);
    __type(value, u16);
    __type(key, u32);
} ports SEC("maps");
```

Výpis 3.1: Ukázka mapy v jazyce C

Při definici mapy tímto způsobem se používá makro, které se nazývá atribut sekce *SEC("maps")*. Toto makro řekne jádru, že tato struktura je BPF a podle toho musí být vytvořena.

3.6.1 Typy map

Dokumentace Linuxu definuje mapy jako generické datové struktury, do kterých lze ukládat různé datové typy. V průběhu let bylo vytvořeno mnoho specializovaných datových struktur, které jsou efektivnější při specifických případech použití.

Seznam hlavních struktur, které lze využít při práci s BPF:

- HASH
- ARRAY
- PERF EVENT ARRAY
- QUEUE
- STACK

3.7 BTF formát

BTF je formát pro metadata, který zakóduje informace o ladění související s programem BPF. Původně byl používán k popisu datových typů, ale byl rozšířen, aby zahrnoval další informace. Využívá se pro tisk map, podpisu funkcí, atd. Informace o řádcích pomáhají generovat zdrojový bajtový kód.

BTF specifikace obsahuje dvě části:

- BTF jádrové aplikační rozhraní
- BTF ELF souborový formát

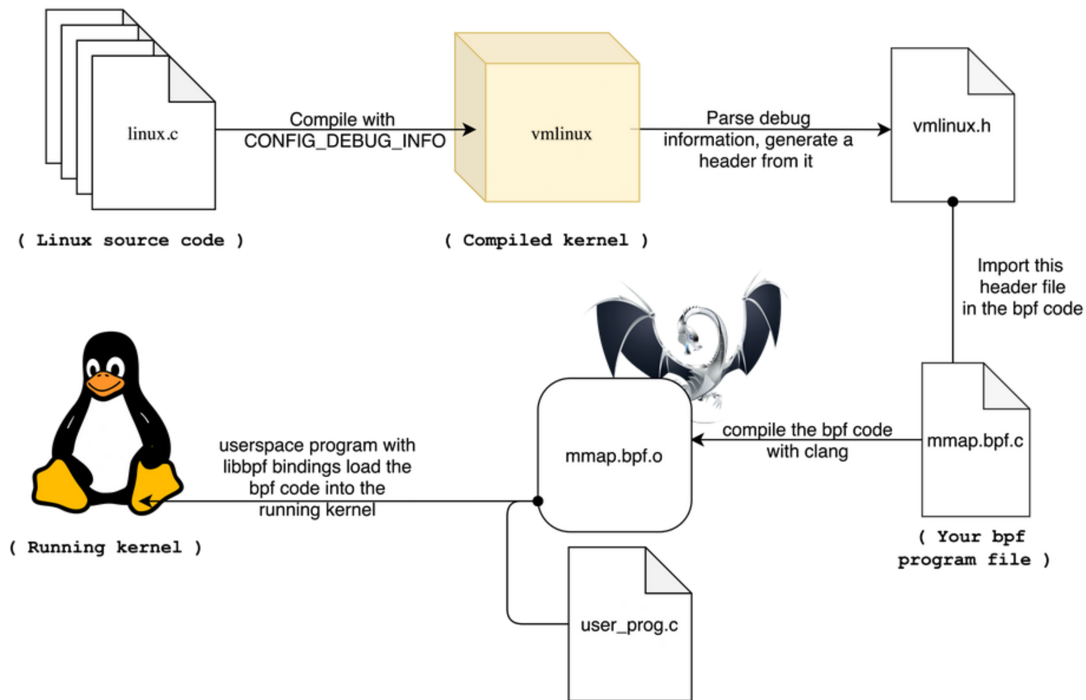
Jádrové aplikační rozhraní je rozhraní mezi uživatelským prostorem a jádrem. Jádro si samo zkontroluje informace o BTF, než informace použije.

ELF souborový formát je smlouva uživatelského prostoru mezi souborem ELF a zavaděčem programu libbpf [12].

3.8 Systémový soubor - vmlinux

Informace o systémovém souboru vmlinux jsou převzaty z [24] a [1]. Soubor vmlinux obsahuje celé zkompilované nekompresované jádro operačního systému a je uložen ve složce */sys/kernel/btf/vmlinux*. Vmlinux nepotřebuje žádné multiplatformní nástroje, aby mohl být zkompilován. Zároveň může být použit na více operačních systémech stejné architektury. Existuje i soubor vmlinuz, který se liší tím, že je komprimovaný.

Pomocí tohoto souboru lze díky nástroji bpftool vygenerovat hlavičkový soubor vmlinux.h, který je vygenerovaný z instalovaného jádra. Soubor obsahuje všechny typové definice, které jádro využívá. Tento hlavičkový soubor se využívá ke kompilování BPF programů, jako je to znázorněné na obrázku níže. Konkrétně je zahrnut do kompilování kódu pro jádro.



Obrázek 3.3: Postup kompilace BPF prgramu. Převzato z [22].

3.9 Superuživatel

V linuxových systémech lze spustit příkazy pod superuživatelem, nebo jiným uživatelem. Využívá se k zápisu či čtení paměti, ke které normální uživatel nemůže přistupovat. Příkladem může být přístup do složky */sys*. Bezpečnostní politika určuje, jaká oprávnění, pokud uživatel nějaká má, musí splnit, aby mohl příkaz spustit. Zásada může vyžadovat ověření uživatelů pomocí hesla, nebo jiným autentizačním mechanismem. Zásady zabezpečení podporují ukládání pověření do mezi paměti, aby uživatel mohl spustit sudo bez nutnosti autentizace. Toto nastavení je nastaveno na určitou dobu. Výchozí nastavení je 5 minut. Samotný příkaz sudo lze spustit s přepínači. Například pro výpis pomocné zprávy se používá přepínač *h*.

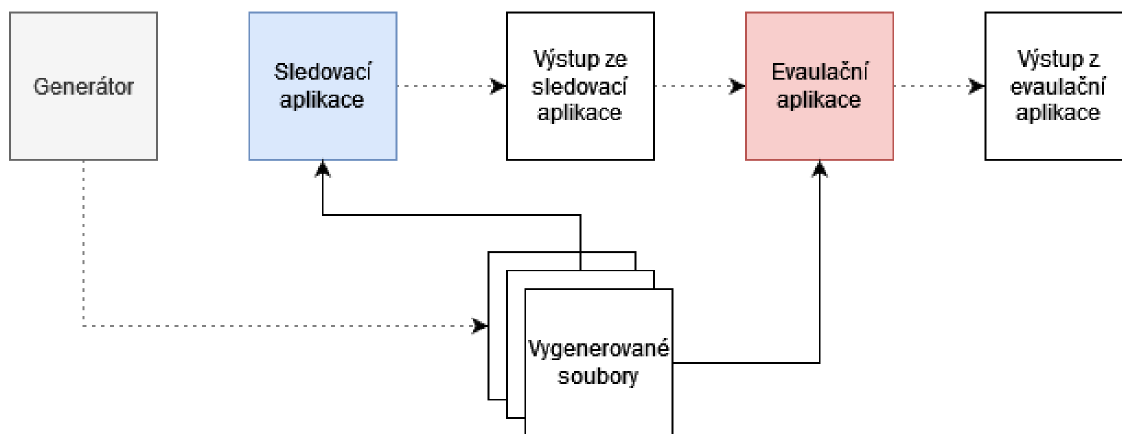
Takové příkazy se provádí pomocí použití slova *sudo* před zadávaný příkaz. Poté se uživatel musí ověřit autentizačním mechanismem nebo heslem. Jakmile je toto splněno, spustí se nová relace v terminálu pod daným uživatelem. Tuto relace lze vypnout pomocí stisknutí kláves *CTRL + C* [20].

Kapitola 4

Návrh systému

V této kapitole je popsán návrh systému, který měří a ukládá data na daném linuxovém systému, které poté vyhodnocuje. V kapitole je i často zmiňován původní návrh systému, čímž je myšlen návrh, který byl vytvořen před implementací. Během implementování systému byl tento návrh mírně upravován.

Hlavními požadavky na systém jsou snadná instalace na linuxový operační systém, stabilita a případná škálovatelnost. Snadnou instalací je míněna možnost instalace tohoto systému na linuxový operační systém. Stabilitou je myšleno, aby systém fungoval tak, jak má a aby zároveň moc nezatěžoval samotný linuxový operační systém.



Obrázek 4.1: Zjednodušené schéma systému pro sběr a vyhodnocování dat

Na obrázku 4.1 je zobrazeno jednoduché schéma navrženého systému. Systém se skládá ze tří hlavních částí. První částí je generátor, který generuje pomocné soubory pro další zbylé části systému. Druhou, a zároveň největší částí systému, je sledovací aplikace. Sledovací aplikace se stará o zachytávání dat potřebných pro vyhodnocování. Tato data jsou zapisována do souboru. Tento soubor je poté využit jako vstupní soubor pro poslední část aplikace, kterou je evaluační aplikace. Evaluační aplikace tato data postupně vyhodnocuje a vytvoří čitelný výstup pro uživatele.

4.1 Návrh generátoru

Na obrázku 4.2 je zobrazeno schéma generátoru, který generuje potřebné soubory jak pro sledovací aplikaci, tak pro evaluační aplikaci.

První částí, která je mimo samotný generátor, je nástroj na zjištění velikostí datových typů na daném operačním systému. Výstupní soubor z tohoto nástroje je pak použit jako vstupní soubor do generátoru. Tento nástroj nebyl v původním návrhu. Hlavním důvodem je, že celkový systém neví, na jakém linuxovém operačním systému je nainstalován a ani jaká je architektura tohoto systému. A protože každý linuxový operační systém může mít jiné velikosti datových typů, je potřeba, aby se velikosti datových typů zjišťovaly dynamicky. Kdyby se velikosti nezjišťovaly dynamicky, tak by pak celková aplikace nemusela pracovat správně.

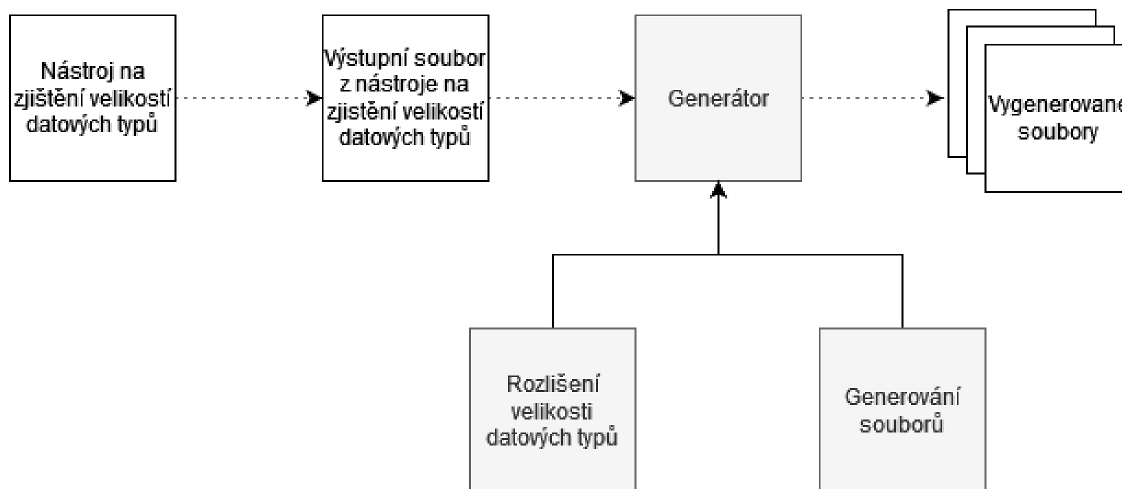
Druhou částí je pak samotný generátor, který se skládá ze dvou částí. Jak je již zmíněno, vstupním souborem do generátoru je výstupní soubor z nástroje na zjišťování velikostí datových typů. První část generátoru tyto velikosti datových typů zpracuje a rozliší do skupin. Druhá část generátoru zajišťuje samotné generování pomocných souborů.

Pomocné soubory pro sledovací aplikace

Pro sledovací aplikaci generátor vytváří pět pomocných souborů. První vygenerovaný soubor obsahuje datové struktury, které jsou použity jako vstupní parametry do funkcí, jež jsou volány při zachycení jednotlivých systémových volání. Při generování těchto struktur jsou využívané rozlišené datové typy. Druhý vytvořený soubor obsahuje enum, který souvisí s datovými strukturami v prvním souboru. Tím je myšleno, že data v enumu odkazují na struktury v prvním souboru. Třetí vygenerovaný soubor obsahuje datové struktury využité v uživatelském prostředí. Tyto tři soubory jsou hlavičkové soubory. Další dva soubory obsahují zdrojový kód. V pořadí čtvrtý vygenerovaný soubor obsahuje funkce využité v prostředí jádra. Poslední vygenerovaný soubor obsahuje funkci, která řeší jednotlivé typy vyvolaných událostí. Tyto dva soubory se zdrojovým kódem lze vygenerovat i v odlehčených verzích. Informace o daných verzích jsou popsány v implementační části této práce.

Pomocné soubory pro evaluační aplikace

Pro evaluační aplikaci generátor vytváří tři pomocné soubory. První vygenerovaný soubor je soubor s datovými strukturami, které jsou použity jako vstupní parametry do funkcí, které jsou volány při zachycení jednotlivých systémových volání. Ten je stejný jako u sledovací aplikace. Druhý vygenerovaný soubor je soubor s enumem a je taktéž stejný jako u sledovací aplikace. Poslední vygenerovaný soubor obsahuje pomocné věci k jednoduššímu čtení a rozdělování vstupních dat.



Obrázek 4.2: Schéma generátoru

V původním návrhu celého systému vytvářel generátor pouze soubor s datovými strukturami. Pro zjednodušení návrhu aplikace je vytváření více pomocných souborů nezbytné. Právě toto zároveň i zjednoduší a urychlí samotnou práci s programováním jednotlivých částí systémů. Všechny tyto vygenerované soubory lze vytvořit ručně. Bohužel toto by musel programátor dělat při každé instalaci na nějaký nový operační systém. Zároveň upravování zdrojového kódu v některých vygenerovaných souborech by bylo velice náchylné na chyby a časově by to také bylo velmi náročné.

4.2 Návrh sledovací aplikace

Na obrázku 4.3 je zobrazeno schéma sledovací aplikace, která sleduje potřebné systémové události a důležité informace o nich si zapisuje do souboru. Legenda k obrázku je, že vygenerované části systému jsou označené šedě a ty, které byly naprogramované, jsou označené oranžově. Jediný soubor, který je vygenerován jinak, je označen zeleně.

Celá sledovací aplikace se dělí na dvě části. První z nich je část aplikace, která pracuje v jádře operačního systému. Na obrázku se nachází vpravo. Druhá část aplikace pracuje v uživatelském prostředí a na obrázku se nachází vlevo. V obou částech této aplikace figurují vygenerované soubory, které byly vygenerovány generátorem zobrazeném na obrázku 4.2.

Část aplikace pracující v jádře operačního systému

První část aplikace, která pracuje v jádře operačního systému, se skládá z několika vygenerovaných souborů. Skoro celá tato část sledovací aplikace je vygenerována pomocí generátoru až na jeden soubor, který je vygenerován pomocí nástroje bpftool. Jedná se o hlavičkový soubor vmlinux. Tato část aplikace zachytává vyvolané události a ukládá je do připravených struktur. Tyto struktury jsou poté odeslány do uživatelské části aplikace.

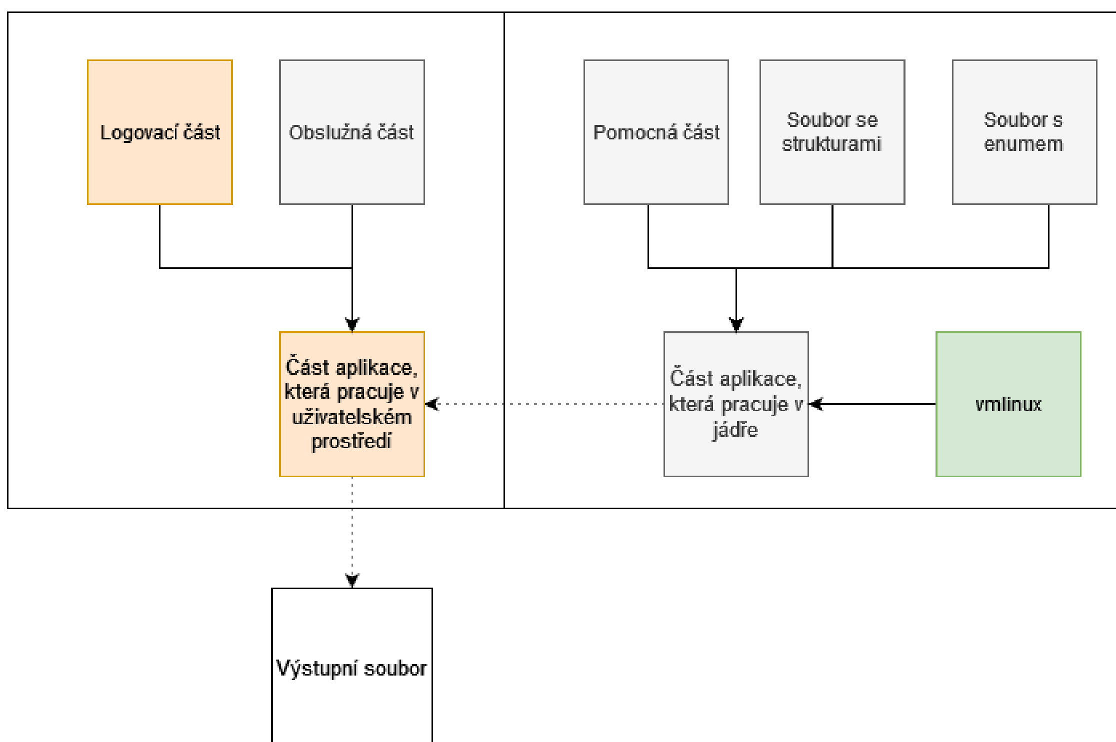
Část aplikace pracující v uživatelském prostředí

Druhá část aplikace, která pracuje v uživatelském prostředí, obsahuje pouze jeden vygenerovaný soubor z generátoru. Tento jeden vygenerovaný soubor obsahuje část kódu, která se

stará o přijímání dat od části, která pracuje v jádře operačního systému. Tato data dále rozliší podle systémové události a uloží je do výstupního souboru.

4.2.1 Způsob ukládání dat

Součástí návrhu sledovací aplikace bylo zjistit a navrhnout, jakým způsobem je nejvhodnější ukládat získaná data. Vzhledem k tomu, že ukládaná data jsou struktury, které mohou obsahovat jak jednoduché, tak i složitější datové typy, jsou všechna tato data ukládaná binárně do jednoho výstupního souboru. Hlavním důvodem použití binárního ukládání dat je, že data jsou uložena přesně za sebou, jak byla zachycena a také není potřeba rozlišovat jednotlivé datové typy. Ukládání dat je ve formátu *<časová značka><typ přijatých dat><případné další informace>*.



Obrázek 4.3: Schéma sledovací aplikace

Ve sledovací aplikaci nenastalo oproti původnímu návrhu mnoho změn. Hlavní změna je, že sledovací aplikace je z vetší části vygenerovaná než byla v původním návrhu. Hlavním důvodem tohoto kroku je možnost snadnější úpravy zdrojového kódu. Sledovaných událostí může být opravdu hodně, což generování hodně usnadňuje a zároveň zajišťuje generičnost.

Použité technologie

Při vytváření návrhu bylo současně zjišťováno, jaké technologie se na implementaci nejvíce hodí. Bylo zřejmé, že se v implementaci bude používat Berkeley Packet Filter. Berkeley Packet Filter je však implementován v mnoha programovacích jazycích. Nakonec bylo vybíráno ze tří programovacích jazyků.

Python3 a C++

První dva programovací jazyky[1], které nabízí svoji implementaci Berkeley Packet Filteru a byly brány v potaz, jsou Python3 a C++. Hlavní výhodou obou implementací je jejich jednoduché použití. Nevýhoda Pythonu bohužel je, že není dostatečně rychlý na to, aby zvládal obsluhovat tuto rozsáhlejší aplikaci. Implementace v C++ navíc neobsahuje takovou škálu funkcí a struktur, které jsou potřebné pro vytvoření sledovací aplikace. Proto jsou oba tyto jazyky pro použití programování sledovací aplikace nevhodné.

Poslední programovací jazyk, který byl brán v potaz, je jazyk C. Programovací jazyk C byl nakonec vybrán jako nejlepší volba pro sledovací aplikaci. Více informací je popsáno v implementaci.

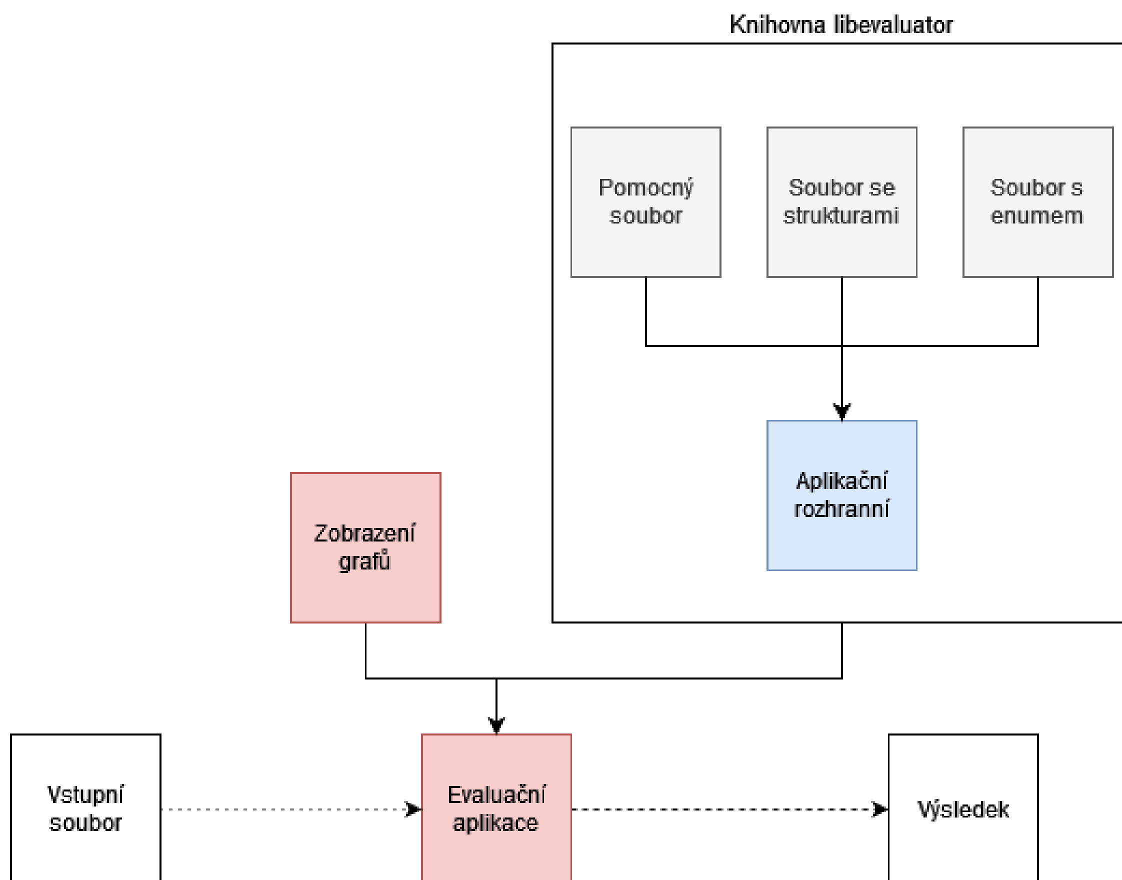
4.3 Návrh evaluační aplikace

Na obrázku 4.4 je zobrazeno schéma evaluační aplikace, která třídí jednotlivá vstupní data a následně je vyhodnocuje. Evaluační aplikace se skládá ze dvou částí. První částí je knihovna, která je zobrazena na pravé straně schématu. Druhou částí je pak evaluační aplikace rozšířená o modul, jenž vykreslí diferenciální grafy a histogramy z naměřených dat.

4.3.1 Knihovna libevaluator

Knihovna libevaluator se skládá z vygenerovaných souborů, které vygeneroval generátor, a aplikačního rozhraní, pomocí kterého lze k těmto souborům přistoupit. Dva vygenerované soubory jsou stejné jako pro sledovací aplikaci a jsou jimi soubor s datovými strukturami a soubor s enumem, který se odkazuje na tyto datové struktury. Poslední vygenerovaný soubor je pomocný soubor s velikostmi těchto datových struktur.

[1] `https://github.com/iovisor/bcc`



Obrázek 4.4: Schéma evaluační aplikace

V původním návrhu bylo naplánováno, že tato aplikace bude implementována v programovacím jazyce C++. Avšak kvůli jednodušší tvorbě a zobrazení grafů a histogramů, je evaluační aplikace implementována pomocí Pythonu ve verzi 3.

4.3.2 Bezpečné provádění

Spouštění evaluační aplikace je potřeba vykonávat na stejné výpočetní jednotce, jako na které byly zachytávány data pomocí sledovací aplikace. Je to nezbytné pro zajištění kompatibility. Generátor využívá výstupní soubor z nástroje na zjištění velikostí datových typů. Proto, kdyby byla evaluační aplikace spuštěna na jiné výpočetní jednotce, tyto velikosti by se mohly lišit, čímž by bylo vyhodnocování znehodnoceno a nemělo by žádný smysl. Je možné, že na jiné výpočetní jednotce budou tyto velikosti stejné a vyhodnocení bude provedeno správně. Toto je nutné zjistit, případně předem, bude-li vyhodnocení prováděno na jiné výpočetní jednotce.

Kapitola 5

Implementace systému

Tato kapitola popisuje implementaci jednotlivých programů, které jsou součástí tohoto systému. Mimo popisu implementační kódové části a ukázky zajímavých částí kódu se kapitola také věnuje přípravě jednotlivých částí systému, jejich instalaci a použité technologii. Návod na přeložení celého systému je přiložený v příloze.

5.1 Nástroj na zjištění velikostí datových typů

Nástroj na zjištění velikostí datových typů je velice jednoduchý a krátký program. Jak již bylo zmíněno v návrhu, aplikace je použita pro dynamické zjištění datových typů na daném linuxovém operačním systému. Nástroj je naprogramován v programovacím jazyce C. Obsahuje pouze funkcionalitu, která do souboru vypíše velikosti některých datových typů. Formát každého řádku je *datový typ:velikost*. Spouští se pomocí příkazu *./sizer*. Výstup se zapisuje do souboru s názvem *sizes*. Název výstupního souboru nelze měnit.

```
char:1
int:4
long:8
unsigned:4
unsigned long:8
void *:8
```

Výpis 5.1: Ukázka výstupního souboru

5.2 Generátor

Generátor je implementován pomocí programovacího jazyku Python ve verzi 3. Proto je nutné mít nainstalovanou tuto verzi Pythonu před prvním spuštěním. Po spuštění generátor nejdříve načte *sizes* a rozdělí si datové typy podle jejich velikosti. O toto načtení a rozdělení se postará třída *TypeResolver*. Pracovní postup generátoru lze ovlivnit pomocí argumentů. První argument *–pattern <vzor>*, nebo jeho zkrácené verze *-p <vzor>*, určuje, jaké složky bude generátor otevírat. V těchto složkách pak otevírá a načítá ze souboru *format*. Pomocí těchto načtených dat lze pak vygenerovat soubor s datovými strukturami, enumem apod. Vzhledem k tomu, že otevírané soubory jsou uložené v adresáři */sys/kernel/tracing/events/syscalls/<pattern>*, musí být generátor spuštěn s právy superuživatele,

neboť pracuje se systémovými soubory. Druhým argumentem *–name <název>*, nebo *-p <název>*, se určuje název výstupního souboru. Oba tyto argumenty jsou povinné.

Nakonec je ještě potřeba zadat jeden poslední argument z následující skupiny argumentů. Tento argument určuje, jaký soubor bude vygenerován. Při každém spuštění generátoru se vygeneruje právě jeden soubor. Generování má na starosti třída *SyscallParser*.

- structure - soubor s datovými strukturami pro systémové volání
- enum - soubor s enumem, který se odkazuje na datové struktury a makra pro velikosti datových struktur
- user - pomocný soubor pro uživatelskou část aplikace, obsahuje pomocné datové struktury
- bpf - soubor se zdrojovým kódem pro jádro operačního systému, který zachytává více informací o systémových volání
- bpf_without_data - soubor se zdrojovým kódem pro jádro operačního systému, který zachytává méně informací o systémových volání
- handler - soubor se zdrojovým kódem, který obsahuje funkci pro zpracování více příchozích dat od jádra operačního systému
- handler_without_data - soubor se zdrojovým kódem, který obsahuje funkci pro zpracování méně příchozích dat od jádra operačního systému
- helper - soubor obsahuje pole s velikostmi datových struktur a velikost enumu
- syscall_name - soubor, který obsahuje pouze jména systémových volání

Ukázka možného spuštění generátoru.

```
sudo python3 generator.py -p sys_enter_openat -n handler.c --handler
```

5.3 Sledovací aplikace

Sledovací aplikace je implementována v programovacím jazyce C s využitím technologie Berkeley Packet Filter. Tato technologie umožňuje vložení zdrojového kódu do jádra operačního systému. Sledovací aplikace při spuštění nemá žádné argumenty, ale musí být spuštěna s právy superuživatele. Spouští se příkazem *sudo ./track*. Sledovací aplikace zachytává signál *SIGINT*. Zachycení tohoto signálu je pro ni znamením, že má ukončit sledování systému a ukončit se.

Před prvním spuštěním je nutné nainstalovat následující knihovny, aby bylo možné sledovací aplikaci přeložit.

- libelf
- libbpf-dev[1]
- llvm-strip
- linux-tools-$(uname -r)[2]
- clang
- cmake

Dále je nutné zkontrolovat, zda jádro operačního systému podporuje BTF formát. Tuto kontrolu lze provést pomocí následujícího příkazu.

```
ls -la /sys/kernel/btf/vmlinux
```

[1] https://github.com/libbpf/libbpf
[2] https://linux.die.net/man/2/uname

Pokud toto není splněno, je potřeba zkompilovat jádro s příznakem *CONFIG_DEBUG_INFO_BTF=y* a poté znovu zkontrolovat, jestli jádro podporuje BTF formát.

5.3.1 Překlad sledovací aplikace

Sledovací aplikace se překládá pomocí nástroje *cmake*. V příloze je popsáno, jak aplikaci přeložit, pokud jsou již nainstalovány potřebné knihovny. Pomocí *cmaku* lze ovlivnit, jestli se budou zachytávat jen typy systémových volání, nebo i jejich další informace. V příloze je příkaz, kterým se budou zachytávat jen typy. Pro zachytávání všech informací se do příkaz musí přidat *-DDATA_SECTION=ON*. *Cmake* se poté postará, aby byl generátor volán se správnými argumenty pro vygenerování všech potřebných souborů pro spuštění sledovací aplikace. Zároveň je tím i zajištěno, že vygenerované soubory jsou spolu kompatibilní.

Dále je v příloze tento příkaz přednastaven ještě s příznakem *-DCMAKE_C_COMPILER=clang*. Tímto příznakem se nastaví to, že samotný překlad sledovací aplikace bude proveden pomocí *clangu* a nebude použit výchozí systémový překladač. Pokud je na cílovém operačním systému výchozí kompilátor *clang*, není tento příznak potřeba. Tento krok je jinak nezbytný, protože *clang* je potřebný pro překlad některých souborů s přednastavenými speciálními příznaky.

Cmake se i postará o vygenerování souboru vmlinux, nebo o vygenerování kostry programu. Přímo v souboru *CMakeLists.txt* lze nastavit vzor(angl. PATTERN), který je použit jako argument generátoru.

5.3.2 Část aplikace v jádře

V této sekci jsou popsány všechny soubory, které jsou použity pro tvorbu části sledovací aplikace, jež pracuje v jádře linuxového operačního systému. Tyto soubory jsou zároveň vygenerovány pomocí generátoru.

Hlavičkový soubor syscall_structures.h

Soubor syscall_structures.h obsahuje datové struktury pro systémové volání. Tyto datové struktury jsou využity jako vstupní argumenty funkcí, které zachytávají tato systémová volání.

```
1 struct sys_enter_openat {
2     long unusedParams;
3     int __syscall_nr;
4     long dfd;
5     void * filename;
6     long flags;
7     long mode;
8 };
```

Výpis 5.2: Ukázka vygenerované struktury v syscall_structures.h

Hlavičkový soubor syscall_enum.h

Soubor syscall_enum.h obsahuje jeden enum s názvem *Types*, který odkazuje na vygenerované struktury v souboru syscall_structures.h.

Dále soubor obsahuje následující makra. Toto makro nám říká, jak je velká daná datová struktura.

```
#define SYS_ENTER_SOCKET_LEN sizeof(struct sys_enter_socket)
```

Hlavičkový soubor user.h

Hlavičkový soubor user.h obsahuje makra a datové struktury. Jejich hlavní využití je v části sledovací aplikace, která pracuje v uživatelském prostředí.

Pomocí makra GETLEN, které se expanduje podle názvu jednotlivých systémových volání, je sledovací aplikace schopna se odkázat do souboru syscall_enum.h na velikosti struktur pro systémová volání a jejich data.

```
#define GETLEN(x) x##_LEN
```

Dále soubor user.h obsahuje jednoduchou strukturu user_type, která je využita k uložení nebo zjištění typu daného systémového volání. K tomu využívá enum ze souboru syscall_enum.h.

```
1 struct user_type {
2     enum Types type;
3 };
```

Poslední makro, které soubor user.h obsahuje, je využito pro vygenerování datových struktur jednotlivých systémových volání pro uživatelské prostředí. Předchozí datovou strukturu user_type lze použít na zjištění typu daného systémového volání. Jakmile aplikace zjistí tento typ, tak strukturu user_type může přetypovat na potřebnou vygenerovanou strukturu podle tohoto typu. Tyto vygenerované datové struktury využívají makro GETLEN pro uložení všech informací z jednotlivých systémových volání.

```
1 #define STRUCT(x) \
2     struct USER_##x { \
3         enum Types type; \
4         char data[GETLEN(x)]; \
5     };
```

Soubor track.bpf.c

Zdrojový kód ze souboru track.bpf.c pracuje přímo v jádře operačního systému. Obsahuje makro, pomocí kterého se vygenerují funkce pro zachytávání jednotlivých systémových volání. Argumentem každé funkce je datová struktura ze souboru syscall_structures.h. Tyto datové struktury jsou naplněny daty přímo z jádra operačního systému. Systémová volání jsou zachytávána na bázi statických sledovacích bodů. Zachycená data jsou poté uložena do připravených struktur ze souboru user.h a poslána do části aplikace, která pracuje v uživatelském prostředí. Data mezi oběma částmi aplikace jsou přeposílána pomocí BPF mapy nazývané ringbuffer. Pokud se ji nepovede alokovat paměť, aby mohla data odeslat, vepíše do souboru */sys/kernel/debug/tracing/trace_pipe* chybovou hlášku. Pro přístup k tomuto souboru jsou potřeba práva superuživatele.

Soubor track.bpf.c lze vygenerovat ve dvou formátech. Tyto formáty se liší pouze v tomto makru na generování funkcí pro zachytávání systémových volání.

Hlavním důvodem této možnosti je fakt, že sledovací aplikace vytvoří výstupní soubor, který obsahuje informace od jednotlivých zachycených systémových volání. Tento výstupní soubor může obsahovat několik stovek tisíc bytů vygenerované během pár sekund a zahltit tím paměť počítače.

První formát tohoto souboru zachytává jednotlivá systémová volání a do uživatelské části aplikace posílá pouze typ zachyceného systémového volání. Na řádku 5 je právě vidět, že mu k tomu stačí využít pouze jednoduchou datovou strukturu user_type ze souboru user.h. Tímto způsobem lze snížit režii ukládaných dat. To navíc přidá možnost sledovat operační systém déle.

```
1  #define FUNCTION(lower, upper) \
2      SEC("tp/syscalls/" #lower "") \
3      int handle_##lower(struct lower *params) { \
4          struct task_struct *task = (struct task_struct
               *)bpf_get_current_task(); \
5          struct user_type *data = {0}; \
6          data = bpf_ringbuf_reserve(&ring_buff, sizeof(*data), 0); \
7          if (!data) { \
8              bpf_printk("Ringbuffer not reserved\n"); \
9              return 0; \
10         } \
11         data->type = upper; \
12         bpf_ringbuf_submit(data, 0); \
13         return 0; \
14     }
```

Výpis 5.3: První formát souboru track.bpf.c

Druhý formát souboru track.bpf.c taktéž zachytává jednotlivá systémová volání. V tomto formátu nevyužívá jednoduchou datovou strukturu user_type ze souboru user.h, ale vygenerované datové struktury z tohoto souboru. Do těchto datových struktur ukládá všechny informace o jednotlivých systémových volání, ke kterým může přistoupit a ty poté posílá i s typem do části aplikace, která pracuje v uživatelské části. Tímto způsobem zachytávání dat se výstupní soubor zaplňuje podstatně rychleji. V tomto případě tak sledování systému bude trvat pravděpodobně kratší dobu, ale na druhou stranu přináší mnohem více detailnějších informací o daných systémových volání.

```
1  #define FUNCTION(lower, upper) \
2      SEC("tp/syscalls/" #lower "") \
3      int handle_##lower(struct lower *params) { \
4          struct task_struct *task = (struct task_struct
               *)bpf_get_current_task(); \
5          struct USER_##upper *data = {0}; \
6          data = bpf_ringbuf_reserve(&ring_buff, sizeof(*data), 0); \
7          if (!data) { \
8              bpf_printk("Ringbuffer not reserved\n"); \
9              return 0; \
10         } \
11         data->type = upper; \
12         bpf_probe_read_kernel(data->data, upper##_LEN, params); \
```

```
13        bpf_ringbuf_submit(data, 0); \
14        return 0; \
15    }
```

Výpis 5.4: Druhý formát souboru track.bpf.c

Každopádně, při použití obou těchto formátů se výstupní soubor plní rychle a je potřeba mít připraveno na disku hodně volného paměťového místa. Operační systémy totiž generují velké množství systémových volání, takže soubor může obsahovat několik gigabytů během několika vteřin. To však zcela závisí na testovaném systému. Pokud toto není možné zajistit, je možné zachytávání provést vícekrát a soubory si uložit někam jinam.

5.3.3 Část aplikace v uživatelském prostoru

Část sledovací aplikace, která pracuje v uživatelském prostoru, se skládá z vygenerovaných hlavičkových souborů, dále pak z přijímání dat z části sledovací aplikace, která pracuje v jádře operačního systému, a logování. Tato část aplikace se mimo jiné stará o chod celého programu a dokonce zachytává signál SIGINT.

Vygenerované hlavičkové soubory jsou stejné jako v části sledovací aplikace, která pracuje v jádře operačního systému.

Soubor handler.c

Soubor handler.c obsahuje funkci, která zachytává příchozí data z jádra operačního systému. Vzhledem k tomu, že část aplikace, která pracuje v jádře, může pracovat ve dvou formátech,je i tato funkce ve dvou formátech.

První formát této funkce zachytí příchozí data a vzhledem k tomu, že tyto data obsahují pouze typy jednotlivých systémových volání, je zavolána pouze jedna logovací funkce.

```
1  int handle(void *ctx, void *data, size_t size) {
2      struct user_type *type = (struct user_type *)data;
3      loggerLogType(&type->type, sizeof(enum Types));
4      return 0;
5  }
```

Výpis 5.5: První formát funkce pro zachytávání dat z jádra operačního systému

Druhý formát této funkce zachytává více dat. Zachycená data se skládají z typu systémového volání a dalších informací k tomuto systémovému volání. Proto je potřeba zavolat dvě rozdílné logovací funkce. Právě toto rozdělení je důležité, neboť každá datová struktura může mít jinou velikost.

```
1  int handle(void *ctx, void *data, size_t size) {
2      struct user_type *type = (struct user_type *)data;
3      loggerLogType(&type->type, sizeof(enum Types));
4      char *body = (char *)data + sizeof(enum Types);
5      switch(type->type){
6          case SYS_ENTER_SOCKET:
7              loggerLogData(body, SYS_ENTER_SOCKET_LEN);
8              break;
9          case SYS_ENTER_SOCKETPAIR:
10             loggerLogData(body, SYS_ENTER_SOCKETPAIR_LEN);
```

```
11            break;
12     }
13 }
```

Výpis 5.6: Druhý formát funkce pro zachytávání dat z jádra operačního systému

Logování

Přijatá data jsou pomocí logeru uloženy do výstupního souboru. Loger má dvě funkce, které logování zajišťují. První logovací funkce nejdříve uloží čas zachycení systémového volání a poté jeho typ. Druhá funkce ukládá pouze zbylé informace tohoto systémového volání. Zbylé informace jsou rozděleny podle jeho typu, protože každá datová struktura může mít jinou velikost. Tyto funkce zároveň musí být rozděleny, protože první formát funkce v souboru handler.c používá pouze jednu logovací funkci. Logovaná data jsou ve výstupním souboru uložena hned za sebou.

5.4 Evaluační aplikace

Evaluační aplikace je implementována pomocí jazyka Python ve verzi 3. Aby aplikace vůbec fungovala, je nutné ještě doinstalovat *tkinter* a *matplotlib*. Po spuštění evaluační aplikace zkontroluje, zda existuje vstupní soubor, který se zadává pomocí argumentu *–input*, nebo jeho kratší verze *-i*, a knihovna libevaluator. Pokud ne, tak vypíše příslušnou chybovou hlášku. Poté načte vstupní soubor a přečte data podle toho, jestli je zadaný vstupní argument *–data*, nebo *–no-data*. Rozdíl mezi těmito dvěma argumenty je v tom, že po zadání *–data* evaluační aplikace čte čas zachycení systémového volání, jeho datový typ a další příslušné informace k tomuto volání. Velikost těchto příslušných dat, kterou čte, je určena podle typu systémového volání. Pokud je zadáno *–no-data*, čte evaluační aplikace jen čas zachycení systémového volání a jeho datový typ.

Podle toho, jaké argumenty jsou zadané z následující skupiny, se vypíše výstup do terminálu, nebo vykreslí graf či histogram. O výpis a vykreslování se stará třída *Grapher*.

- hist - vykreslí histogram
- graph - vykreslí diferenciální graf
- count - vypíše, kolikrát byly zavolané všechny systémové volání dohromady
- called - vypíše, kolikrát bylo každé systémové volaní zavoláno zvlášť

Ukázka možného spuštění evaluační aplikace.

```
python3 evaluator.py -i ../build/output.bin --data --count --graph --hist
```

5.4.1 Knihovna libevaluator

Knihovna libevaluator je implementována v jazyce C. Tato knihovna obsahuje tři funkce, které jsou nezbytné, aby bylo možné přečíst vstupní soubor. První funkce vrací velikost enumu *Types*, druhá vrací velikosti jednotlivých struktur datových volání a třetí funkce vrací velikost datového typu *time_t*. Tato druhá funkce je využívána pouze, pokud je zadán argument *–data*. Toto je zároveň rozhraní, přes které lze ke knihovně přistupovat.

Překlad knihovny libevaluator

Knihovna libevaluator se překládá pomocí nástroje *cmake*. V příloze je popsáno, jak knihovnu přeložit. V souboru *CMakeLists.txt* lze nastavit vzor(angl. PATTERN), který je použit jako argument generátoru. *Cmake* se také postará, aby byl generátor volán se stejnými argumenty pro vygenerování všech potřebných souborů pro knihovnu. Tímto je i zajištěno, že vygenerované soubory jsou spolu kompatibilní. Generátor vygeneruje tři soubory. První soubor obsahuje datové struktury, druhý soubor obsahuje enum a třetí soubor obsahuje pole, které se odkazuje na velikosti datových struktur.

Spolu s knihovnou se vygeneruje ještě jeden soubor, který obsahuje pouze jména systémových volání a evaluační aplikace si ho načítá zvlášť.

5.5 Git

Po celou dobu tvorby implementační části systému byl využíván verzovací systém Git, který poskytuje jednoduchou správu verzí, a zároveň byl využíván k přenosu celého systému mezi jinými výpočetními jednotkami.

Kapitola 6

Testování

Tato kapitola popisuje postup testování celého systému i jeho jednotlivých komponent. Kapitola je také doplněna o obrázky s grafy a histogramy. Testování bylo prováděno pouze na operačních systémech zmíněných níže. Je vysoce pravděpodobné, že při instalaci a použití na jiných linuxových operačních systémech, se mohou objevit nové problémy, které v této sekci nejsou zmíněny. Pro zatížení počítačového procesoru na 100% byl využit nástroj stress[1].

6.1 Výpočetní jednotky

V této sekci jsou popsány jednotlivé výpočetní jednotky, na kterých byl systém testován. Jednotlivé podsekce obsahují informace o použitém linuxového jádře a operačním systému na dané výpočetní jednotce. Cílem je tedy otestovat celý systém jako celek na jednotlivých výpočetních jednotkách.

Pro každé měření výpočetní jednotky je sledovací aplikace přeložena podle návodu přiloženého v příloze. Tímto návodem je sledovací aplikace přeložena do prvního formátu, který zaznamenává pouze čas zachycení systémových volání a jejich typ. U jednotlivých výpočetních jednotek je podsekce s názvem *První formát*, přičemž je jím myšleno překládaní sledovací aplikace právě tímto způsobem. Pokud je název podsekce *Druhý formát*, je sledovací aplikace přeložena ještě s příznakem *-DDATA_SECTION=ON* a zaznamenává navíc datové typy jednotlivých systémových volání s jejich hodnotami. Přeložení knihovny libevaluator pro evaluační aplikace je vykonáno podle návodu a nijak se neliší pro oba tyto formáty. Poté stačí spustit sledovací aplikaci, která vytvoří vstupní soubor pro evaluační aplikaci. Než se spustí sledovací aplikace, aplikace, pro které byl tento systém vytvořen, by už měly být spuštěny. Nakonec stačí spustit evaluační aplikaci s tímto souborem a dalšími příslušnými argumenty. Výsledky jednotlivých měření jsou popsány níže.

Pro zobrazení informací o linuxovém jádře byl používán příkaz:

```
$ uname -a
```

Pro zobrazení informací o verzi operačního systému, byl používán příkaz:

```
$ lsb_release -a
```

[1] https://linux.die.net/man/1/stress

6.1.1 Instalace potřebných knihoven

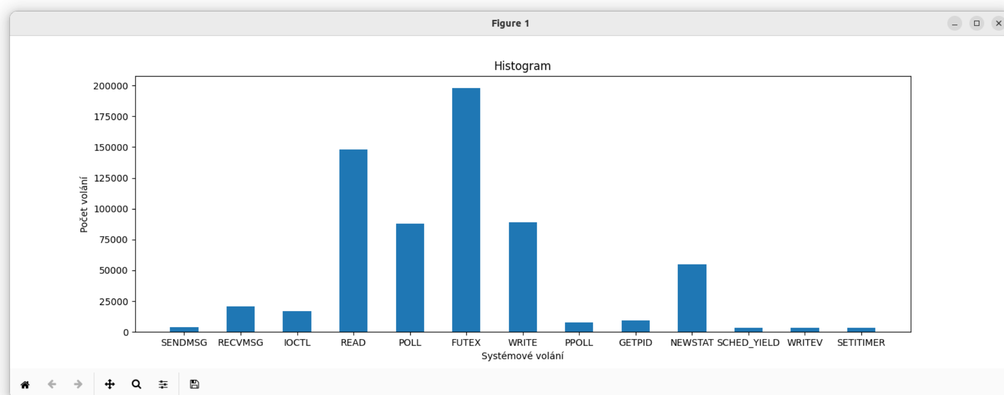
K instalaci všech potřebných knihoven 5.3 pro sledovací aplikaci a *tkinter*, který je využit v evaluační aplikaci, byl použit příkaz *apt install*. K instalaci *matplotlib* byl použit příkaz *pip3 install -r requirements.txt*. Soubor *requirements.txt* je připraven u zdrojového kódu.

6.1.2 Jednotka s Ubuntu

Na této výpočetní jednotce je nainstalován operační systém Ubuntu ve verzi 22.04 s kódovým označením *jammy*. Dále používá generické linuxové jádro verze 5.15.0-25. To, že je toto linuxové jádro generické, je důležitá informace, splňuje tím totiž podmínku v sekci 5.3. Po této kontrole lze provést instalaci potřebných knihoven pro fungování systému. Po úspěšné instalaci potřebných knihoven přichází na řadu přeložení systému. Na tomto systému v době měření běželo několik instancí Firefoxu, Visual Studio Code, Spotify a terminál, který ovládal měřící systém.

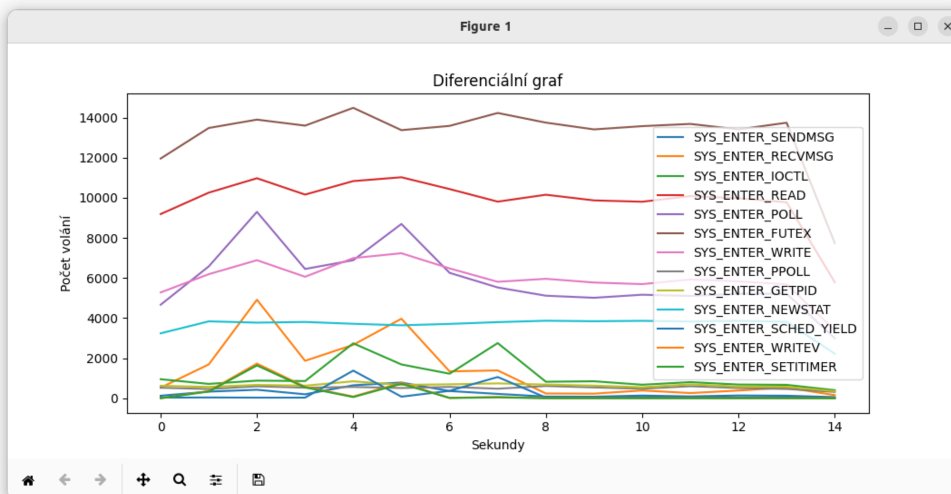
První formát

Naměřená data v tomto běhu jsou v prvním formátu sledovací aplikace. Celkový počet vyvolaných systémových volání je 18702647 za přibližných 14 sekund. Na histogramu 6.1 jsou zobrazena nejvíce vyvolávaná systémová volání během tohoto běhu. Pro zobrazení konkrétního počtu všech systémových volání je nutné použít jiný argument v evaluační aplikaci.



Obrázek 6.1: Histogram v prvním formátu na Ubuntu

Na diferenciálním grafu 6.2 je zobrazeno, ve kterých časech byla jednotlivá systémová volání vyvolávána nejvíce. Jsou zde zobrazena pouze nejvíce vyvolávaná systémová volání. Taky lze vidět, že jejich vyvolávání je konzistentní a systém je zvládá obsluhovat. Pro lepší kontrolu výsledků je proveden ještě další běh.

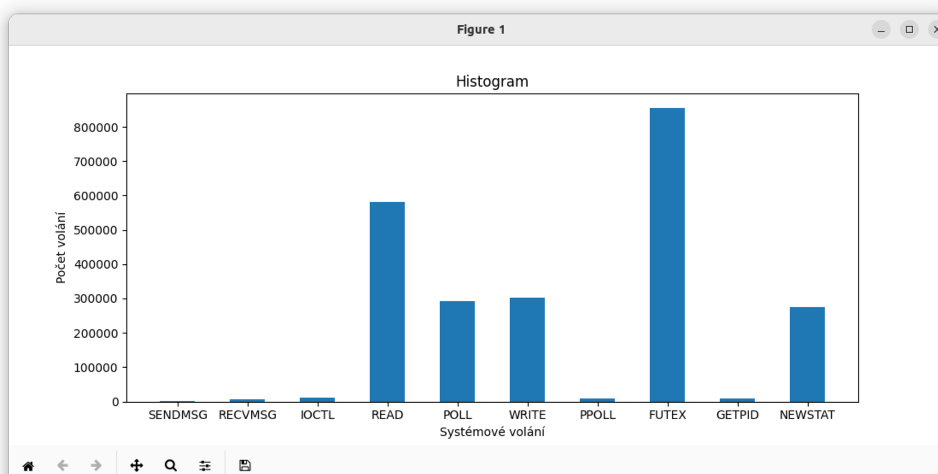


Obrázek 6.2: Diferenciální graf v prvním formátu na Ubuntu

Druhý formát

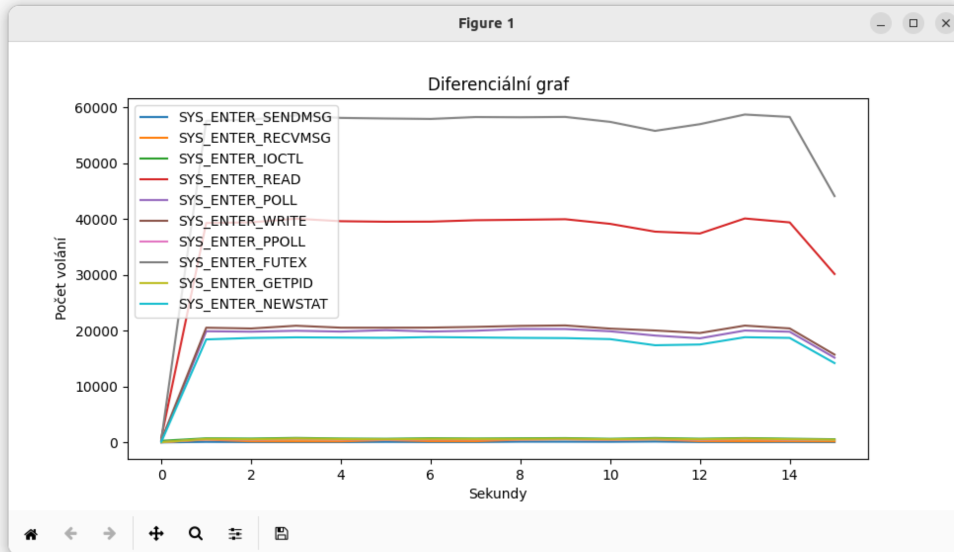
Data, která byla naměřena v tomto běhu, jsou ve druhém formátu, tudíž jsou zaznamenávány i informace navíc o jednotlivých systémových volání. Aktuální implementace evaluační aplikace s těmito daty, která obsahují více informací, zatím nijak nepracuje, je zde však prostor na případné rozšíření.

Celkový počet vyvolaných systémových volání v tomto běhu je 18690028 za přibližných 14 sekund. Na histogramu 6.3 jsou opět zobrazena systémová volání, která byly v tomto běhu nejvíce vyvolávána. Oproti prvnímu běhu je zobrazeno méně systémových volání, to i přesto, že čas měření zůstal skoro stejný.



Obrázek 6.3: Histogram ve druhém formátu na Ubuntu

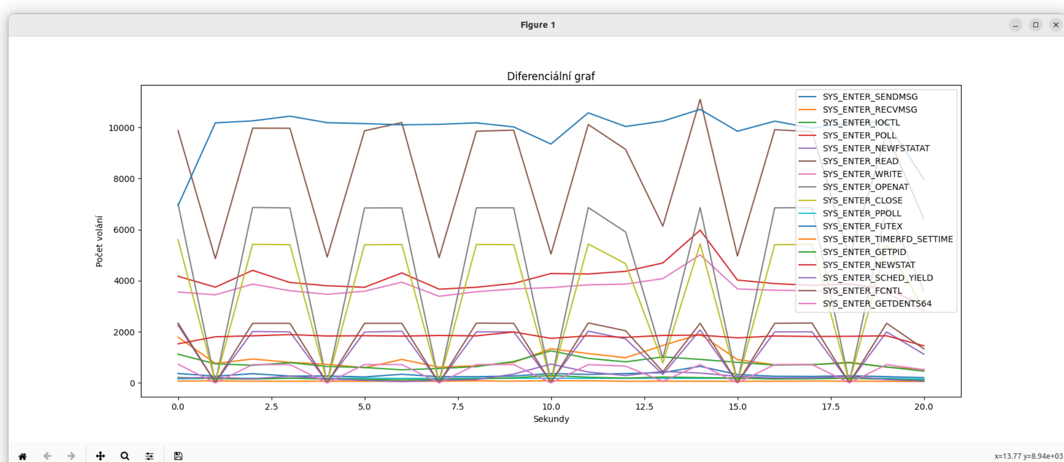
Na grafu 6.4 jsou i zde zobrazena jednotlivá systémová volání s časy, ve kterých byla nejvíce vyvolávaná. Z grafu lze poznat, že tento běh nebyl tak konzistentní jako předchozí běh. Nicméně, výkyvy nejsou zvláště rozdílné.



Obrázek 6.4: Diferenciální graf ve druhém formátu na Ubuntu

Stresové měření Ubuntu

Během stresového měření běžely všechny uvedené aplikace a navíc ještě nástroj *stress*, který se postaral o maximální vytížení procesoru. Na diferenciálním grafu 6.5 je zobrazeno zachytávání systémových volání v doby, kdy byl procesor počítače vytížen na 100%.



Obrázek 6.5: Stres test na Ubuntu

Ve srovnání s běhy, kdy na počítači běželi pouze potřebné aplikace, je zaznamenáno velké množství systémových volání během krátké doby. Konkrétně ve srovnání s prvním formátem je vidět, že systém pracoval hlavně na začátku měření a ke konci už byl více méně v klidovém režimu. Pokud je srovnán s druhým formátem, který běžel více méně konstantně, tak druhý formát nevyvolal tolik rozdílných systémových volání.

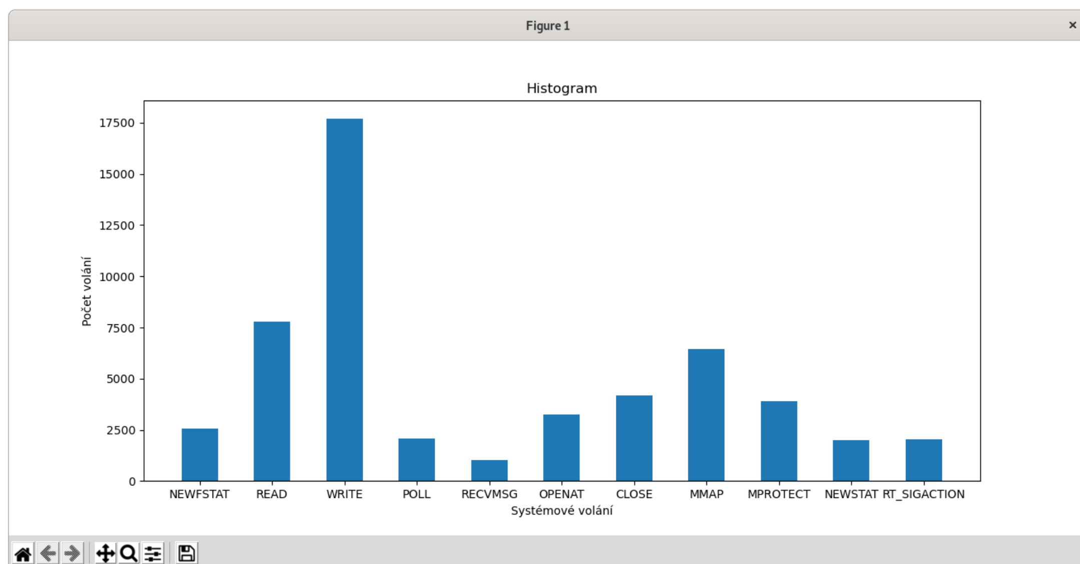
Tento systém lze tedy označit za dostatečně výkonný pro běh aplikací, které na něm běžely během bez stresových měření této výpočetní jednotky. Pro kvalitnější výsledky je lepší provést toto měření vícekrát, a to i po delší dobu. Pokud je do systému přidána nějaká nová aplikace, je důležité provést celé měření od začátku.

6.1.3 Jednotka s Debianem

Tato výpočetní jednotka má nainstalovaný operační systém Debian ve verzi 11 s kódovým označením *bullseye*. Linuxové jádro tohoto systému je ve verzi 5.10.0-14-amd64. První věc, kterou je nutné udělat, je kontrola, že jádro splňuje podmínku v sekci 5.3. Toto jádro tuto podmínku splňuje a podporuje BTF typový formát, tudíž na řadu přichází instalace všech potřebných knihoven. Po instalaci knihoven se nakonec zkompiluje celý systém a začne se se samotným měřením. Na této výpočetní jednotce byl spuštěn Firefox, Visual Studio Code a terminál. Je třeba podotknout, že tato výpočetní jednotka je pouze virtuální počítač, takže je značně omezený.

První formát, první běh

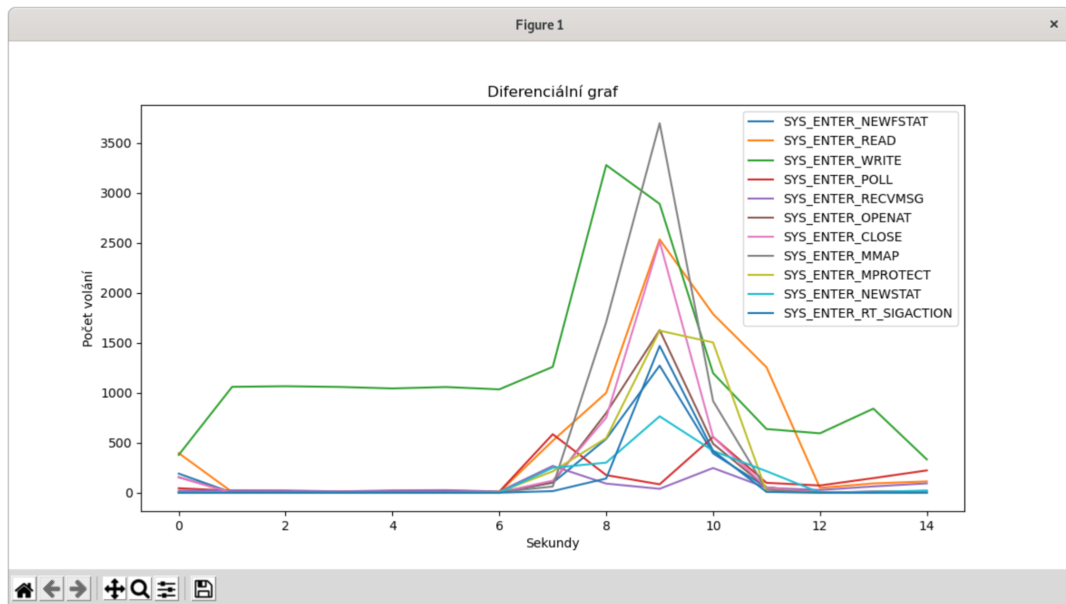
Měření tohoto systému je rozděleno na dva běhy v prvním formátu. Naměřená data zobrazená na histogramu 6.6 ukázala systémová volání, která byla nejvíce vyvolávána během tohoto běhu, ale také to, že systémové volání pro zápis bylo vyvolávané úplně nejvíce.



Obrázek 6.6: Histogram v prvním běhu na Debianu

Na diferenciálním grafu 6.7 je zobrazeno, že nejvíce vyvolávaná systémová volání nejsou v tomto běhu moc konzistentní. Navíc většina z nich byla vyvolávána ve dvou vteřinách

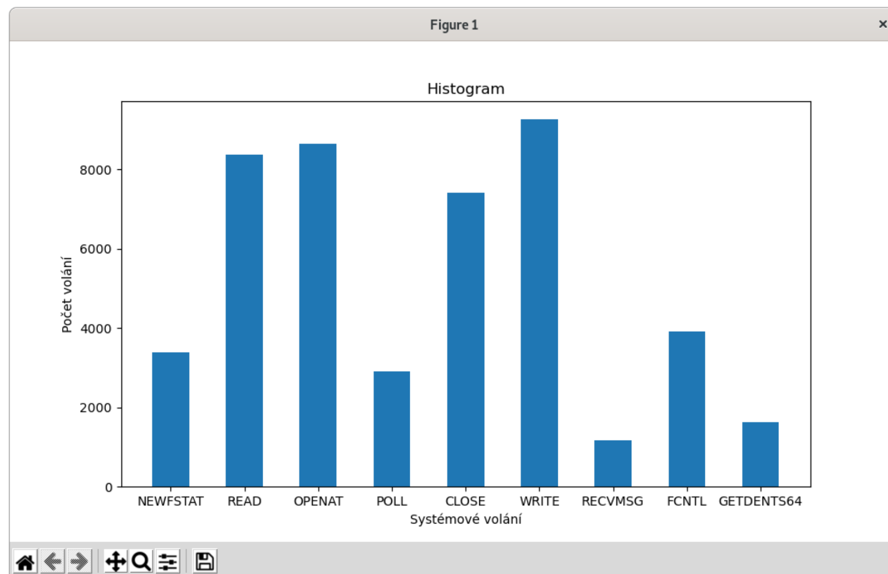
celého běhu, a to mezi vteřinami 8 a 10. Toto znamená, že systém obsluhoval nejvíce přerušení právě v tomto momentu.



Obrázek 6.7: Diferenciální graf v prvním běhu na Debianu

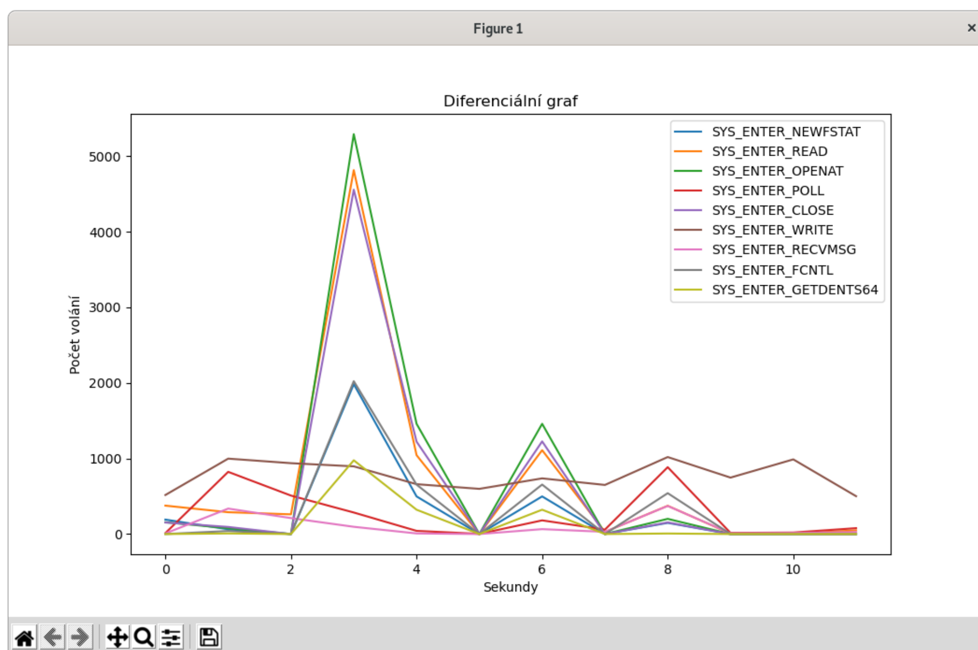
První formát, druhý běh

Vzhledem k tomu, že evaluační aplikace zatím neumí pracovat s podrobnějšími daty o jednotlivých systémových volání, bylo provedeno ještě jedno měření v prvním formátu. Toto druhé měření je označeno jako druhý běh. Na histogramu 6.8 je ukázáno, že počet jednotlivých obsloužených systémových volání bylo v podobném poměru s menšími výjimkami.



Obrázek 6.8: Histogram ve druhém běhu na Debianu

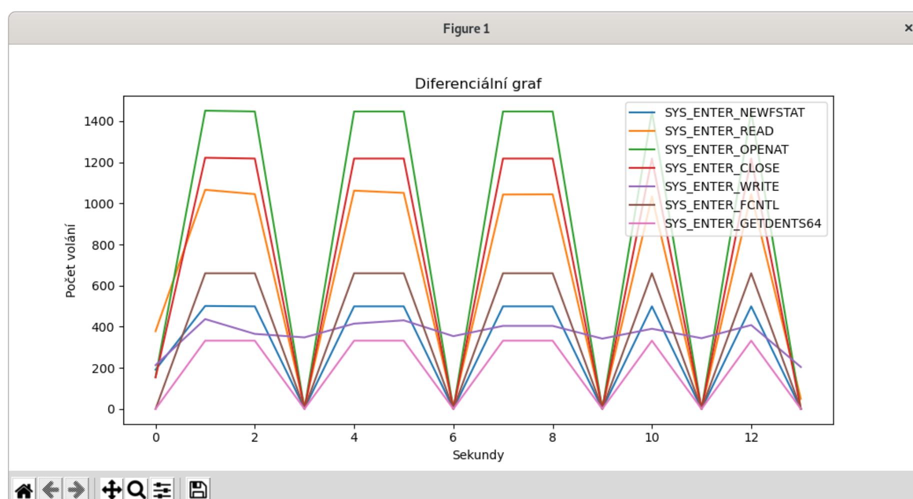
Z diferenciálního grafu 6.9 lze odvodit, že vyvolané systémové volání v jednotlivých časech jsou více konzistentní. Nicméně oproti diferenciálnímu grafu z prvního běhu 6.7 se jejich výsledky značně liší.



Obrázek 6.9: Diferenciální graf ve druhém běhu na Debianu

Stresové měření Debian

Během stresového měření byl použit nástroj *stress*, který se postaral o maximální vytížení procesoru, a všechny uvedené aplikace. Na diferenciálním grafu 6.10 je zobrazeno zachytávání systémových volání v doby, kdy byl procesor počítače vytížen na 100%.



Obrázek 6.10: Stres test na Debianu

Při porovnávání stresového měření s předchozími běhy je vidět, že běhy, kdy systém nebyl vytížen na 100%, se mu v určitých momentech blíží. Chování celkového systému je ale v každém běhu rozdílné, takže nelze s jistotou označit tento systém za dostatečně výkonný pro běh těchto aplikací. Po provedení těchto měření se tento systém jeví spíše jako nedostačující, ale pro úplnou jistotu je lepší provést další měření. Možností je systém vylepšit po hardwarové nebo softwarové stránce a celé měření provést znovu.

6.2 Problémy

Tato sekce se věnuje problémům, které byly nalezeny v průběhu měření a jeho případné přípravě.

6.2.1 Operační systém pro Raspberry Pi

Oficiální operační systém pro Raspberry Pi se nazývá Raspberry Pi OS[2]. I když je tento operační systém postavený na linuxovém jádře, tak je do značné míry upraven. Obsahuje pouze čtyři systémová volání, která se nenachází v adresáři */sys/kernel/tracing/events/syscalls/* jako v případně jiných linuxových systémů. Aktuální implementace celého systému nepodporuje jiný způsob uložení systémového volání. Problém spočívá v tom, že generátor není schopen vygenerovat potřebné soubory pro sledovací a evaluační aplikace.

6.2.2 Linuxové jádro OEM

Linuxové jádro OEM[3] je speciální odvozené jádro Ubuntu používané pro OEM projekty. OEM jádro je také využívané pro hardware, který je nový a není ještě podporovaný generickým jádrem, které se běžně používá. Toto jádro nepodporuje BTF konverzi a tudíž neobsahuje svoje zkompilované jádro - soubor vmlinux. Bohužel pro použití tohoto sytému jsou obě tyto podmínky nezbytné. Možné řešení je, že by se jádro zkompilovalo s potřebným příznakem, což však v této práci není řešeno.

[2] `https://www.raspberrypi.com/software/operating-systems/`
[3] `https://wiki.ubuntu.com/Kernel/OEMKernel`

Kapitola 7

Závěr

Cílem této bakalářské práce bylo nastudovat informace o vyhodnocovacích metrikách systému, fungování operačního systému Linux a linuxového jádra a navrhnout a zároveň vytvořit systém pro ověření minimálních potřebných zdrojů pro běh aplikace. Výsledný systém se skládá z generátoru, sledovací aplikace a evaluační aplikace. Generátor je navržen a implementován tak, aby mohl vygenerovat hlavičkové a zdrojové soubory jak pro sledovací aplikace, tak i evaluační aplikaci. Na jeho vytvoření byl použit Python ve verzi 3 pro zajištění snadné instalace a kompatibility v linuxových systémech. Sledovací aplikace je implementována tak, aby zachytávala systémová volání a do výstupního souboru zapisovala čas jejich zachycení a potřebná data o nich samotných. Zapisování do výstupního souboru je v binární podobě. Zapisování dat může fungovat ve dvou formátech. První formát uloží pouze čas zachycení systémového volání a jeho typ. Druhý formát zachytávání ukládá navíc datové typy jednotlivých systémových volání s jejich hodnotami. Při implementaci sledovací aplikace bylo důležité zvolit správný programovací jazyk, aby bylo zajištěno nejrychlejšího možného zachytávání systémových volání a jejich uložení. Proto byl zvolen a použit programovací jazyk C s technologii Berkeley Packet Filter. Evaluační aplikace je implementována tak, že otevře a čte výstupní soubor ze sledovací aplikace. Také pracuje ve dvou formátech, aby bylo možné rozlišit, zda čte pouze čas zachycení systémového volání a jeho typ, nebo ještě další informace k nim. Přečtená data jsou roztříděna a podle vstupních parametrů jsou zobrazena v terminálu, nebo vykreslena v histogramu, nebo diferenciálním grafu. Evaluační aplikace je také naprogramována v jazyce Python ve verzi 3 s využitím vykreslovací knihovny matplotlib. Tato knihovna právě slouží pro jednoduché vykreslení grafů a histogramů.

Z výsledných dat, diferenciálního grafu a histogramu lze poté zjistit, jak linuxový operační systém pracoval. Pro kvalitnější porovnání výsledků je lepší toto měření provést vícekrát a pak všechna výsledná data porovnat. Z těchto výsledků je možné zjistit, zda je systém dostačující pro běh aplikací.

Při implementaci byl využit verzovací systém Git pro lepší organizaci celého projektu. Zároveň jeho další výhodou byl snadný přesun zdrojových souborů mezi jinými linuxovými operačními systémy.

Tato práce je přínosná pro její využití v reálném prostředí, kde slouží pro výměnu výpočetních jednotek v autonomních systémech, které jsou vyvíjeny a používány partnerskou firmou. Osobním přínosem této práce bylo pracování s technologií Berkeley Packet Filter, neboť pro mě byla zcela nová. Právě z důvodu novosti práci považuji za dobrou zkušenost a zároveň také rozšíření mých obzorů o fungování linuxových operačních systémů.

Možné rozšíření tohoto systému by mohlo být zachytávání jiných systémových událostí a s ním spojené nalezení více informací o fungování linuxového operační systému při zátěži. S tímto rozšířením se také pojí i rozšíření zpracování těchto dat a jejich následné vyhodnocení a zobrazení v grafech. Dalším možným rozšířením by mohlo být, že by systém filtroval systémové události a zachytával by informace pouze o vybraném procesu a jeho podprocesech. Toto by mohlo vést ke zjištění, kdy daný proces pracuje pod nejvyšší zátěži a případnému odstranění tohoto problému.

Literatura

[1] ALONI, D. Cooperative linux. In: *Proceedings of the Linux Symposium*. 2004, sv. 2.

[2] CALAVERA, D. a FONTANA, L. *Linux Observability with BPF*. 1. vyd. O'Reilly Media, Inc., 2019. ISBN 1492050202.

[3] DESNOYERS, M. *Using the Linux Kernel Tracepoints* [online]. [cit. 2022-04-11]. Dostupné z: `https://www.kernel.org/doc/Documentation/trace/tracepoints.rst`.

[4] D.SC., P. F. a PH.D., H. M. *Computer Systems Performance Evaluation and Prediction*. 2. vyd. Digital Press, 2003. ISBN 1555582605.

[5] FEITELSON, D. G. *Workload Modeling for Computer Systems Performance Evaluation*. 1. vyd. Cambridge University Press, 2015. ISBN 1107078237.

[6] FOUNDATION, T. L. *Bpftool* [online]. [cit. 2022-04-13]. Dostupné z: `https://www.mankier.com/package/bpftool`.

[7] FOUNDATION, T. L. *EBPF* [online]. 2021 [cit. 2022-04-13]. Dostupné z: `https://ebpf.io/`.

[8] GREGG, B. *BPF Performance Tools*. 1. vyd. Addison-Wesley Proffesional, 2019. ISBN 0-13-655482-2.

[9] GREGG, B. Computing Performance: On the Horizon. In:. USENIX Association, červen 2021.

[10] HATRY, H. P. *Performance Measurement: Getting Results*. 2. vyd. Rowman & Littlefield Publishers, 2007. ISBN 0877667349.

[11] KENISTON, J., PANCHAMUKHI, P. S. a HIRAMATSU, M. *Kernel Probes (Kprobes)* [online]. [cit. 2022-04-11]. Dostupné z: `https://docs.kernel.org/_sources/trace/kprobes.rst.txt`.

[12] KERNEL.ORG. *BPF Type Format (BTF)* [online]. [cit. 2022-04-11]. Dostupné z: `https://www.kernel.org/doc/html/latest/_sources/bpf/btf.rst.txt`.

[13] KERNEL.ORG. *EBPF maps* [online]. [cit. 2022-04-11]. Dostupné z: `https://www.kernel.org/doc/html/latest/_sources/bpf/maps.rst.txt`.

[14] KERNEL.ORG. *Event Tracing* [online]. [cit. 2022-04-11]. Dostupné z: `https://www.kernel.org/doc/Documentation/trace/events.rst`.

[15] LILJA, D. J. *Measuring Computer Performance: A Practitioner's Guide*. 1. vyd. Cambridge University Press, 2000. ISBN 1555540775.

[16] LINUX, B. *Kernel Space Definition* [online]. February 8, 2005 [cit. 2022-04-11]. Dostupné z: http://www.linfo.org/kernel_space.html.

[17] LINUX, B. *User Space Definition* [online]. February 8, 2005 [cit. 2022-04-11]. Dostupné z: http://www.linfo.org/user_space.html.

[18] LOVE, R. *Linux Kernel Development.* 3. vyd. Addison-Wesley Proffesional, 2010. ISBN 0-672-32946-8.

[19] MAL'KOVSKII, S., SOROKIN, A. A., KOROLEV, S. P., ZATSARINNYI, A. a TSOI, G. Performance evaluation of a hybrid computer cluster built on IBM POWER8 Microprocessors. *Programming and Computer Software.* Springer. 2019, sv. 45, č. 6.

[20] MILLER, T. C. *Sudo(8) — Linux manual page* [online]. 2021 [cit. 2022-04-13]. Dostupné z: https://man7.org/linux/man-pages/man8/sudo.8.html.

[21] OBAIDAT, M. S. a BOUDRIGA, N. A. *Fundamentals of Performance Evaluation of Computer and Telecommunication Systems.* 1. vyd. John Wiley & Sons Inc, 2010. ISBN 0471269832.

[22] RICHMAN, G. S. *What is vmlinux.h?* [online]. 2021 [cit. 2022-04-13]. Dostupné z: https://www.grant.pizza/blog/vmlinux-header/.

[23] ROBERTSON, A. *Bpftrace* [online]. [cit. 2022-04-13]. Dostupné z: https://github.com/iovisor/bpftrace.

[24] VASUDEVAN, A. *The Linux Kernel HOWTO - Kernel Files Information* [online]. 2003 [cit. 2022-04-13]. Dostupné z: https://www.csse.uwa.edu.au/programming/linux/Linux-HowTo-9/Kernel-HOWTO/kernel_files_info.html#vmlinuz.

[25] WARD, B. *How linux works.* 2. vyd. No Starch Press, 2014. ISBN 1593275676.

Příloha A

Příkazy

Tato sekce obsahuje potřebné příkazy pro vytvoření různých částí systému.

A.1 Přeložení sledovací aplikace

```
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_C_COMPILER=clang
$ sudo make
```

A.2 Přeložení knihovny libevaluator pro evaulační aplikaci

```
$ mkdir build
$ cd build
$ cmake ..
$ sudo make
```